Practical attacks against second generation GSM networks

Praktiske angreb på andengenerations GSM-netværk

Frederik M. Madsen - 201706759, Torkil A. Helgeland - 201209097
& Rune N. T. Thorsen - 201505509

Advisor: Ivan B. Damgaard

Bachelor thesis

Department of Computer Science
Aarhus University
Denmark

16th June 2020

Acknowledgements

We would like to thank our supervisor, Ivan, for helping us through this project. Not only did he relay a lot of valuable information to us and helped us understand and figure out a lot, it also means a lot to us, that he was willing to take us in and personally supervise us.

Rune would also like to extend a thank to Camilla A. Grevy for emotional support throughtout the lockdown period of the society during the project.

Abstract

In this thesis we will describe and analyze the A5/2 stream cipher used in second generation GSM networks. We will provide an implementation and argue that it is possible to greatly improve the performance of the cipher on a modern processor. The analysis reveals a fundamental security flaw in the cipher which we take advantage of to build a practical known plaintext attack. We then expand the known plaintext attack to a known ciphertext attack by exploiting a flaw in the error correction process used in the GSM network. We provide arguments for why these attacks are practical and can be efficiently executed.

Finally we discuss later developments in mobile communication cryptology and potential further investigations, specifically how the flaws of A5/2 have been corrected and more generally what could be done in order to expand our work.

Colophon

Practical attacks against second generation GSM networks

Bachelor thesis by Frederik M. Madsen, Torkil A. Helgeland
& Rune N. T. Thorsen

The project is supervised by Ivan B. Damgaard

Typeset by the authors using LaTeX and the memoir document class, using Linux Libertine and Linux Biolinum 11.0/14.41597pt.

Contents

1 Introduction

The 2G GSM network represents the transition from analog to digital telecommunication. Of course this transition also necessitated new digital cryptosystems which is where the A5/2 and A5/1 ciphers came into the picture. A5/1 and A5/2 are the different stream ciphers used to encrypt communication on the GSM network.

GSM provides a standard for call establishment, broadcast channels and more. GSM networks establish calls as can be read in appendix B.

1.1 Implementation

Our goal is to explain how exactly the A5/2 cipher, as used in GSM networks, works and how it can be implemented more efficiently on modern architectures than when it was originally designed. From our detailed explanation and examples on the inner workings of the cipher, its shortcomings and flaws should become clear. We take special note of how we have implemented the cipher using modern SIMD vector instructions.

The SIMD instruction are called through compiler intrinsics and can be differentiated from other functions by their `_mm` prefix. Should you require documentation for what these functions do, it can be found by searching for the function name on the Intel Intrinsics Guide.

1.2 The attack

The flaws we will exploit in our implementation of the known plaintext attack first described by (Barkan et al. 2008). Following their approach we will implement the attack as well as explain all the details of both the attack itself and our implementation, with focus in particular on how the attack works and what we do to obtain solutions for a guess on the value of $R4$ and knowing whether or not such a guess is correct.

Following the known plaintext attack we will analyze the known ciphertext attack, as originally presented by (Barkan et al. 2008). After explaining its details and inner workings we will cover our implementation of the attack as well as look briefly at its practicality and efficiency.

1.3 Further investigations

After explaining why A5/2 is broken by discussing two different attacks against we will discuss later developments within mobile cryptography. We will look what its successor, the KASUMI cipher, also called A5/3, has done to mitigate the weaknesses in its predecessor, additionally we will present some attacks on KASUMI. We will also briefly look at the replacement cipher used in the 4G LTE standard and its potential flaws.

Finally, we end with by outlining potential further investigations by presenting other areas pertaining to security in the GSM networks, which could relate to A5/2 and how security was handled previously in GSM. Examples of this include the attacks on KASUMI and the AES cipher being used to encrypt data on LTE networks.

2 The A5/2 stream cipher

The A5/2 cipher does not have a public standard, but it was reverse engineered by (Briceno et al. 1999). In this section we will first describe the cipher in general and then give an optimized implementation, taking advantage of modern processor architecture and the Streaming SIMD Extensions (SSE) instructions in x86.

2.1 Description of A5/2

2.1.1 Stream Ciphers

A5/2 is a stream cipher which in general is a type of symmetric encryption scheme where a shared secret key, K, is used to produce a much larger keystream utilizing some known keystream generation procedure, G. Given plaintext, p, with the bits, $p_1, p_2, \dots, p_n$, (this could for instance be digitized call data) and the generated keystream bits, $k_1, k_2, \dots, k_s$, encryption is then simply done by XOR'ing the corresponding bits:

$$E_K(p) = p_1 \oplus k_1, p_2 \oplus k_2, \dots, p_s \oplus k_s = c_1, c_2, \dots, c_s = c \tag{2.1}$$

Where c denotes the resulting ciphertext. It should be clear that anyone that knows K can run G to produce $k_1, k_2, \dots, k_s$ and then decrypt:

$$D_K(c) = c_1 \oplus k_1, c_2 \oplus k_2, \dots, c_s \oplus k_s = p_1, p_2, \dots, p_s = p \tag{2.2}$$

It might of course be the case that the plaintext is longer than the keystream produced by one run of G. Running this simple G, that only takes input K, repeatedly in order to get enough key bits is a security problem since the output from $G(K)$ will always be the same and therefore an eavesdropper could easily detect the same message being sent twice. To avoid the impracticality of constantly having to distribute new session keys, the input to G should therefore include some initialization vector, IV, that changes every time.

2.1.2 Shift Registers

In A5/2 the generation procedure is based on Linear Feedback Shift Registers (LFSR's) which are essentially bit arrays of some fixed length. We let Ri denote register i and $Ri[0], Ri[1], \dots, Ri[n-1]$ denote the bits in the array for Ri with length n. Some subsets of these bits are designated as feedback taps, which we denote by $tap_1, \cdots, tap_n$ or output taps.

When a LFSR is "clocked" all bits are shifted away from position 0, meaning that a new bit is inserted in $Ri[0]$ while an old bit is removed in the other end of the array (essentially performing a left shift with a mask for the n bits in the register). As the name suggests the bit being inserted is determined by a linear combination of some set of feedback taps, tap_i in the register. For A5/2 the linear operation used is XOR, and the feedback function $feedback$ as:

$$feedback(tap_1, \dots, tap_n) = tap_1 \oplus \dots \oplus tap_n \tag{2.3}$$

Let $value = feedback(tap_1, \cdots, tap_n)$ be the value of the feedback function on the taps $tap_1, \dots, tap_n$ on some register RA. Let $a << n$ denote a logical left shift of a value a, n times, shifting in zeroes and discarding any bits that overflow. We then define clocking the register RA as:

$$\begin{aligned} RA &= RA << 1 \\ RA[0] &= value \end{aligned} \tag{2.4}$$

We will see later in this section that this linearity makes the implementation of the cipher very easy and efficient, but we will also see that for encryption, linearity is not a nice property. The linearity of the feedback function (among other flaws) can be exploited and is what allows us to efficiently break the cipher.

2.1.3 A5/2

Specifically the input to the A5/2 keystream generation procedure is a 64-bit secret session key, K_c, and a 22-bit public frame number, f, which functions as the IV. The inner workings of A5/2 is based on exactly four linear feedback shift registers (abbreviated LFSR, we will refer to them simply as registers): $R1$, $R2$, $R3$ and $R4$ with lengths of 19, 22, 23 and 17 bits respectively.

The first three register $R1$ through $R3$ are clocked according to the values of the latter, $R4$. The generation procedure starts by initializing the registers using K_c and f as shown in the pseudocode in fig. 2.1.

```
R1,R2,R3,R4 = 0;
for(i = 0; i < 64; i++) {
    clock_all_registers();
    R1[0] = R1[0] XOR K_c[i];
    R2[0] = R2[0] XOR K_c[i];
    R3[0] = R3[0] XOR K_c[i];
    R4[0] = R4[0] XOR K_c[i];
}
for(i = 0; i < 22; i++) {
    clock_all_registers();
    R1[0] = R1[0] XOR f[i];
    R2[0] = R2[0] XOR f[i];
    R3[0] = R3[0] XOR f[i];
    R4[0] = R4[0] XOR f[i];
}
R1[15] = 1;
R2[16] = 1;
R3[18] = 1;
R4[10] = 1;
```

FIGURE 2.1: Key setup algorithm for A5/2.

It then runs for 327 cycles each outputting one bit - the first 99 of these bits are discarded and the last 228 are output as the keystream. Figure 2.2 shows an example of one such cycle.

