§6.5 : Baby CSS.

It turns out that the main vulnerability of the LFSR and LFSR sum methods just introduced (at least, the main vulnerability when it comes to known plaintext attacks) is linearity of the system. An attacker has powerful algebraic techniques (such as row reduction / linear algebra) at their disposal to reverse engineer everything.

Our fix: Introduce some "non linearity" that preserves the simplicity of encryption but makes the linear algebra / methods used in the known plaintext attack fail.

The trick for Baby CSS is essentially:

Replace XOR in LFSR sum with binary addition.

I.e.

Instead of this:

$$\left.\begin{array}{r} 101001 \\ \oplus\, 011001 \\ \hline 110000 \end{array}\right\} \text{XOR}$$

we have to carry some 1's:

$$\begin{array}{r} {\scriptstyle 1\,1\;\;\;\;1\;\;} \\ 101001 \\ +\, 011001 \\ \hline 1000010 \end{array}$$

§6.5 : Baby CSS.

It turns out that the main vulnerability of the LFSR and LFSR sum methods just introduced (at least, the main vulnerability when it comes to known plaintext attacks) is linearity of the system. An attacker has powerful algebraic techniques (such as row reduction / linear algebra) at their disposal to reverse engineer everything.

Our fix: Introduce some "non linearity" that preserves the simplicity of encryption but makes the linear algebra methods used in the known plaintext attack fail.

The trick for Baby CSS is essentially:

Replace XOR in LFSR sum with binary addition.

I.e.

Instead of this:

$$\left.\begin{array}{r} 101001 \\ \oplus\ 011001 \\ \hline 110000 \end{array}\right\} \text{XOR}$$

we have to carry some 1's:

$$\begin{array}{r} \scriptstyle 1\,1\quad\;\; 1\;\; \\ 101001 \\ +\ 011001 \\ \hline 1000010 \end{array}$$

However we'll also chunk the keystreams into blocks of length 5 when summing in this way (this fixes the issue of having to add right-to-left when carrying, while wanting to generate a keystream left-to-right).

Baby CSS Algorithm:

- Choose a 3-bit LFSR and a 5-bit LFSR. As before, assume the rightmost bit in the seed of each is 1, to avoid the possibility of accidentally seeding with the zero.
- Generate the LFSR-3 and LFSR-5 keystreams. Chunk them into blocks of 5, and add (not XOR). Carry leftover ones (at the end of the block) to the next block. This generates the Baby CSS keystream.
- XOR the keystream with the plaintext to get the ciphertext.
- To decrypt, generate the keystream again and XOR it with the ciphertext.

Example:

Suppose LFSR-3 is $b_3 = b_2' + b_1'$ and

LFSR-5 is $c_5 = c_4' + c_2' + c_1'$.

Seed LFSR-3 with 0 1 $\underline{1}$ — we insist on this 1

Seed LFSR-5 with 1010 $\underline{1}$

Run the registers for a few iterations:

LFSR-3

step	string		
0	0	1	1
1	0	0	1
2	1	0	0
3	0	1	0
4	1	0	1
5	1	1	0
6	1	1	1
7	0	1	1
			1
			0
			0
			⋮
			etc

LFSR-5

step	string				
0	1	0	1	0	1
1	1	1	0	1	0
2	0	1	1	0	1
3	0	0	1	1	0
4	1	0	0	1	1
5	0	1	0	0	1
6	0	0	1	0	0
7	0	0	0	1	0
8	1	0	0	0	1
9	1	0	0	0	0
10	0	1	0	0	0
					⋮
					etc.

Add the keystreams in blocks, with carrying (adding in binary)

carry to next block

	1			1		1	1	1	1		1
LSFR-3	1	1	0	0	1		0	1	1	1	0
LSFR-5 +	1	0	1	0	1	+	1	0	0	1	0
BabyCSS	0	1	1	1	0		0	0	0	0	1

leftover, carried to next block <u>or</u> discarded if this is the last block we need.

Now if our message is:

$$00111 \quad 01000$$

then we create the ciphertext via:

$$\begin{array}{rccccccccccc}
 & 0 & 0 & 1 & 1 & 1 & & 0 & 1 & 0 & 0 & 0 \\
\text{XOR} \longrightarrow \oplus & 0 & 1 & 1 & 1 & 0 & & 0 & 0 & 0 & 0 & 1 \\
\hline
\underline{\text{ciphertext}} \rightarrow & 0 & 1 & 0 & 0 & 1 & & 0 & 1 & 0 & 0 & 1
\end{array}$$

Since we used XOR in this last step, we recover the plaintext by XORing the keystream with the ciphertext (it works since addition and subtraction are the same mod 2).

