

Some thoughts on Trivium

Steve Babbage
Vodafone Group R&D, Newbury, UK
steve.babbage@vodafone.com

19th January 2007

Abstract: This paper pulls together some thoughts about how the Trivium stream cipher might be attacked. It does not contain a successful attack, but I thought it was worthwhile sharing these thoughts with others, in the hope that they may be able to take them further. Observations from other researchers are presented, together with some of my own.

Keywords: Trivium, stream cipher, eStream, cryptanalysis, algebraic cryptanalysis, solving equation systems

1 Introduction

Trivium is a hardware-oriented stream cipher submitted to eStream by Bart Preneel and Christophe de Cannière. It allows for highly parallelised implementations, making it probably the best performing hardware submission where speed is of interest.

Trivium is a very simple design with apparently low security margin. This is quite deliberate on the part of the designers, who say “we strongly discourage the use of Trivium at this stage”.

This paper is about the potential cryptanalysis of Trivium. It does not contain very significant original results; instead it collects some of my thoughts and some observations by others, and presents them in a way that we hope will inspire further cryptanalytical work.

2 Trivium specification

The official specification is here:

http://www.ecrypt.eu.org/stream/p2ciphers/trivium/trivium_p2.pdf. It’s only 7 pages and easy to follow. There’s also a fuller paper at

https://www.cosic.esat.kuleuven.ac.be/ecrypt/intern/STVL/upload/All/STVL1-IAIK-28-Trivium-1_0.pdf.

Dan Bernstein and others have re-written the Trivium equations in a way that is perhaps slightly easier to understand (<http://www.ecrypt.eu.org/stream/phorum/read.php?1,448>):

$$x[n] = z[n-66] + z[n-111] + z[n-110]z[n-109] + x[n-69]$$

$$y[n] = x[n-66] + x[n-93] + x[n-92]x[n-91] + y[n-78]$$

$$z[n] = y[n-69] + y[n-84] + y[n-83]y[n-82] + z[n-87]$$

$$\text{and an output bit } o[n] = z[n-66] + z[n-111] + x[n-66] + x[n-93] + y[n-69] + y[n-84].$$

The recursion starts from $x[-1], x[-2], \dots, x[-93]$; $y[-1], y[-2], \dots, y[-84]$; $z[-1], z[-2], \dots, z[-111]$. (The TRIVIUM specification calls these $s[1], s[2], \dots, s[93]$; $s[94], s[95], \dots, s[177]$; $s[178], s[179], \dots, s[288]$.) TRIVIUM allows many of these bits to be an attacker-controlled nonce but compensates by dropping the first 1152 bits of output.

So note that:

- the output bit itself is derived linearly from the state;
- the state update function is non-linear (degree 2);
- by deliberate design, to allow faster parallelised implementations, the three newly computed state bits $x[n]$, $y[n]$ and $z[n]$ are not used as input to any calculations for the next 66 clockings of the register. (As one specific consequence of this, given a state of the register at a given time, the next 66 output bits are all linear combinations of those state bits.)

3 Approaches to cryptanalysis

The obvious cryptanalytic approaches involve creating a set of equations and then somehow trying to solve them. I see two most promising approaches:

1. Consider a single 288-bit register state at some point during the generation of keystream. Let these be our 288 unknown variables. Express (known) keystream bits as equations in these variables, simplify and solve.
2. Start with an initial set of 288 variables as above. But each time we clock the register, add three new variables $x[n]$, $y[n]$ and $z[n]$ (so the number of variables grows). Derive equations in the variables from the register update equations and the keystream output equation.

With approach 1, the number of variables is smaller but the individual equations become more complex. With approach 2, the number of variables is larger but the equations are simpler (sparse and of degree ≤ 2).