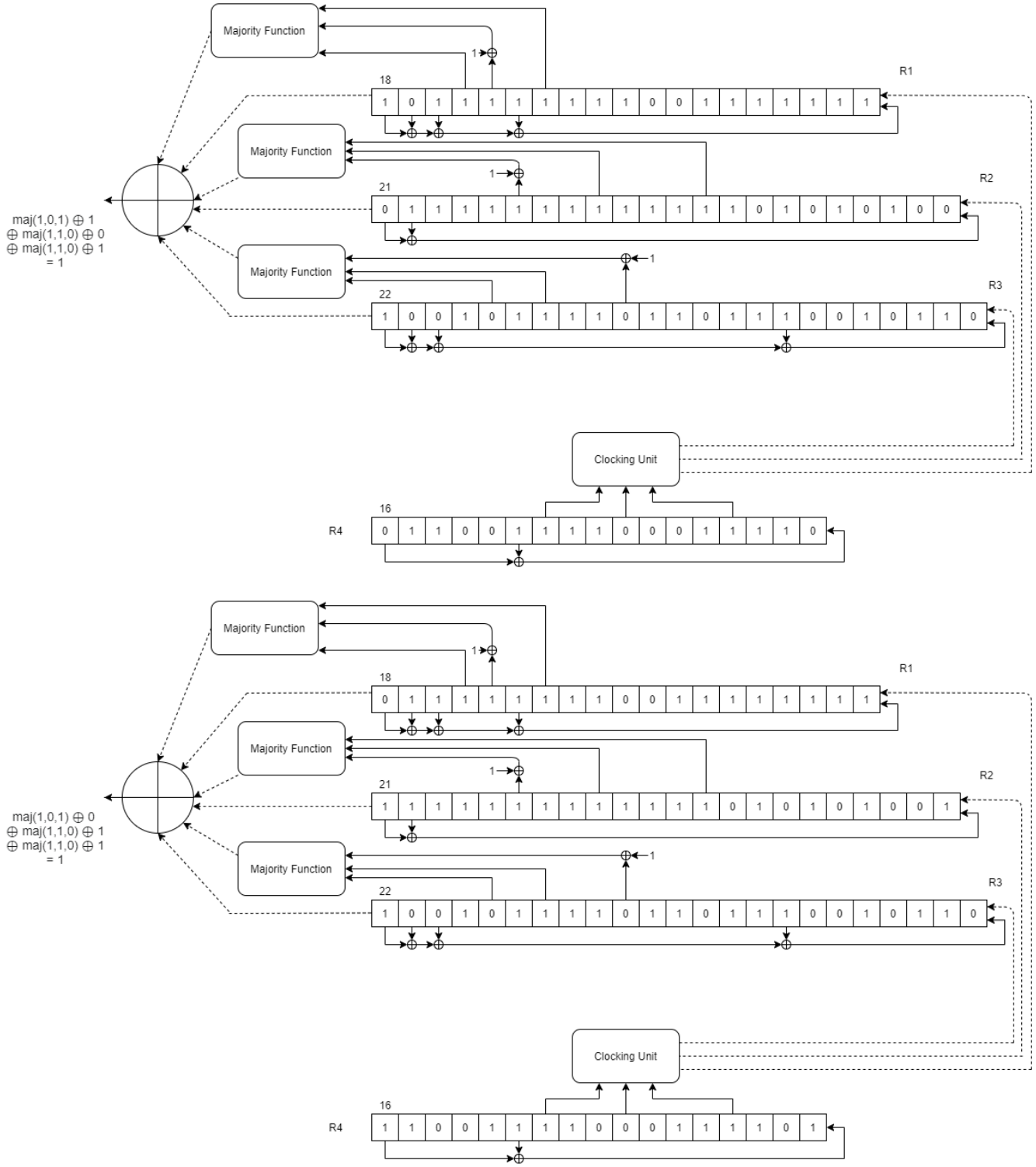

FIGURE 2.2: Example of one full cycle inspired by (Barkan et al. 2008, Fig. 1).

For each of the four registers we define a set of output taps, that are fed into the feedback function when clocking the register. Each register also has a loaded bit, which is set to one after the frame number has been clocked into it. Additionally for the first three registers $R1, \cdots R3$ we define a set of outputs taps, and a control tap. These are listed in section 2.1.3.

Register	Feedback taps	Output taps	Control tap	Loaded bit
R1	13,16,17,18	12,$\overline{14}$,15	10	15
R2	20,21	9,13,$\overline{16}$	3	16
R3	12,14,15,16	$\overline{13}$,16,18	7	18
R4	11,16			10

FIGURE 2.3: Feedback and output taps for the different registers.

During the setup we always clock all four of the registers however, when generating the output bits we only always clock $R4$. For each register, we define a control tap in $R4$, if the value of these control taps agree with the majority of all three control taps the register will be clocked. The three other registers are only clocked, if their control tap is equal to the majority of the controls taps of $R4$. We define the majority function as:

$$majority(a, b, c) = ab \oplus ac \oplus bc \tag{2.5}$$

Given that there are three control taps, we know that at least two of them will agree with the majority of all three. Suppose that the majority of the three control taps is 1, then we know that at least two of the taps must also be 1. Because of this, we will always clock at least two of the registers during a cycle.

The clocking strategy of A5/2 also constitutes the primary difference between it and A5/1, since A5/1 has the same basic architecture, but lacks the $R4$ register. Instead the clocking in A5/1 is determined based on the values of $R1$, $R2$ and $R3$. This is a more secure approach. The reasons for which will become clear later, but the headline is that having the clocking depend on the values of all three register, vastly increases the number of bits we need to guess to be able to know exactly how everything is clocked, compared to the 17 bits of $R4$.

When we need to generate an output bit for the keystream, we XOR together the last bit of each of the registers $R1$ through $R3$. We then XOR this value together with the majority values for the outputs taps of each of the registers. We define the output functions as (where $\overline{b}$ denotes the complement of b):

$$\begin{aligned} output() = (R1[18] \oplus R2[21] \oplus R3[22]) &\oplus majority(R1[12], \overline{R1[14]}, R1[15]) \\ &\oplus majority(R2[9], R2[13], \overline{R2[16]}) \\ &\oplus majority(\overline{R3[13]}, R3[16], R3[18]) \end{aligned} \tag{2.6}$$

Finally, to generate the output keystreams we would clock the registers, as described earlier, according to the value of $R4$, we then generate an output using the $output()$ function. As previously mentioned we do this a total of 228 times - the first 114 generate the bits for the incoming keystream and the next 114 generate the bits for the outgoing keystream.

2.2 Implementation of A5/2

Our implementation of the A5/2 cipher largely follows that of (Briceno et al. 1999) with the primary modifications being how the calculations are carried out and how the registers are represented to facilitate these calculations. We also use their examples of valid inputs and outputs to test our implementation.

Each register is represented by a 32-bit unsigned integer, u, with the least significant bit corresponding to $Ri[0]$. Of course all of our registers are shorter than 32 bits meaning none of them will take up all the bits in u, we will set the excess bits in u to zero. Seeing as we can fit

each register into a 32-bit integer, we can also fit all four registers sequentially into a 128-bit value. We do this by packing them sequentially into the 128-bit value as $R4|R3|R2|R1$ where $|$ denotes concatenation. We start with $R1$ at the least significant bits and end with $R4$ in the 32 most significant bits. In C we can create a union type corresponding to this representation:

```
typedef union {
    __m128i V;
    u32 R[4];
    u08 B[16];
} u32_4x;
```

This allows us to use modern SIMD (single instruction, multiple data) instructions to perform the calculations on all registers simultaneously, where our implementation primarily uses instructions provided by the SSE x86 extensions. However, for simplicity we will stick to describing the ideas behind the calculations, as if they were performed on a single 32-bit register at a time in the following explanation. The distinction between this approach and using the SIMD instructions, is simply that we can perform calculations on all 4 registers, in parallel at the same time.

Ideally, this would mean that we can speed up most of the computations by a factor of four, since we can compute what would take at least four instructions, if done individually on each register, in just one calculation using a single SIMD instruction. We of course need to add a disclaimer, that there is much more to optimizing code than just using more efficient instructions, though we believe that by utilizing these we end up with a faster implementation, than had we used regular instructions to compute things sequentially rather than in parallel.

Generally we can access specific bits by applying suitable masks and using shift operations. Particularly interesting is of course the mechanics of clocking a register represented by a 32-bit unsigned integer. In fig. 2.4 we go through the steps of computing the new incoming feedback bit, when clocking the register $R1$: first we compute the bitwise AND of the integer representing R1 and the mask for the feedback taps. Then we count the 1's in the result - the only reason for this is to figure out if this number is odd or even, since if the number of 1's in the feedback taps is odd (and hence having a 1 in position 0) then XORing them together equals 1, if instead it is even, then of course the feedback bit is 0.

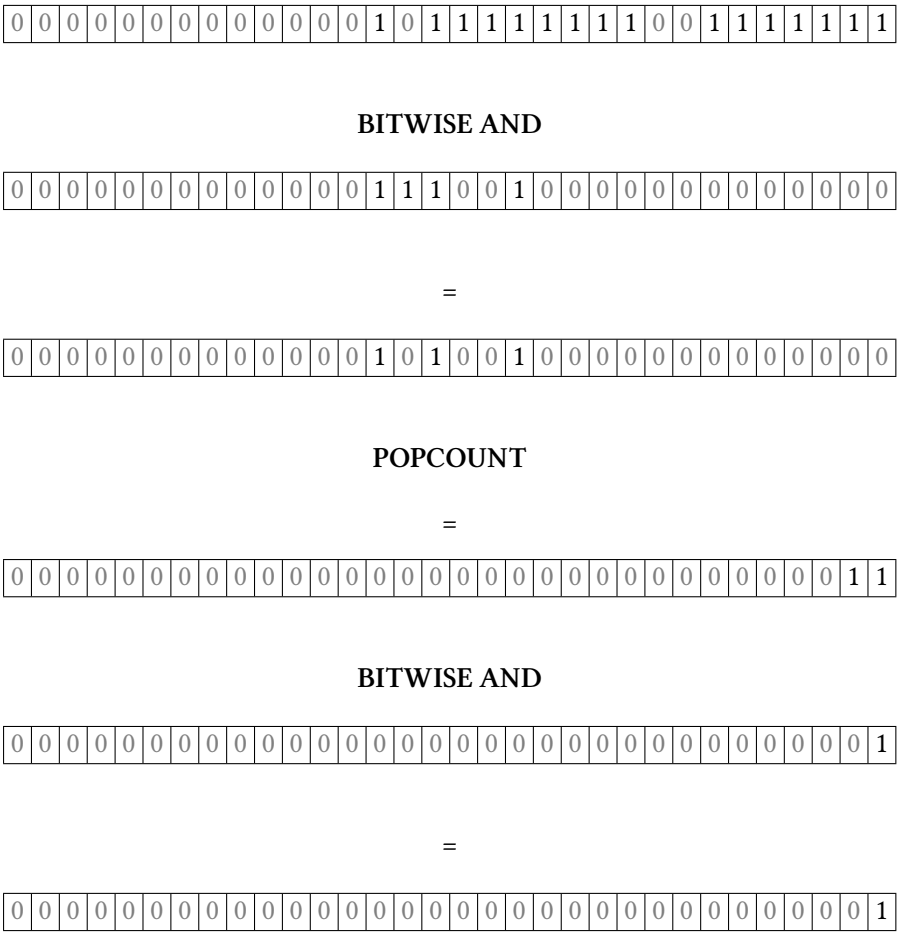

FIGURE 2.4: Finding the feedback bit when clocking R1 represented as a 32-bit integer.

The same example continues in fig. 2.5 where we show the operations involved in setting the register state after the clock. The register is simply shifted to the left and then a mask is applied to remove any 1 shifted out of the register (but still contained in the full 32 bits). Finally the feedback bit calculated in fig. 2.4 is applied using a bitwise OR.