<u>Breaking Baby CSS</u> with a known plaintext attack (§ 6.6).

Let's see if our same technique used to break the LFSR-sum cipher using a known plaintext attack will work.

We proceed as in the case of LFSRsum. We don't know the seed for the 3-bit LFSR, but assume we <u>do</u> know the formula $b_3 = b_2' + b_1'$ and that the seed's rightmost bit is '1'. So its keystream will be:

(this is identical to before):

(use a seed of $k_1 k_2 1$)

LFSR - 3

step	string		
0	k_1	k_2	1
1	$1+k_2$	k_1	k_2
2	k_1+k_2	$1+k_2$	k_1
3	$1+k_1+k_2$	k_1+k_2	$1+k_1$
		$1+k_1+k_2$	k_1+k_2
			$1+k_1+k_2$
			$1+k_1$
			1
			$\vdots$
			repeats.

Similarly, using a seed of $k_3\ k_4\ k_5\ k_6\ 1$ in LFSR-5 gives a keystream:

step	String				
	k_3	k_4	k_5	k_6	1
					k_6
					k_5
					k_4
					k_3
					$1+k_6+k_4$
					$k_6+k_5+k_3$
					$\vdots$
					etc.

For LFSR-sum, we needed to know the first 6 bits of the plaintext and the first 6 bits of the ciphertext in order to get a 6×6 linear system that allowed us to solve everything (it all fell apart). Let's see what happens here.

Suppose we know the first 6 bits of the plaintext and the first 6 bits of the ciphertext, and we XOR them to get the first 6 bits of the keystream. It turns out to be

$$0\ 1\ 0\ 0\ 0\ 1.$$

Recall that we added in blocks of 5 to get this. So, we know:

LFSR-3	1	k_2	k_1	$1+k_1$	k_1+k_2	$1+k_1+k_2$	....
LFSR-5	1	k_6	k_5	k_4	k_3	$1+k_6+k_4$	....
keystream:	0	1	0	0	0	1	....

First equation: $k_1+k_2+k_3 = 0 \bmod 2.$

Next equation: $1+k_1+k_4+\underbrace{(\text{possibly a 1 carried from first sum})}_{\text{how to deal with this?}} = 0 \bmod$

Well, the only way we ended up carrying a '1' is if k_1+k_2 and k_3 are both 1.

A clever way of writing the carried one, then, is:

$$(k_1+k_2)\cdot k_3 .$$

Note that this product is 1 exactly when k_1+k_2 and k_3 are both 1, and zero otherwise. So we can cleverly write the next equation as:

<u>Second eqn</u>: $1+k_1+k_4+\underbrace{(k_1+k_2)\cdot k_3}_{\text{non linear!}} = 0 \bmod 2.$

<u>Third eqn</u>:

$$k_1+k_5+(\text{possibly a 1 carried from previous eqn}) = 0 \bmod 2$$

Well, a 1 is carried from the previous equation if two or more of the terms $1+k_1$, k_4, and $(k_1+k_2)\cdot k_3$ are equal to 1. So consider:

$$E = (1+k_1)\cdot k_4 + k_4\cdot(k_1+k_2)\cdot k_3 + (1+k_1)\cdot(k_1+k_2)\cdot k_3 \bmod 2.$$

This is 1 exactly if two (or all three) of $1+k_1$, k_4, and $(k_1+k_2)\cdot k_3$ are equal to 1. So eqn 3 is:

$$k_1+k_5+E = 0 \bmod 2$$

very complicated nonlinear.

And subsequent equations only get worse.

Comparison with "real" CSS

content scramble system
not cascading style sheets

- "Real" CSS uses a 17-bit and 25-bit register.
- The rightmost bit of each seed is not 1, instead the fourth bit is 1.
- Bits are added in chunks of size 8 (ie bytes), with carrying as described.
- XOR with plaintext to get ciphertext.

Why is "real" in quotes? This is a bare-bones description of CSS, and while the algorithm above is at the core, implementation is a bit more complicated

---

§ 6.6 ~~cont'd~~: Known plaintext and BabyCSS cont'd.

BabyCSS is not that secure, since keys are only 6 bits. So there are only $2^6 = 64$ keys, meaning brute force will work fine.

Breaking "real" CSS by brute force ~~has~~ is a bit more of a problem since there are $2^{40} \approx 1 \times 10^{12}$ keys. However it is still doable. With a known plaintext attack, we can reduce BabyCSS to $2^2 = 4$ keys one needs to check, and "real" CSS to 2^{16} $= 65536$ keys that one needs to check. Let's see how this works for BabyCSS, and call our study of CSS done:

Suppose that someone is using BabyCSS to send messages. We also know the first bits of the plaintext, which we XOR with our eavesdropped ciphertext to get the start of the keystream they're using. It is:

01001 1 1 1 1 0

Suppose that we guess 001 for the LFSR-3 seed, the only other guesses are 011, ~~0~~11, 101. We'll show how this single guess allows us to generate the rest of the keystream. With our guess:

LFSR-3

step	string		
0	0	0	1
1	1	0	0
2	0	1	0
			1
			0
			1
			1
			1
			0 ... etc.

fill in the table.