3.1 *Keeping the number of variables fixed*

I have just one simple observation here. A simplistic approach would be as follows:

- let the 288 unknown variables be the register state values immediately before keystream is generated;
- clock the registers 66 times; the keystream bits generated are all linear combinations of our variables; meanwhile new register bits are produced that are all degree-2 combinations of our variables;
- clock the registers 66 more times; the keystream bits generated are all degree-2 combinations of our variables; meanwhile new register bits are produced that are almost all degree-4 combinations of our variables;
- clock the registers 66 more times; the keystream bits generated are all degree-4 combinations of our variables; meanwhile new register bits are produced that are almost all degree-8 combinations of our variables;
- etc, producing ever higher degree equations.

However, we can do slightly better. Don't take the variables as being the register state at the beginning of the known keystream sequence; instead, take them as being the register

state somewhere part way through the known keystream sequence. That way the set of equations generated can be of smaller maximum degree.

This only helps a little, though. When the generator is clocked backwards, we don't have the 66-bit delay before any newly generated bit is used as input to a calculation. So the degree of the equations grows much faster when we clock the generator backwards.

3.2 **Letting the number of variables grow**

Start with 288 variables. Then, for each clocking of the generator:

- add three new variables $x[n]$, $y[n]$ and $z[n]$;
- add three new sparse degree-2 equations for generator update, i.e. the equations used to compute $x[n]$, $y[n]$ and $z[n]$;
- add another new equation, linear this time, for the new output keystream bit.

After k clocks of the generator this gives $3k$ degree-2 equations and k linear equations in $288+3k$ unknowns. But then we can also obtain 66 more linear equations for free, without introducing any more unknowns, by looking at the next 66 keystream bits.

So we have $3k$ degree-2 equations and $66+k$ linear equations in $288+3k$ unknowns. (For instance, putting $k=222$, we can have 954 equations in 954 unknowns; of the equations, 288 are linear and the rest are sparse degree-2 equations.)

We can of course easily eliminate the linear equations, to obtain $3k$ degree-2 equations in $222+2k$ unknowns. The equations become less sparse, though.

3.2.1 **Try to solve sparse equations of degree ≤ 2**

We can look for methods of solving these equations. Håvard Raddum tries this in <http://www.ecrypt.eu.org/stream/papersdir/2006/039.ps>, for instance. It may be possible to find some dedicated method of solving these very sparse, particular format, low degree equations.

3.2.2 **Treat them as noisy linear equations¹**

This is my own idea, not mentioned elsewhere as far as I know.

If you look back to section 2, you'll see that the degree-2 equations are of the form

$$bit_1 = (\text{XOR}_2 \text{ of bits}) \oplus bit_3.bit_4$$

Even if linear equations are eliminated, we end up with degree-2 equations of the form

$$(\text{XOR}_1 \text{ of bits}) \oplus (\text{XOR}_2 \text{ of bits}).(\text{XOR}_3 \text{ of bits}) = 0$$

And of course this is strongly correlated to a linear equation:

$$(\text{XOR}_1 \text{ of bits}) = 0 \quad \text{with probability } 0.75$$

¹ Acknowledgement: I would like to express my thanks to an anonymous SASC reviewer, whose comment alerted me to a silly mistake in the original version of this paper.

So we have a system of noisy linear equations. Are there any good techniques for solving this system? Some sort of iterated error correction method, for instance, such as the ones used in fast correlation attacks on stream ciphers?

We can do a quick information theoretic analysis:

- the error in one of our noisy linear equations is a boolean variable equal to 1 with probability 0.25;
- the entropy of this variable is $\eta=0.811$;
- so, speaking loosely, the noisy linear equation gives us $\theta = 1-\eta = 0.189$ bits of information about the linear term.

We observed earlier that we can obtain $3k$ degree-2 equations in $222+2k$ unknowns. The bad news is that, for any positive k , $3k\theta < 222+2k$. So on its own this does not appear to produce a soluble system. But perhaps this approach will be useful in conjunction with some other idea.

Note that one interpretation of this set of noisy linear equations is as a decoding problem in a linear code. Given the value of the unknowns, the non-noisy linear equations give us a codeword, and the noisy linear equations give us a received word with errors. However, the observation above that $3k\theta < 222+2k$ means that we will not generally be able to decode with an error rate of 25%.