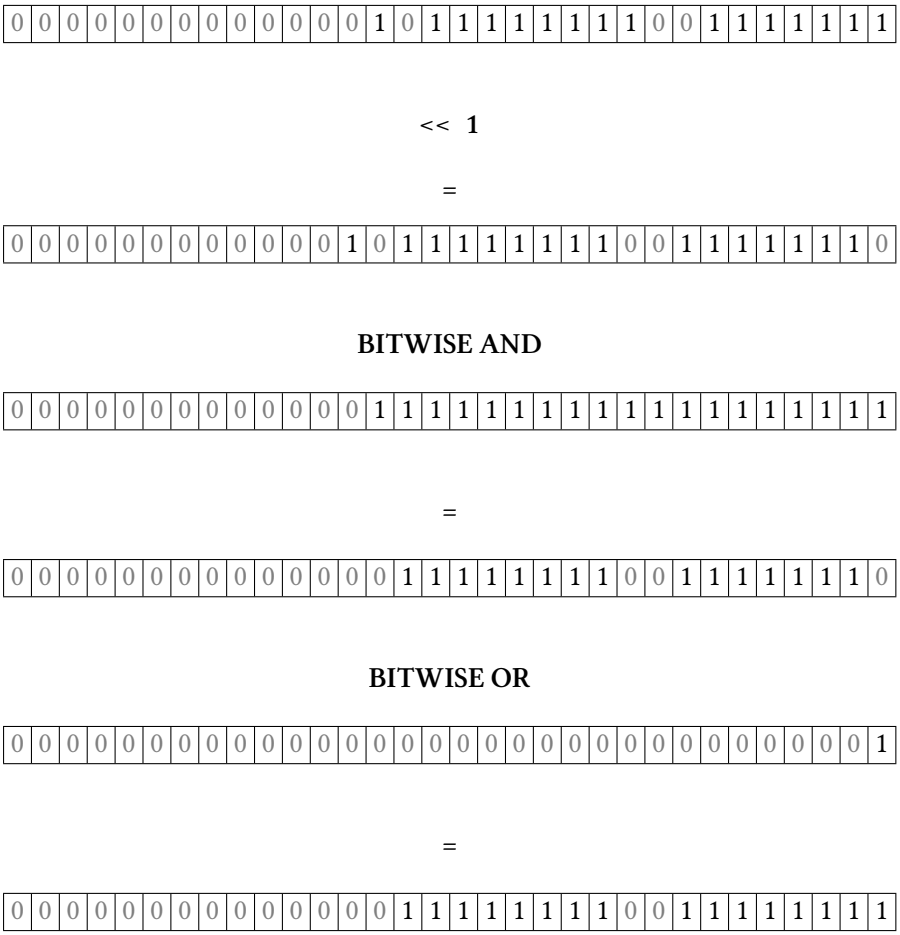

FIGURE 2.5: Setting the new register value when clocking R1 represented as a 32-bit integer.

We notice that these calculations are linear, but the same cannot be said for the majority functions 2.5 that play a part in both deciding which registers are to be clocked and more importantly in the calculation of the output. Of course this means we can not describe the full generation procedure as just a simple linear transformation, but as we will discover in the next section this little quadratic problem can be "linearized" with some extra effort.

Looking at the `A52_clock` and `A52_clock_masked` functions in fig. 2.6, we can see how the clocking is done in our implementation of A5/2. It shows how the clocking can be performed in parallel using SIMD vector instructions, for example the call to $\_mm\_and\_si128$ will perform a bitwise and on all 128 bits in one go, whereas a regular implementation would have to do so using four different operations (or two if packed into 64-bit integers).

```
static inline void
A52_clock_masked(A5Context *ctx, u32_4x *mask) {
    u32_4x parity = __zero;
    set_feedback_bits(ctx, &parity);

    u32_4x value = __zero;
    set_next_register_state(ctx, &value, &parity);
    // Apply mask to get states of only those registers that should be
    ↪ clocked
    value.V = _mm_and_si128(mask->V, value.V);

    // Reverse the mask to get a mask for registers that should not be
    ↪ clocked
    u32_4x mask_old = { .V = _mm_cmpeq_epi32(mask->V, __zero.V) };
    // Keep registers corresponding to mask_old intact, insert new register
    ↪ values in registers corresponding to mask
    ctx->registers.V = _mm_and_si128(ctx->registers.V, mask_old.V);
    ctx->registers.V = _mm_or_si128(ctx->registers.V, value.V);
}
static inline void
A52_clock(A5Context *ctx) {
  u32 R4 = ctx->registers.N[3];
  // Calculate the majority of the control bits in R4
  u32 majority = u32_majority((R4 & R4TAP1) > 0, (R4 & R4TAP2) > 0, (R4 &
  ↪ R4TAP3) > 0 );

  // Build a clocking mask for R1,R2,R3
  u32_4x mask;
  mask.V = _mm_set1_epi32(ctx->registers.N[3]);
  mask.V = _mm_and_si128(mask.V, __control_taps.V);
  mask.V = _mm_cmpgt_epi32(mask.V, __zero.V);
  mask.V = _mm_cmpeq_epi32(mask.V, majority ? __max.V : __zero.V);

  // R4 is set to always be clocked
  mask.N[3] = 0xFFFFFFFF;
  A52_clock_masked(ctx, &mask);
}
```

FIGURE 2.6: Clocking functions used in our A5/2 implementation.

3 The known plaintext attack

The goal of the known plaintext attack is to obtain the internal state of the cipher at the point in the setup right after the key has been clocked in. In this section we will present our implementation of an attack based on (Barkan et al. 2008).

Simply put, the attack is done by deducing the value of three registers $R1$, $R2$ and $R3$ from the information we have available, namely the publicly known frame number and the known plaintext. We note that since A5/2 is a stream cipher, knowing the plaintext bits and the ciphertext bits means also knowing the keystream bits and as such we will focus on just the keystream bits for simplicity.

After obtaining the internal state of the registers $R1$ through $R3$ we can use them to retrieve the secret session key used for encryption, by reversing the linear setup of the cipher, where we clock the key into the registers. This is particularly interesting in the context of some possible related-key attacks that we will discuss later, but if the goal is to decrypt other messages encrypted using A5/2, it is not necessary since the internal state found is exactly the state before any frame number is clocked in for that given session key, so for any other frame the cipher can simply be run from that point on to generate the correct keystream. Furthermore, since we can generate the keystreams, it is also possible to encrypt frames, as if they were being sent by the client to the server or in the opposite direction.

To execute the attack we need the publicly known frame numbers and the plaintext bits for the incoming and outgoing streams for at least four different frames. Our attack can be easily adapted to a situation where we only know the plaintext for either the incoming or outgoing stream. In this case we would get half as much data from each frame, and as such we would instead require the plaintext from at least eight different frames instead of the four.

We have implemented the attack as described by (Barkan et al. 2008) whose approach works by guessing the value of $R4$, and then taking advantage of how $R4$ controls the clocking of the other three registers. Recall that the size of the $R4$ register is 17 bits, meaning we need at most 2^{17} guesses to break the cipher and obtain its internal state.

With the knowledge of how the clocking will be done we can run through the cipher symbolically. Instead of operating on actual bits, like the "real" cipher would, we keep track of what each register will look like, as a linear combination of the internal state and then apply the same operations, we would on the bits, to the linear combinations.

We do this for all three registers, and whenever we would output a bit, we note what the linear combination for this output bit is. Then, since we know the plaintext (and thus the keystreams), we know what all these linear combinations should equal. This allows us to construct a linear system of equations, which we are then, given enough equations, able to solve, giving us the internal state of the register at the time that we started tracking the state.

We will do this for any possible value of $R4$, looking at the linear system to decide whether or not the guess was valid, and eventually we should be able to find only one solution which is valid, namely the $R4$ associated with the key, which was used to encrypt the frames of plaintext in question. Finally, knowing the value of $R4$, we simply apply the clocking in reverse to obtain the values of the other registers, and in the end we have the values of all 4 registers, right after the key was clocked in.

3.1 Representing the state symbolically

Let $R1_j$ be the state of $R1$ at time j, $R1_0$ be the initial state of $R1$, and $R1_j[i]$ be the i'th bit of $R1$ at time j (counting from zero, the first bit would be $R1_j[0]$). Because all operations on the register

are linear, we can represent each $R1_j[i]$ as a linear combination (using XOR) of the bits $R1_0[i]$ of the initial state and some constant c. This allows us to represent all bits symbolically and perform the necessary operations on the registers, without knowing the actual value of any of the bits. Instead we would start out with each bit being equal to itself:

$$R1_0[i] = 0 * c \oplus 0 * R1_0[0] \oplus \cdots + 1 * R1_0[i] \oplus \ldots \oplus 0 * R1_0[18] \tag{3.1}$$

Part of the reason for including a constant in the linear combination, is that it allows us to represent actual values of bits, for example if we know the value of one of the bits, we would set all coefficients to 0 and the constant to 1. Recall that during the setup phase of A5/2, after we have clocked all bits from the frame number we force one bit in each to 1, which we represent as just described (we call these bits the *loaded bits*). Additionally it enables us to represent outputs from different frames as linear combinations of the same internal state/frame number, allowing us to obtain more equations from different frames. We will get back to the details of this later.

We have defined the following structure to represent the linear combination, *constant* is the coefficient in front of the constant, $R1[0]$ would be the coefficient for the first bit from $R1$, $R1[18]$ the coefficients for the 19'th and last bit of $R1$, the same applies for $R2$ and $R3$:

```
typedef struct
{
  u08 constant;
  u08 R1[R1_LEN];   // R1_LEN = 19
  u08 R2[R2_LEN];   // R2_LEN = 22
  u08 R3[R3_LEN];   // R3_LEN = 23
  u08 R1P[R1P_LEN]; // R1P_LEN = 171
  u08 R2P[R2P_LEN]; // R2P_LEN = 231
  u08 R3P[R3P_LEN]; // R3P_LEN = 253
} Vector;
```

Of course the bits from a register will only ever depend on bits from the same register, however recall that the output bits will depend on bits from all three registers. For simplicity and to make the code easier to use, we have chosen to define a general Vector structure, which can represent bits from all the registers as well as bits from the output keystreams. This significantly simplifies things, as the same definitions will work in both cases.

The problem with representing the current state as a linear combination of the initial state, is of course that all operations used when generating an output bit are linear, except for the majority function which is quadratic.

Luckily we can linearize the majority function, by introducing a variable for each of the possible products we can end up with when applying the function. We only ever apply the majority function on bits from the same register, which greatly reduces the number of products we need to introduce new variables to account for.