Then we know that if the LFSR-5 keystream is

$$k_1\ k_2\ k_3\ k_4\ k_5 \quad k_6\ k_7\ k_8\ k_9\ k_{10}\ \dots \text{etc},$$

the k's all 0 or 1

Then each block satisfies:

$$
\begin{array}{lcccccccccccc}
\text{LFSR-3} & & 1 & 0 & 0 & 1 & 0 & & 1 & 1 & 1 & 0 & 0 \\
\text{LFSR-5} & + & k_1 & k_2 & k_3 & k_4 & k_5 & + & k_6 & k_7 & k_8 & k_9 & k_{10} \\
\hline
\text{BabyCSS Keystream} & & 0 & 1 & 0 & 0 & 1 & & 1 & 1 & 1 & 1 & 0
\end{array}
$$

where addition is simply regular binary addition, with extra 1's carried to the next block. So to find the LFSR-5 keystream, subtract <u>with borrowing from the next block</u>.

<u>Sidebar</u>: I know it's insulting to review subtraction. This is <u>not</u> about reviewing how to subtract, it's about reviewing how ~~the~~ algorithm is written and how "borrowed 1's" are tracked. People learn differently depending on where they did school.

Recall:

<u>In decimal</u>:

$$
\begin{array}{rrrr}
\cancel{1} & \overset{9}{\cancel{0}} & \overset{9}{\cancel{0}} & {}^{1}0 \\
- & & & 6 \\
\hline
 & 9 & 9 & 4
\end{array}
$$

<u>In binary</u>:

$$
\begin{array}{rrrr}
\cancel{1} & \overset{1}{\cancel{0}} & \overset{1}{\cancel{0}} & {}^{1}0 \\
 & & - & 1 \\
\hline
 & 1 & 1 & 1
\end{array}
$$

It works the same in binary.

BabyCSS keystream	¹0	X	¹Ø	¹0	1	1	1	1	X	¹Ø
LFSR-3	− 1	0	0	1	0	− 1	1	1	0	0
LFSR-5	1	0	1	1	1	0	0	0	0	1.

Q: Where did this get borrowed from? A: Here

So, for each guess of the seed for LFSR-3, a known plaintext attack allows us to reconstruct the LFSR-5 keystream. Then we do a known-plaintext attack on the LFSR-5 register to get its key. In this case, it's

1 0 1 1 1

(Recall: if you know the start of the keystream for a single register, you know the seed).

Therefore, to use known plaintext against BabyCSS, we only need to try all four "keys" of the LFSR-3 register (solving for the key of the LFSR-5 register in each case).

For "real" CSS, we can similarly try all 2^{16} "keys" of the LFSR-17 register. This is an amount that is easy to handle on a modern computer, so a known-plaintext attack involves 2^{16} cases.

Ch 7 : Public - channel cryptography

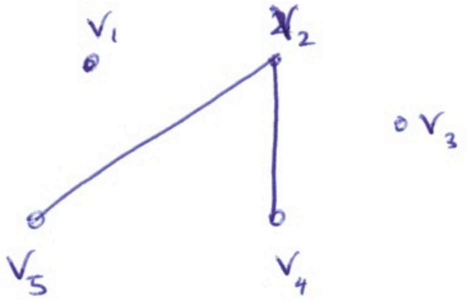
We have thusfar studied symmetric ciphers: Systems where knowing the key used to encrypt the message is enough to decrypt it.

Public-channel cryptography refers to cryptosystems where two people are able to send secure messages to one another despite only communicating in public channels. These systems are generally based on mathematical problems where it's easy to "make the problem", but hard to solve it. Let's see an example.

§7.1 Perfect codes and graphs.

A graph is a collection of vertices V and a set of edges E, where an edge is a set of two vertices $\{v_1, v_2\}$, with $v_1, v_2 \in V$.

E.g.



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{\{v_2, v_4\}, \{v_2, v_5\}\}$.