3.2.3 Guess some bits

Shahram Khazaei makes a nice observation in a reply posted at <http://www.ecrypt.eu.org/stream/phorum/read.php?1,448>. By guessing some unknowns, we can turn degree-2 equations into linear equations. So let's assume that our known keystream sequence starts when the generator contains the bits $x[t-1], x[t-2], \dots, x[t-93]$; $y[t-1], y[t-2], \dots, y[t-84]$; $z[t-1], z[t-2], \dots, z[t-111]$.

Then guess the 45 alternate bits $x[t-91], x[t-89], x[t-97], \dots, x[t-3]$; guess the 45 alternate bits $y[t-82], y[t-80], y[t-78], \dots, y[t+6]$; and guess the 45 alternate bits $z[t-109], z[t-107], z[t-105], \dots, z[t-21]$.

We then obtain a set of linear equations (here we write particular bits in italics to show that they are guessed, i.e. known):

$$\begin{aligned}
 x[t] &= z[t-66] + z[t-111] + z[t-110]z[t-109] + x[t-69] \\
 x[t+1] &= z[t-65] + z[t-110] + z[t-109]z[t-108] + x[t-68] \\
 x[t+2] &= z[t-64] + z[t-109] + z[t-108]z[t-107] + x[t-67] \\
 x[t+3] &= z[t-63] + z[t-108] + z[t-107]z[t-106] + x[t-66] \\
 &\dots \\
 x[t+89] &= z[t+23] + z[t-22] + z[t-21]z[t-20] + x[t+20] \\
 \\
 y[t] &= x[t-66] + x[t-93] + x[t-92]x[t-91] + y[t-78] \\
 y[t+1] &= x[t-65] + x[t-92] + x[t-91]x[t-90] + y[t-77] \\
 &\dots
 \end{aligned}$$

$$y[t+89] = x[t+23] + x[t-4] + x[t-3]x[t-2] + y[t+11]$$

$$z[t] = y[t-69] + y[t-84] + y[t-83]y[t-82] + z[t-87]$$

$$z[t+1] = y[t-68] + y[t-83] + y[t-82]y[t-81] + z[t-86]$$

...

$$z[t+89] = y[t+20] + y[t+5] + y[t+6]y[t+7] + z[t+2]$$

Then, from the set of 558 variables $x[t-93] \dots x[t+89]$, $y[t-84] \dots y[t+89]$, $z[t-111] \dots z[t+89]$:

- $3 \times 45 = 135$ bits have been guessed ;
- we have 270 linear equations from the generator update equations, as shown above ;
- we have 90 linear equations from the keystream bits being output while those generator updates were taking place;
- we get the usual 66 free linear equations from the next 66 keystream bits.

Overall we have more linear equations than unknowns, and can solve the system.

Of course we have guessed 135 bits here, and the key size of Trivium is only 80 bits, so this isn't a successful attack. But it illustrates a guess-and-determine technique that may be useful in developing an attack.

More generally, if k is the number of clocks of the generator, suppose that we guess g well chosen bits, in the way that Khazaei proposes, so that degree-2 equations become linear. Each guess convert two degree-2 equations into linear ones, as well as reducing the number of unknowns by one.

Let n_2 be the number of degree-2 equations, n_1 be the number of linear equations, and u the number of unknowns. Without this guessing, we have:

$$n_2 = 3k; \quad n_1 = 66+k; \quad u = 288+3k.$$

Guessing g well chosen bits we have

$$n_2 = 3k-2g; \quad n_1 = 66+k+2g; \quad u = 288+3k-g; \quad \text{and note the constraint } 3k \geq 2g.$$

Putting $k = 90$ and $g = 135$, as in Khazaei's full guessing attack, we have

$$n_2 = 0; \quad n_1 = 426; \quad u = 423; \quad \text{and clearly the system is soluble.}$$

4 Conclusions

Trivium is a very simple design with apparently low security margin. It is easy to generate quite small systems of low degree equations, such that solving these systems would suffice to break Trivium. I encourage people to take inspiration from the ideas collated in this paper and try to develop attacks.