For example, since there are 19 bits in $R1$ we will need variables for $19^2 = 361$ different products. However, we need to keep in mind that we are operating on bits. This means that multiplication is commutative, and we also note that $a^2 = a$ for any $a \in \{0, 1\}$. This leaves us with $\frac{19*18}{2} = 171$ products from $R1$, or $\frac{n^2-n}{2}$ in the general case. In total we end up with $171 + 231 + 253 = 655$ additional variables to represent the possible products between all three registers, for a total of 719 variables including the 64 variables we need to represent the bits from the registers $R1$ through $R3$.

We can represent the products between n variables as a $n \times n$ matrix, where by the arguments above, the products we have introduced variables for will be everything above the diagonal going from the top left to the bottom right (corresponding to every entry with a number in this example):

$$\begin{bmatrix} - & 0 & 1 & 2 & 3 \\ - & - & 4 & 5 & 6 \\ - & - & - & 7 & 8 \\ - & - & - & - & 9 \\ - & - & - & - & - \end{bmatrix} \tag{3.2}$$

Let $R1P[i]$ denote the i'th product from $R1$. This product will correspond to the entry with the number $i - 1$ (since our implementation starts counting at 0), which in turn will correspond to the product between the variable corresponding to the given row and column of the matrix.

The 5×5 matrix in 3.2 shows us how to index products between 5 different variables. The 5th product (the one numbered 4) will correspond to the product between the 3rd and 2nd variables, because it is in the 3rd column and 2nd row.

Our implementation needs to be able to map this relationship both ways, since we need to know which variables correspond to which products, but also which products correspond to which variables. We found a nice computational formula for converting from the 2-dimensional index to a 1-dimensional index (mapping two variables to their corresponding product):

```
1  static u32
2  product_1d_index(const int size, int x, int y) {
3    if (x < y) { //XOR swap
4      x ^= y;
5      y ^= x;
6      x ^= y;
7    }
8    const int diagonalSize = y + 1;
9    const int areaBelowDiagonal = (diagonalSize * diagonalSize - diagonalSize) /
     ↪  2;
10   const int unusedEntries = areaBelowDiagonal + diagonalSize;
11   const int usedEntries = diagonalSize * size - unusedEntries;
12   return usedEntries - (size - x);
13 }
```

We have however been unable to find a nice computational way to convert from a 1-dimensional index to a 2-dimensional index (mapping a product to its two corresponding variables). Of course this can be naïvely implemented using two loops, to count the number of variables, as in the example with the matrix above, but this is not very efficient. What we have done instead, is to precompute the indices by counting them, allowing us to look up the corresponding 2d index in constant time, at runtime.

To multiply two vectors, lets call them v and u, we would start by looking at the constants for each of the vectors. If the constant is 1 we need to XOR each of the coefficients in the other vector into the result, we do this for both v and u. For example, suppose the constant for v is 1 and the constant for u is 0, then we would need to XOR the coefficients from u into the result, and *not* the constants from v.

We XOR them in, because of course we are working with bits and XORing with some value twice, is the same as not XORing it at all. We have defined a function for multiplying two vectors:

```
1  static void
2  vector_multiply(Vector *dest, Vector *left, Vector *right) {
3    // Constants
4    dest->constant = left->constant * right->constant;
5    for (u32 i = 0; i < ARRAY_SIZE(left->data); ++i) {
6      const u08 lp = left->constant * right->data[i];
7      const u08 rp = right->constant * left->data[i];
8      dest->data[i] = lp ^ rp;
9    }
10
11   // Multiply R1
12   u32 size = ARRAY_SIZE(dest->R1);
13   for (u32 i = 0; i < size; ++i)
14     for (u32 j = 0; j < size; ++j) {
15       // Check that we are not multiplying by 0
16       if (!(left->R1[i] * right->R1[j])) continue;
17
18       // Special case: a^2
19       if(i == j) {
20         dest->R1[i] ^= 1;
21         continue;
22       }
23
24       dest->R1P[product_1d_index(size, i, j)] ^= 1;
25     }
26
27     // Multiply R2 and R3 parts the same way...
28 }
```

This function multiplies the vector `left` with the vector `right` and stores the result in `dest`. Since we know that we never multiply vectors depending on the state of more than one register, as we only have to account for multiplication of bits from the same register, we can keep the implementation relatively simple. To make the function easier to use, and better suited for a more general case, it will multiply the $R1$, $R2$, and $R3$ parts separately. Although in practice we would only ever see non-zero coefficients for one of the registers, meaning that for the multiplications we perform, only one of the for-loops would ever be relevant. We found it easier to have one function that works in all the cases, rather than three separate functions.

3.2 Tracking the outputs as a function of the internal state

For a given value of $R4$ we need to represent the output as a function of the internal state after the frame number has been clocked in and the loaded bits set. To help us do this we have defined the following struct:

```
typedef struct
{
  Vector *R1[R1_LEN];
  Vector *R2[R2_LEN];
  Vector *R3[R3_LEN];
  Vector **output;

  u32 R4;
  struct
  {
    u08 R1[R1_LEN];
    u08 R2[R2_LEN];
    u08 R3[R3_LEN];
    u08 R4[R4_LEN];
  } constants;
} State;
```

We maintain a vector for each bit in each of the three unknown registers, called $R1$ through $R3$, as well as the current value for $R4$. Additionally we keep a list of vectors for each output bit, adding a vector as we output more keystream "bits". We will get back to the constants and what their exact purpose is in the following section.

We start off with no outputs and the vector for each of the bits in each of the 3 registers initialized to be equal to itself (that is the linear combination with 0 for the constant and every coefficient except for the one corresponding to the bit itself). There are 3 bits, one in each register, though, that we always set to 1 (the loaded bits), so no matter what, we will always know the values of these 3 bits, which are set right after clocking in the frame number. To represent this, we set every coefficient in the vector for the corresponding bits to 0, and the constant to 1, using the `state_track_loaded_bit` function:

```
static void
state_track_loaded_bit(State *state) {
  vector_set(state->R1[15], 0, 1);
  vector_set(state->R2[16], 0, 1);
  vector_set(state->R3[18], 0, 1);
}
```

For the state we have implemented similar functions, which look a lot like their counterparts from the implementation of the actual cipher. For example the `state_clock` function keeps track of clocking the registers, and works the same way as the `A52_clock` function, by actually clocking $R4$, but instead symbolically clocks the other three registers, by applying the same operations that we would have applied to the actual registers on their respective vectors (as for A5/2 we have a function, `state_clock_all`, which clocks all registers regardless of masks):

```
static void
state_clock(State *state) {
```

```
  // Calculate masks for clocking the same way as done for A5/2
  // ..

  state_clock_r4(state);
  if (!state) return;
  if (mask.N[0]) state_track_clock_R1(state);
  if (mask.N[1]) state_track_clock_R2(state);
  if (mask.N[2]) state_track_clock_R3(state);
}
```

For each of the three different registers we have a function that keeps track of the clocking for the register. They perform the exact same binary operations on the bits of the register as the actual cipher would, only, again, operating vectors instead of actual bits. The implementation for all three registers is the same, though of course we need to operate on different taps, which is why we have distinct functions for each register.

For example, when we have to clock $R1$ we need to compute the next value by XORing the four different taps on the register (by applying the feedback function on them, which is just their XOR sum). We store this value, then shift everything to the left, by copying the value from the first bit to the second bit and so on, starting from the last bit. When we have done this we copy the value we computed into the first bit, and we have now clocked $R1$ symbolically. The functions for clocking $R2$ and $R3$ symbolically are identical, only the XOR is in the respective taps for the register. The implementation of the function that clocks $R1$ looks like this:

```
static void
state_track_clock_R1(State *state) {
  const u32 size = ARRAY_SIZE(state->R1);

  // Compute the first bit
  Vector *first = vector_alloc(0);
  vector_xor(first, state->R1[13]);
  vector_xor(first, state->R1[16]);
  vector_xor(first, state->R1[17]);
  vector_xor(first, state->R1[18]);

  // Shift the other bits left
  for (u32 i = size - 1; i > 0; --i) {
    vector_copy(state->R1[i], state->R1[i - 1]);
  }

  vector_copy(state->R1[0], first);
  free(first);
}
```

We also define some helper functions `state_init`, which makes sure that everything in the state struct is initialized and ready to use (it also helps us keep track of the XOR difference – more on that later). Likewise, we have `state_start`, which sets all the vectors to their initial states, and the state we track will be a function of the internal state at the point where we call this function.

These functions together can be combined into a function which tracks the setup phase of the cipher, by first initializing the state, clocking in the frame number, etc. the same way the actual cipher would to it. This gives us the function:

```
static void
state_setup(const u32 r4, const u32 frame, State *state, State* firstState) {
  state_init(state, r4);

  // Clock frame number
  for (u08 i = 0; i < 22; ++i) {
    state_clock_all(state, 1);

    const u08 value = frame >> i & 1u;
    state->R4 ^= value;
    state_track_frame_xor(state, value);
  }
  state_start(state, firstState);

  // Loaded bit
  state->R4 |= __loaded_mask.N[3];
  state_track_loaded_bit(state);

  for(u08 i = 0; i < 99; ++i)
    state_clock(state);
}
```

At this point, we know what the value of each bit of the register will be after the setup, as a function of the state when we called `state_start`. All that remains is to track the outputs bits as a function of the state as well. When running the cipher we can use the aforementioned functions to keep track of the clocking, but we need some other way to keep track of the keystream bits we output. Luckily we found a way to linearize this, so it is just a matter of applying the same functions we would on the bits, on the symbolic representation to a symbolic representation for the output bits. `state_track_output_majority` helps us perform the multiplication that is done to compute the majority from a register (and is left out for brevity), our function for tracking the output is:

```
static void
state_output(State *state) {
  // XOR the leaving bits from each register
  Vector *output = vector_alloc(0);
  Vector *product = vector_alloc(0);
  vector_xor(output, state->R1[18]);
  vector_xor(output, state->R2[21]);
  vector_xor(output, state->R3[22]);

  // Majority from R1
```

```
11   vector_invert(state->R1[14]);
12   state_track_output_majority(output, product, state->R1[15], state->R1[14],
     ↪  state->R1[12]);
13   vector_invert(state->R1[14]);
14
15   // Majority form R2
16   vector_invert(state->R2[16]);
17   state_track_output_majority(output, product, state->R2[16], state->R2[13],
     ↪  state->R2[9]);
18   vector_invert(state->R2[16]);
19
20   // Majority from R3
21   vector_invert(state->R3[13]);
22   state_track_output_majority(output, product, state->R3[18], state->R3[16],
     ↪  state->R3[13]);
23   vector_invert(state->R3[13]);
24
25   free(product);
26
27   // Adds output to the array of outputs
28   sb_push(state->output, output);
29 }
```

Finally we have the `state_cipher` function. Again it does the same thing the cipher would, Clocking the registers $114 * 2$ times, and outputting a keystream bit each time:

```
1 static void
2 state_cipher(State *state) {
3   for (u08 i = 0; i < 114; ++i) {
4     state_clock(state);
5     state_output(state);
6   }
7
8   for (u08 i = 0; i < 114; ++i) {
9     state_clock(state);
10    state_output(state);
11  }
12 }
```

To get the symbolic representation we would call the `state_setup` and `state_cipher` function the same way we would call the `A52_setup` and `A52_cipher` functions. Of course the problem with this approach is that we track everything after clocking in the frame number, which means that for different frames the linear combinations we get will be for different internal states. Given that one frame (regardless of whether we have the plaintext for both keystreams or just one) does not give us enough equations to solve the system, we need some way to be able to combine equations from several states to be able to solve it.