In our discussion we don't want edges like:

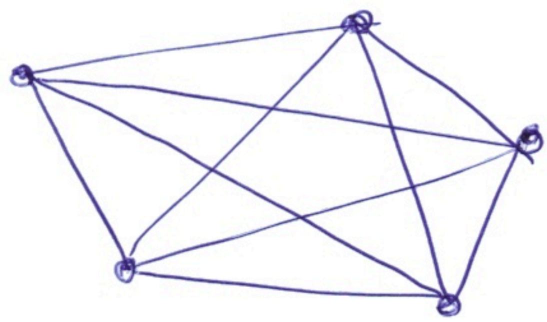
, so sets $\{v, v\}$ are not permitted as edges

We also do not allow

so the set $\{v_1, v_2\}$ (for a given v_1, v_2) can only occur <u>once</u> in E.

When $\{v_1, v_2\} \in E$ we say v_1 and v_2 are <u>adjacent</u>. The number of edges containing a vertex v is called the <u>degree</u> of v.

<u>E.g.</u>



This is a graph where all vertices are adjacent and all have degree 5.

<u>Definition</u>: Given a graph with vertices V and edges E, a <u>perfect code</u> on the graph is a subset $V_{PC} \subset V$ of vertices such that:

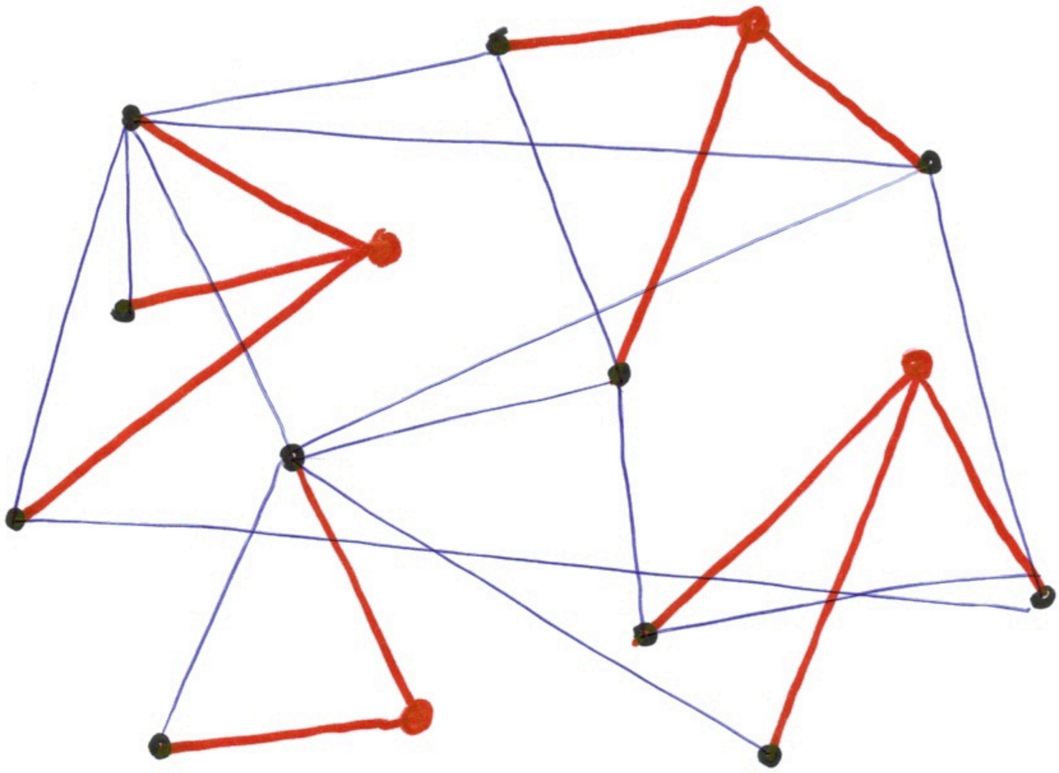
(1) If $v_1, v_2 \in V_{PC}$ then $\{v_1, v_2\} \notin E$
(ie no two vertices in the perfect code are adjacent)

(2) If $v \in V$ and $v \notin V_{PC}$, then there is a unique $v' \in V_{PC}$ with $\{v, v'\} \in E$.
(ie. every vertex <u>not</u> in the perfect code is adjacent to exactly one vertex in the perfect code)

It's easy to make a graph that has an associated perfect code. Here's how:

1. Start with some vertices. (In black)
2. Pick a few arbitrary vertices, and connect every vertex to exactly one of our choices. These vertices are our perfect code. (In red)
3. Don't stop there! Add a bunch of ~~vertices~~ edges between vertices not in the perfect code so that nobody knows your choice of perfect code (In blue pen)

Imagine trying to find the perfect code below without colors to help!



Now I can create a graph with a perfect code that I know, and share it with everyone. The problem of finding the perfect code in my shared graph is (hopefully!) hard enough that nobody will figure out my secret.

Q: How to make a cryptosystem out of this?