3.2.1 Combining several states from different frames

Given that we track the state of the outputs as a function of the internal state after clocking in the frame number, outputs from different frames will depend on different internal states, since the state of the registers will be different, because we clocked in a different frame number. We would like to get around this, so we can combine equations from several different frames into one system of equations.

To get around this we will track the clocking of the registers, whenever we clock in a a bit from the frame number. Additionally we will track the frame number bit that we XOR with each of the registers, by XORing it with the constant of the corresponding vectors:

```
static void
state_track_frame_xor(State *state, u08 bit) {
  state->R1[0]->constant ^= bit;
  state->R2[0]->constant ^= bit;
  state->R3[0]->constant ^= bit;
}
```

After clocking in all bits from the frame number, the constants for each of our vectors will tell us how the frame number affected the specific bit. Naturally this will be different for different frame numbers.

All the constants should be zero when we start tracking what is going on, but if we instead set them to the XOR difference of the constants from the initial frame and the constants of the current frame, we will represent the state as a function of the state of the initial frame instead of the current one. (As the XOR difference is exactly the difference between the two states after clocking in the frame number).

This means that for the initial frame the constants will all be set to zero, while for other frames they will be equal to the XOR difference between the initial frame and the current one. It is important to note that we are still tracking things as a function of the state after clocking in the frame number, so we have to remember to reset all the coefficients that will have changed during the clocking of the frame number. This is done by the call to `state_start`, which sets the coefficients so that a given bit is equal to one times itself.

```
static void
state_start(State *state, State *initialState) {
  vector_reset_state(state->R1, ARRAY_SIZE(state->R1), 0);
  vector_reset_state(state->R2, ARRAY_SIZE(state->R2), ARRAY_SIZE(state->R1));
  vector_reset_state(state->R3, ARRAY_SIZE(state->R3), ARRAY_SIZE(state->R1) +
  ↪  ARRAY_SIZE(state->R2));

  // Set the constants
  for (int i = 0; i < R1_LEN; ++i) {
    state->constants.R1[i] = state->R1[i]->constant;
  }
  for (int i = 0; i < R2_LEN; ++i) {
    state->constants.R2[i] = state->R2[i]->constant;
  }
```

```
14    for (int i = 0; i < R3_LEN; ++i) {
15      state->constants.R3[i] = state->R3[i]->constant;
16    }
17
18    // XOR Constants from frame number
19    if (initialState) {
20      for (int i = 0; i < R1_LEN; ++i) {
21        state->R1[i]->constant ^= initialState->constants.R1[i];
22      }
23
24      for (int i = 0; i < R2_LEN; ++i) {
25        state->R2[i]->constant ^= initialState->constants.R2[i];
26      }
27
28      for (int i = 0; i < R3_LEN; ++i) {
29        state->R3[i]->constant ^= initialState->constants.R3[i];
30      }
31    }
32  }
```

It is worth noting that at this point we haven't utilized the specific plaintext/keystream bits, which means that this first part of the attack can be precomputed before these bits are known to speed up the attack, this leaves us with the next part, namely solving the system of linear equations, as the only significant computational work an attacker will have to do when the data is sent in realtime.

Though it important to note that with our current implementation it would require a substantial amount of memory to store all precomputed matrices, we approximate around 80 gigabytes. However, there are several ways to get around this, firstly one could easily optimize the matrix we store, by not accounting for the loaded bits or their products, this would reduce the size quite a bit. Another possible optimization is to pack the bits bytes such that instead of having one byte represent each binary value, we would represent eight of them in one byte, reducing the amount of memory required by a factor eight. We would estimate that by applying these optimizations the amount of memory required to hold every possible matrix would be around 9 gigabytes (not account for padding or alignment). Note that the during the precomputation we would also perform the Gauss-Jordan elimination, which is the most time consuming part of the attack.

Furthermore, we see that the guesses themselves are not in any way dependent of each other, this means that we can partition the possible guesses into ranges, and run the attack for these ranges in parallel with each other. Because of this, we can very easily scale the attach horizontally, the more computing power we have, the faster we can run the attack and guess for all 2^{17} possible values of $R4$.

3.3 Solving the system to obtain the internal state

Following the steps described in the previous section, we can run our attack and generate a set of linear combinations, which express the value of each of the output bits as a function of the internal state. Since we know the plaintext, we also know the keystream which we can obtain by XORing

the plaintext with the ciphertext. This effectively gives us a set of linear equations, because with the actual keystream bits, we know what the value of each of the linear combinations should be.

Since we express each output bit as a linear combination of the 64 bits of the three registers, their 655 products and one constant for a total of 719 unknowns, we should in theory be able to solve the system and obtain solutions for all variables, given exactly 719 equations.

However, because of the way the products are linearized there will be some dependencies and constraints that the linear system cannot express, so in practice even given 719 equations we cannot obtains solutions for every unknown in the system. Although it is our experience that we, from this many equations, can obtain enough solutions, we are only really interested in solutions to the first 64 variables, while additional solutions make it a bit easier for us down the line (exactly why is explained in the next section, on knowing when a guess is correct).

Due to how the products are linearized, and perhaps to the way the cipher works, there are some not so obvious dependencies between the variables, of which we encountered some when writing the initial implementation of the attack.

For example, we decided to first try and implement a version of the attack, where we start tracking the state of the system right after we clock in the key, but before we clock in the frame number. By doing so we would not have to account for the XOR difference between the frame numbers (as described in the previous section), since everything would be a function of something depending solely on the key, though easier to implement. Of course the cost of tracking everything from this point, and not instead after the loaded bits were set meant that we did not get the solutions for three of the bits and we would have more unknowns in our system

After implementing the attack this way, we found that no matter how many equations we had in our system, we were never able to obtain solutions for all 64 variables corresponding to bits from the register (even when trying with equations from 1024 frames, totalling $1024 * 114 * 2 = 233472$ equations). We would always end up not being able to obtain solutions for the same set of bits and their corresponding variables, regardless of which key and frame number we used, and the number of equations.

This led us to explore ways to simplify the system and ways to deterministically obtain solutions for some of the variables the Gauss-Jordan elimination algorithm could not obtain solutions for.

We started by looking at the products. Of course given a solution to the two variables in a product, means that we can compute the product. If either of the variables are 0, we know that product must also be 0. After simplifying the products, we went through all of them, because if we know that the product is 0 and one of the variables are 1, we know that the other variable must be 0 (after which we can then simplify products involving the now 0 variable).

We found that even doing this, did not solve the remaining unknown bits, so we looked at if there was a way to determine the value of an unknown variable from the products we know. For example, for some unknown x we can set it to zero and look at the products involving x which we have solutions for. For $x = 0$ these products should still hold, if they don't we know that it must then be the case that $x = 1$. Likewise if they hold for $x = 0$, but not $x = 1$ we know it must be the case that $x = 0$.

Though again, we found that by doing this there were none of the unknowns for which we could determine the value, by ruling out one that would violate the products. At this point we decided to instead implement the tracking so that it would account for the XOR difference, which would reduce the number of variables. We could perhaps have explored, whether or not there were combinations of values that would hold, whereas others would not, but we found that it would probably be easier to implement it the other way, rather than having to start checking possible combinations until we found something valid.

As such we ended up implementing the attack so that it tracks the outputs as a function of the internal state after the frame number has been clocked in and the loaded bits set, the same way as described by (Barkan et al. 2008). We did however decide to keep the implementation simple, by having 719 "unknowns", even though we had the value for the three loaded bits, which in turn means that we need to linearize less products (because the loaded bit is 1, then the product will be equal to the other operand).

After doing this, we were always able to obtain solutions for the first 64 variables, the constant and a larger portion of the products. Our implementation starts by constructing the linear system of equations, which will be a matrix of coefficients from the linear combinations, with an additional column for the right hand side of the equations.

Once we have the matrix we perform a standard Gauss-Jordan elimination on it, from which we check which variables we found solutions for. At this point, since we solve for a guess on $R4$, we need to somehow know whether or not this guess was valid. We will distinguish between a solution being valid and being correct. In this case *valid* means that all constraints hold, and we can obtain solutions for the bits, whereas *correct* means that the solution was valid and the correct value of $R4$.

3.3.1 Knowing when a guess is wrong

While it might seem inefficient to include variables for bits that we know the value of in the system, when we could account for this and reduce the number of equations required, it also has one very beneficial advantage. Of course since the linear combinations of the loaded bits are constants, no equation in our system will actually depend on the loaded bits, but rather the constant.

This means that the columns corresponding to the loaded bits will always be all zero, and for any valid solution to the system we know that the variables corresponding the loaded bits should always solve to a null row (also called a free variable), after we have performed our Gauss-Jordan elimination on the system.

Furthermore, we include a column for the constant in our system, as some of the equations might depend on it. Naturally we know that the value of the constant must be 1, even though this constraint is not reflected in the system, so we can discard any solution in which the constant solves to something other than 1 as being an invalid solution.

Since our goal is to obtain solutions for the first 64 variables (the ones corresponding to the bits of the register $R1$ through $R3$) and because we know that there exists a solution to the system, we can discard any solutions where we do not obtain solutions for the first 64 variables (whilst accounting for the fact that three of them should solve to free variables).

Assuming we get this far, we will have a solution where we know the value of all three registers. Inevitably we will also obtain solutions for some of the products. Since our linear system of equations can't reflect the constraints that the variables corresponding to the products should indeed be equal to the product of the two variables, we can check that this is the case, and discard any system where the products do not hold as invalid.

From this we have a four step process, determining whether or not a given solution is valid or not:

1. Check that the constant solves to 1
2. Check that the variables corresponding to the loaded bits solve to free variables
3. Check that we obtain solutions for the first $64 - 3$ variables (excluding the loaded bits)

4. Check that the solutions we obtain for the products are equal to the product of the variables

We find that in practice these four checks are enough to validate a solution and from our testing we have yet to find a guess for the wrong $R4$ that yields a solution that would be valid according to the above checks.

We cannot exclude that there might be several solutions, which are valid according to the above steps. When this is the case we will iterate over each of the valid solutions and run the cipher from the given state of the register, from which we can generate a keystream for the frame number we have. Since we know the plaintext, we can XOR it with the ciphertext to obtain the keystream, to check whether or not the guess is correct. We then compare the keystreams and if they are equal the guess is correct. It should never be the case that two different internal states yield the same keystream.

3.3.2 Reverse clocking the registers

Recall that when clocking some register R with a set of n different taps, $tap_1, \cdots, tap_n$, we compute the XOR value of the taps $value = tap_1 \oplus \cdots \oplus tap_n$, which we XOR with a bit in the frame number, called a $framebit$, when clocking in the frame number. We can compute the new value of the first bit of the register after shifting as:

$$bit = (tap_1 \oplus \cdots \oplus tap_n) \oplus framebit \tag{3.3}$$

The taps for each of the registers are chosen such that one of them will always be the last bit of the register. This means that when shifting the bits in the register, we shift out the value of the last tap, meaning that when we look at some state of a register R_n, we can get the values for $tap_1, \cdots, tap_{n-1}$ from the state R_{n-1}. Likewise after shifting we see that $R_n[1] = R_{n-1}[0]$.

With a publicly known frame number, the value of all but one of the taps and the value of the first bit we get an equation in one variable, for the value of the last of the taps:

$$tap_n = (tap_1 \oplus \ldots \oplus tap_{n-1}) \oplus framebit \oplus bit \tag{3.4}$$

Then, assuming there are m bits in the register R, to reverse clock from R_n to R_{n-1} we compute the value of the last tap, tap_n with the formula above. We then shift everything to the right and set $R_n[m-1] = tap_n$.

We will illustrate this with an example; assuming we are reverse clocking $R1$, let $framebit = 1$, $R1_1$ be the register after the clocking operation, and $R1_0$ the register before the clocking operation. Now assume $R1$ is equal to the following bit string (with taps from $R1_0$ highlighted in red, the first bit from $R1_0$ in blue):

$$R1_1 = 10111010001001110 \tag{3.5}$$

From this we get the values of the $n-1$ taps and the value of the first bit, which we then insert into the aforementioned formula to compute the value of tap_n:Ø

$$\begin{aligned} &tap_1 = 1,\ tap_2 = 0,\ tap_3 = 1,\ bit = 0, \\ &tap_4 = tap_1 \oplus tap_2 \oplus tap_3 \oplus bit \oplus framebit \\ &\quad = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \\ &\quad = 1 \end{aligned} \tag{3.6}$$

We now shift the register to the right: $R1'_1 = R1_1 >> 1$, and set the last bit $R1'_1[16] = tap_n = 1$, after which we see that $R1'_1 = R1_0$:

$$R1'_1 = 1101110100010011 1 = R1_0 \tag{3.7}$$

After clocking in the frame number we set a loaded bit in each of the three registers $R1$ through $R3$, and since we start tracking the state after setting this loaded bit, the solution we obtain in the attack will be the values of the registers $R1$ through $R3$ after this loaded bit was set. Luckily, the loaded bits do not overlap with any of the taps in any of the three registers (neither does it for $R4$). Since the bit is forced to 1, we have no way of knowing the previous value of the bit, but we are lucky that the frame numbers are 22-bits, and the longest register $R3$ is 23 bits, with the loaded bit at $R3[18]$.

As such, since we have to reverse clock (and thus shift right) 22 times to reverse clock the entire frame number, we end up shifting the loaded bits out of each of their corresponding registers, meaning that they are of no significance to the reverse clocking process.

Once we have reverse clocked the registers, we know their value right after the key was clocked in. We can use these in conjunction with the cipher to generate keystreams for following or previous frames and decrypt/encrypt them at will. Additionally it is also possible to reverse clock the register (though in a different way from the one described) to obtain the session key itself, as the process of clocking in the keybits is linear.

3.3.3 A demonstration of the implementation

We have provided a demonstration of the known plaintext attack in `plaintext.c`. The code will generate the keystreams for a known key and frame number (incrementing the frame number each frame). After this it will it will try and guess every value in the range $[R4 - c, R4 + c]$, for some configurable constant c.

If a guess is invalid it will print why the guess is invalid, if it is valid it will add it to a list. When we are done guessing we iterate over all the valid guesses (usually there is only one), for the internal state corresponding to that guess. We reverse clock the register to obtain their value before the key was clocked in. We can now use the values of the registers to generate the keystream. If the generated keystream matches the known keystream we know that the guess is correct.

4 The known ciphertext attack

In this section we expand on the attack from the previous section allowing us to break A5/2 with only knowing the ciphertext.

4.1 Error correction codes and their use in A5/2

When sending data across a network errors can occur. Specifically it might be the case that a few bits in a message are flipped because of noise on the channel. In order to correct such errors GSM uses error correction codes (ECCs).

In general an ECC is generated using a known generator matrix, $\mathbf{G}$, and checked with a matching parity-check matrix, $\mathbf{H}$. For any vector, v, the vector, w generated by $\mathbf{G}v = w$ is said to be a valid codeword which can be verified by checking that $\mathbf{H}w = \mathbf{0}$. The idea is that w can then be sent over an error-prone channel: let e be the error vector such that the receiver of w actually sees $w \oplus e$, the idea is then that

$$\mathbf{H}(w \oplus e^T) = \mathbf{H}w \oplus \mathbf{H}e^T = \mathbf{0} \oplus \mathbf{H}e^T = \mathbf{H}e^T \tag{4.1}$$

If the population count of e is small enough, that is there are few errors, it is then possible to definitely determine w when H is known. A similar calculation will play a role in the ciphertext attack, but the main reason why we can attack the ciphertext has nothing to do with the design of the ECC itself and everything to do with how or rather when it is used.

4.1.1 Encoding before Encryption

Like (Barkan et al. 2008) we consider the ECCs of the Slow Associated Control Channel which is commonly used early in a call. The message, m, that is to be encoded is a fixed 184 bits and after multiplying with the 456x184 generator matrix, $\mathbf{G}$, this is expanded to 456 bits which means each message is split into 4 frames because the encryption is done after encoding (recall that each frame gives us 114 bits of keystream in each direction). Note that there is an interleaving operation in this procedure as well which (Barkan et al. 2008) describe as a constant vector, g, but this is inconsequential for both the analysis and implementation, so we will disregard it even though it would of course be a necessary detail in a real-world attack.

Let k denote the 456 keystream bits we get from the 4 frames of A5/2, and let $\mathbf{H}$ be a 272x456 parity-check matrix, then the ciphertext, c, is given by $\mathbf{G}m \oplus k^T$ if error correction is done before encryption. This means that

$$\mathbf{H}c = \mathbf{H}((\mathbf{G}m) \oplus k^T) = \mathbf{H}(\mathbf{G}m) \oplus \mathbf{H}k^T = \mathbf{0} \oplus \mathbf{H}k^T = \mathbf{H}k^T \tag{4.2}$$

Since c and $\mathbf{H}$ are known we can get linear equations over the bits of k and of course this would not be possible if only the designers of A5/2 had simply changed the order of encoding and encryption which would clearly have been equally effective for error correction.

4.1.2 Implementing ECCs in A5/2

For our purposes we do not necessarily need to implement a specific ECC, as long as we create a generator matrix, $\mathbf{G}$, and a parity-check matrix, $\mathbf{H}$, such that $\mathbf{HG} = \mathbf{0}$, the equations in the previous section will hold. However, we quickly realized when experimenting that a highly rank

deficient **H** will make a large fraction of potential codewords valid and therefore this would be undesirable not only for our attack, but in fact also for error correction.

The standard forms we have used for **G** and **H** are

$$\mathbf{G} = \left[\begin{array}{c} \mathbf{I}_{k\times k} \\ \hline \mathbf{R}_{(n-k)\times k} \end{array}\right], \quad \mathbf{H} = \left[\begin{array}{c|c} \mathbf{R}_{(n-k)\times k} & \mathbf{I}_{(n-k)\times(n-k)} \end{array}\right],$$

where $\mathbf{I}_{a\times b}$ is the identity matrix of size $a \times b$.

We note that for **G** with dimensions $n \times k$, **H** has dimensions $(n-k) \times n$ and clearly we have that

$$\mathbf{HG} = \mathbf{R} \oplus \mathbf{R} = \mathbf{0}$$

In order to avoid rank deficiency **R** is chosen randomly in our implementation.

4.2 Implementing the attack

The implementation of the known ciphertext attack builds upon the implementation of the known plaintext attack. For a guess on the value of $R4$, we effectively run the known plaintext attack, from which we get a set of equations, expressing the keystream bits $K_1, \ldots, k_{114}$ as a function of the internal state:

$$\begin{aligned} k_1 &= \alpha_{1,0}c \oplus \alpha_{1,1}R1_0[0] \oplus \ldots \alpha_{1,719}R3_0[22] \\ &\vdots \\ k_{114} &= \alpha_{114,0}c \oplus \alpha_{114,1}R1_0[0] \oplus \ldots \alpha_{114,719}R3_0[22] \end{aligned} \tag{4.3}$$

In the known ciphertext attack, we do not know the value of the keystream bits, so these equations alone do not allows us to obtain the internal state. However, we can take advantage of how the 184 bit plaintext is encoded into a 456 bit message and then encrypted, to transform these equations into equations over something we know.

When we encode the message with the parity matrix **G**, we append an additional 272 bits to the end of the message. We see in eq. (4.2) how the ciphertext relates to the keystream. Effectively, $\mathbf{H}c^T$ expresses the 272 bits at the end of the ciphertext, as a function of the bits in the key.

$$\begin{aligned} c_{184} &= \alpha_{1,1}k_1 \oplus \ldots \oplus \alpha_{1,114}k_{114} \\ &\vdots \\ c_{456} &= \alpha_{272,1}k_1 \oplus \ldots \oplus \alpha_{272,114}k_{114} \end{aligned} \tag{4.4}$$

For each of these equations for c_i, we would XOR together the linear combinations for all k_j where the coefficient is $\alpha_{i,j} = 0$. Doing this we end up with equations expressing the ciphertext bits as a function of the internal state. Because of this, it is clear that we will have the same 719 unknowns as we did in the known plaintext attack, meaning we need the same amount of equations to be able to solve the system.

Each encoded message consists of four different frames, and provides us with 272 equations. As such we end up needing exactly three messages, consisting of 12 frames in total, to obtain enough equations to be able to solve the system. Though in practice we need more than just the three messages, as it makes knowing when a guess is wrong easier. More on this in the following section.

For a given value of $R4$ we have implemented a function, `ciphertext_guess_r4`, which performs the first part of the attack for that given $R4$, tries to solve for the internal state and

returns the result of the guess. To do this, we are given a set of messages. We run through each of these messages and multiply them with the parity check matrix, **H**. Then, for each of the four frames in this message we run parts of the plaintext attack to obtains equations for the keystream bits.

After this, we transform the equations from each frame as described previously, and build their corresponding matrix. We feed the matrix to our `solve` function, which performs Gauss-Jordan elimination to obtain solutions, and finally check whether or not the solution is valid, according to the steps outlined in section 3.3.1.

The most important part of the known ciphertext attack is implemented in the `ciphertext_guess_r4` function, which guesses on a value of $R4$ as outlined above. Note that we pass the pointer to a `CiphertextAttack` struct, which contains information about the attack. In an initial step we have populated this struct with the known messages and their frame numbers (this is done just once). This struct is also where we temporarily store the equations while guessing.

```
1  static int
2  ciphertext_guess_r4(CiphertextAttack *attack, const u32 r4, Registers
   ↪  *solution) {
3    Vector *output = attack->outputs;
4    u08 *outputValue = attack->values;
5
6    int count = 0;
7    State *initialState = &(attack->messages[0].states[0]);
8    for(int i = 0; i < CIPHERTEXT_MESSAGES_REQUIRED; ++i) {
9      KnownMessage *message = attack->messages + i;
10     u08 *parity = matrix_multiply(attack->H, H_WIDTH, H_HEIGHT,
       ↪  message->message, 1, H_WIDTH);
11
12     u08 *parityBit = parity;
13     u08 *parityRow = attack->H;
14
15     for(int j = 0; j < FRAMES_PER_MESSAGE; ++j) {
16       State *state = message->states + j;
17       state_setup(r4, message->frameNumbers[j], state, initialState);
18       state_cipher(state);
19     }
20
21     // From the second frame in the message we get 114 - (184 - 114) = 44
       ↪  equations
22     // From the 3rd and 4th everything gives us an equation
23     // Build the vector for the output bit from the parity matrix
24     for(int f = 1; f < 4; ++f) {
25       for(int k = f == 1 ? 70 : 0; k < FRAMESIZE; ++k) {
26         *outputValue++ = *parityBit++;
27
28         vector_set(output, 0, 0);
29         for (int h = 0; h < H_WIDTH; ++h) {
30           if (parityRow[h]) {
```

```
31              Vector *vector = ciphertext_get_vector_for_bit(message, h);
32              vector_xor(output, vector);
33            }
34          }
35
36          parityRow += H_WIDTH;
37          ++output;
38          ++count;
39        }
40      }
41
42      free(parity);
43    }
44
45    // Attempt to solve the matrix
46    ciphertext_populate_matrix(attack);
47    for(int m = 0; m < CIPHERTEXT_MESSAGES_REQUIRED; ++m) {
48      KnownMessage *message = attack->messages + m;
49
50      for(int f = 0; f < FRAMES_PER_MESSAGE; ++f) {
51        State *state = message->states + f;
52        state_free(state);
53      }
54    }
55
56    return solve(attack->matrix, CIPHERTEXT_EQUATIONS, solution);
57  }
```

While the guessing method relies on some helper methods, most of the heavy lifting is done by the implementation of the `state_*` methods, which we have from the known plaintext attack.

4.2.1 Knowing when a guess is wrong

We have found, that when we solve the equations, we can always obtain solutions for the first 64 variables, those corresponding the bits in the registers, and the constant. However, we never obtain solutions for any of the products. They always end up being either unknown or a free variable. This means that when performing a guess, it is much more likely to be a valid guess, since we have no products to verify the solution with, which results in somewhere between 40 – 50 of the guesses resulting in a valid solution.

We call a solution or guess *valid* when it passes all the checks described in section 3.3.1, we call the solution *correct* if it passes our stronger tests. A solution can be valid, but not correct, but not the other way around.

If a guess is valid, we can use the internal state we obtained from the solution to generate keystreams for the three messages we used to obtain solutions for. We use the keystreams to decrypt the messages. Then, by multiplying them with the parity check matrix, we should end up with a zero vector, provided the message is a valid codeword. If it is not, we know that the guess was wrong (assuming that there were no errors in the messages, when we got them initially).

For a good choice of encoding matrix **G**, there will be few valid codewords, so the odds that a message decrypts to a valid code word for a wrong guess of $R4$ is rather low.

We might still encounter a wrong guess of $R4$, where the messages we have used to obtain equations from still decrypts to valid codewords. One way to get around this is to check against more than just those three messages. Naturally the more messages we check against, the lower the probability that they will decrypt to a valid codeword is, if the guess for $R4$ is wrong. So while we can obtain solutions, with just the three messages required to get enough equations it is beneficial to have more messages, that we can perform the parity check against to validate whether or not a solution is correct.

When checking a guess for the right value of $R4$ (being the one that was actually used to encrypt the messages), we will obtain the original messages, which were encoded using **G**. This means that for any message, we will always obtain an all 0 result when checking the parity. Because of this, the right guess will always end up being correct, so we can guarantee that the right value of $R4$ will always be one of the correct guesses (provided we end up with more than one).

Of course we might end up with more than one correct guess. When this is the case we will have to apply some sort of heuristic to the decrypted message, to figure out which of the correct guesses is the right one. Our attack does not do this, it simply returns a (usually short) list of correct guesses.

We have tried to run the attack, against a known key and value of $R4$, with three messages to obtain equations from and 5 additional messages to parity check against. We guess on all possible 2^{17} values of $R4$, for a total of 131072 guesses. Guessing ran in parallel on seven different cores, and took less than 25 minutes to complete. From this we saw that about 50% of the guesses where valid, while only 96 of these valid guesses ended up being correct, naturally the actual right value of $R4$ was amongst the correct values.

Just like with the know plaintext attack, we can paralellize the known ciphertext attack, exactly the same way, by partitioning the set of possible guesses and running the attacks in parallel on these ranges.

4.2.2 Putting it all together

The process of checking the parity against the messages from which we get equations and any additional messages, is left to the caller of the function that checks a guess. We have also implemented a demonstration of the attack in `ciphertext.c`. The demonstration starts by encoding and encrypting three different plaintexts using a known key. Additionally it will encode and encrypt an extra set of messages that we use for checking parity against.

After this is done, it sets up the attack by adding to the `CiphertextAttack` struct. After this it will guess some values of $R4$, using the guessing function. If a guess is valid we will add it to the list. After we are done guessing, we iterate over all the valid guesses we found and if the guess is valid we will reverse clock the frame number and use our A5/2 implementation to generate keystreams for the messages. We then decrypt each of the messages and parity check them. If the parity vector is not all zero, we know the guess is not correct.

This implementation is a demonstration of how the attack can be run, in a somewhat realistic scenario. We could easily replace the code that encodes and encrypts our plaintext, with code that instead reads or otherwise gets the data somewhere else, in order to be able to run the attack against an actual set of data, for which we do not have the plaintext or the key.

5 Later security developments in mobile cryptography

So far our project has been about GSM and the A5/2 cipher. Now we will look at what has been done in more recent years, in order to fix some of these security issues which we have highlighted earlier. We will stick to only looking at the encryption used in later standards, and not cover the new 5G networks and their improvements. We will instead focus on changes implemented by the 3G and 4G LTE network standards.

In chapter 3 and chapter 4 we have outlined an attack on the A5/2 cipher, which also works against the A5/1 cipher, given that they both use the same key and our attack allows us to recover the session key. We estimate that these attacks are possible, primarily because of the linearity of the cipher and the ease at which the only non-linear part of the cipher can be linearized. Additionally the fact that the encoding for error checking is done before encryption, poses a fundamental problem which allows our attack to be successful, by exploiting this weakness.

Later encryption standards are open, which means that anyone can look at the inner workings of the system, which we deem to be a good thing for two different reason. Firstly an open standard can be tried and tested by anyone, meaning that more people will be able to look at the cipher and find problems, than if it had not been an open standard. Secondly, the concept of a closed standard is leaning dangerously close towards security by obscurity, which we regard to be a bad idea, given that someone who is determined enough, most likely will be able to reverse engineer it to figure out its inner working, as we saw was done with both A5/1 and A5/2 by (Briceno et al. 1999) amongst others.

While the later open standards we will discuss still have their own issues, they do mitigate the linearity problem that A5/1 and A5/2 have. The later 3G standard do this by using the KASUMI (A5/3) block cipher. Using a block cipher also mitigates the problem we see in A5/2, where we encode the message before encrypting, since we XOR with a key to encrypt, there will be a one-to-one correspondence between an encoded bit, and the encrypted bit. This is what we exploit in our known ciphertext attack. Naturally encoding before encrypting with a block cipher would serve no purpose, as an error in a single bit would propagate and avalanche to many more errors after decryption.

5.1 Third generation networks and the KASUMI (A5/3) cipher

The third generation of mobile communication networks changed the specification to use the so called KASUMI block cipher, under the name A5/3 – it is a modified version of the MISTY1 block cipher. KASUMI replaced MISTY1. Even though it should have been a strengthened version of MISTY1 running in counter mode, it is not as strong as MISTY1 (Dunkelman et al. 2010) (Biryukov 2004).

5.1.1 The KASUMI cipher

The KASUMI cipher is a 64 bit block cipher with 64-128 bit keys. The GSM standard requires that 64-bit keys be used (Barkan et al. 2008). Just like MISTY1 it has a recursive Feistel structure with 8 rounds, where each round itself consists of 2 functions: one of the functions consists of a 3-round 32-bit Feistel construction, called FO, while the other function, called FL, linearly mixes a 32-bit subkey with the data.

The order of the functions depends on the rounds; for even round numbers, the FO is applied first and for uneven round numbers, the FL function is instead applied firstly (Dunkelman et al. 2010).

The issue of linearity present in the A5/2 cipher has been defeated by the KASUMI cipher due to the non-linear S-boxes of the *FI* function. For a more in depth explanation of how the KASUMI cipher works, see the appendix A.

5.1.2 The attack on KASUMI

The attack on the KASUMI block cipher was done with an, at the time, new type of attack called a *sandwich attack*. The attack manages to find a distinguisher for 7 out of the 8 rounds of the KASUMI cipher, and then utilizes these to analyze the last round, completing the attack and obtaining the entire 128-bit key used (Dunkelman et al. 2010).

5.2 The sandwich attack

The *related-key sandwich attack* builds upon some of the same assumptions as those made in the *related key boomerang attack* (Biham et al. 2005) (Hong et al. 2005) (Kim et al. 2004), in order to exploit the dependence between the underlying differentials of the cipher, such that a more accurate estimation of the probability of the distinguisher can be obtained (Dunkelman et al. 2010).

The assumptions made in the sandwich attack are that the cipher is considered a cascade of three sub ciphers $E = E_1 \circ M \circ E_0$. It is also assumed, that there exists a related-key differential $\alpha \longrightarrow \beta$ for E_0 under key difference ΔK_{ab} with probability p. Likewise it is assumed that there exists a related-key differential $\gamma \longrightarrow \delta$ for E_1 under key difference ΔK_{ac} with probability q (Dunkelman et al. 2010)

A chosen plaintext version of the attack also exists and is called a *related-key rectangle-like sandwich attack*. The idea with this attack is that one encrypts a lot of different plaintexts with difference α and look for pairs of pairs that conform to the requirements of the boomerang attack (Dunkelman et al. 2010).

(Dunkelman et al. 2010) notes that the sandwich attack is a framework that could be considered as a formal treatment and generalization of ideas proposed by (Biryukov et al. 2003), (Biryukov et al. 2009) and (Wagner 1999).

Complexity of the sandwich attack The attack on KASUMI achieves to construct a distinguisher for 7 out of 8 rounds of KASUMI with a probability of 2^{-14}. Using such distinguishers and analyzing the last round, the complete 128 bit key can be achieved using only 4 related keys, 2^{26} data, 2^{30} bytes of memory and 2^{32} time. This low complexity has the consequence that the attack has been simulated on a single PC in less than 2 hours (Dunkelman et al. 2010).

5.2.1 So 3G is not secure

The sandwich attack proves, first of all that KASUMI is not as safe as MISTY1, since it takes less time to run than an attack on MISTY1. It is also not possible to break MISTY1 in the same way (Dunkelman et al. 2010). Though more importantly that the attack(s) on A5/3 guarantee that mobile communication using it are not secure.

Since 3G mobile communications use A5/3, as stated previously, we can not deem these to keep a guaranteed security.

5.3 The LTE standard

It should come as no surprise that the LTE standard also encrypts telecommunication. This is done using the SNOW cipher (ETSI 2006). Unfortunately the cipher is not completely secure, since it is susceptible to a related-key attack due to a so-called sliding property of the cipher and thus present a non-random property of the cipher yielding related-key distinguishers (Kircanski et al. 2011).

6 Potential further investigations on security flaws in telecommunication

There are certainly other types of security issues at stake when dealing with mobile communication networks. Some of these will be very shortly covered here.

6.1 A5/3

An implementation of the KASUMI cipher and its subsequent attack could be done as a future investigation, though this is beyond the scope of this project. For now we just give a brief introduction to the cipher and how its attack works, as can be read in section 5.1.1 and section 5.1.2.

6.2 More types of attacks on GSM networks

Besides the previously mentioned sandwich attack, there are also at least 3 other different types of attacks on the KASUMI cipher, that could be leveraged – all of these 3 attacks require that the mobile communication device supports A5/2, though if the devices it currently using A5/1 we can force it to use A5/2 instead, by performing something like a man-in-the attack. Since the same key is used for all three ciphers A5/1, A5/2 and A5/3 our attack on A5/2 allows us to attack the other two ciphers.

Additionally they require the A5/3 cipher to be run with 64 bit keys, as is specified by the GSM standard. We have not covered these attacks, as they do not have anything to do with the encryption standard itself, but rather are specific to GSM networks (Barkan et al. 2008, pp. 611-612 of the short version of the paper).

6.2.1 The 3 attacks on KASUMI mentioned in (Barkan et al. 2008)

The 3 attacks are a man in the middle attack on the victim customer, an attack where the bits are flipped (can be done as a man in the middle) so as to make sure that the A5/2 cipher is used and an attack where the adversary simply impersonates the mobile network for a short amount of time in order to make the customer switch to using A5/2.

These are all possible because the network is not authenticated to the phone in GSM networks and can be done whenever a GSM network is used alongside a phone that supports the A5/2 cipher (Barkan et al. 2008).

6.2.2 Attack on GPRS

(Barkan et al. 2008, p. 612 of the short version of the paper) also describe an attack on GPRS-networks, that are similar to the third additional attack on A5/3.

6.3 Data on the LTE network

The LTE networks also provide encryption of user data. This could be explored further, but we will suffice to say that LTE user data is encrypted in counter mode (AES-CTR) (Rupprecht et al. 2019). It is fair to say that this is a very secure encryption algorithm and thus we are unable to find any practically viable attacks on this algorithm, by using the ciphertext itself as an attack vector.

7 Conclusion

In this project we have presented an implementation of the A5/2 stream cipher, used for mobile communications on second generation GSM networks.

Based on our analysis of this cipher we conclude that the degree of linearity of its operations constitutes a fundamental security flaw. We exploit this flaw by constructing a linear system of equations over the internal state of the cipher and the known keystream bits, which we then solve to obtain the internal state of the cipher. From our implementation of the attack and by the arguments presented along it, we can conclude that it is practically feasible to break the cipher by solving at most 2^{17} of these linear systems even though the cipher uses a 64-bit key.

We also analyze the use of error correction codes in GSM, in which we find another fundamental security flaw, namely the fact that encoding is performed before encryption, which means there is a linear dependency between the encoding bits and the keystream. With a simple modification this allows us to reuse the the known plaintext attack.

While both of our attack implementations are feasible as is, we have identified several ways in which they can be optimized, making them significantly more efficient and faster than they already are. Both the implementations are easily parallelizable, making the attacks effectively limited only by how much horizontal computing power an adversary has. Additionally we argue that precomputing the first phase of the known plaintext attack can speed up the process significantly at a cost of memory.

We found additional literature which discuss later developments in mobile communication cryptography and highlight potential security flaws in these. Specifically we treated GSM networks that use the A5/3 cipher and we show how this tries to solve the issues we found with A5/2 by using S-boxes to avoid linearity and by being a block cipher that can not encode before encryption. We also treat LTE networks, that use the SNOW cipher for voice- and text telecommunication. We have found security flaws in literature in both these ciphers and shortly described these.

Additionally we present a short description of other potential security flaws regarding GSM networks, that has nothing directly to do with the encryption standards, though these do rely on the fact that the A5/2 cipher has been broken. Lastly we mention the use of AES in counter mode used for data encryption in the LTE standard and thus we have not found any practically viable attack on this.

Bibliography

3GPP (July 2001). *Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification.*

Barkan, Elad, Biham, Eli and Keller, Nathan (July 2008). 'Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication'. In: *Journal of Cryptology* 21, pp. 392–429.

Biham, Eli, Dunkelman, Orr and Keller, Nathan (Jan. 2005). 'Related-Key Boomerang and Rectangle Attacks'. In: vol. 3494, pp. 507–525.

Biryukov, Alex (May 2004). 'Block Ciphers and Stream Ciphers: The State of the Art'. In:

Biryukov, Alex, Cannière, Christophe and Dellkrantz, Gustaf (Jan. 2003). 'Cryptanalysis of Safer++'. In: pp. 195–211.

Biryukov, Alex and Khovratovich, Dmitry (Dec. 2009). 'Related-key cryptanalysis of the full AES-192 and AES-256'. In: vol. 2009, p. 317.

Briceno, Mark, Goldberg, Ian and Wagner, David (1999). *A pedagogical implementation of the GSM A5/1 and A5/2 "voice privacy" encryption algorithms.* URL: `http://cryptome.org/gsm-a512.htm` (visited on 03/06/2020).

Dunkelman, Orr, Keller, Nathan and Shamir, Adi (Jan. 2010). 'A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony'. In: *IACR Cryptology ePrint Archive* 2010, p. 13.

ETSI (May 2006). *Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification.*

Hong, Seokhie et al. (Jan. 2005). 'Related-key rectangle attacks on reduced versions of SHACAL-1 and AES-192'. In: vol. 3557, pp. 368–383.

Kim, Jongsung et al. (July 2004). 'The related-key rectangle attack - Application to SHACAL-1'. In: vol. 3108, pp. 123–136.

Kircanski, A. and Youssef, A.M. (Dec. 2011). 'On the sliding property of SNOW 3 G and SNOW 2.0'. In: *Information Security, IET* 5, pp. 199–206.

Rupprecht, David et al. (May 2019). 'Breaking LTE on Layer Two'. In: pp. 1121–1136.

Wagner, David (1999). 'The Boomerang Attack'. In: *Proceedings of the 6th International Workshop on Fast Software Encryption.* FSE '99. Berlin, Heidelberg: Springer-Verlag, pp. 156–170.

A Full explanation of the KASUMI block cipher

This somewhat short overview of KASUMI tries to explain everything that is needed to know on the KASUMI cipher. For any further investigations into the cipher we refer to (3GPP 2001).

KASUMI in general is a 64-bit block cipher with 128-bit keys. In the GSM standard however 64-bit keys are used (Barkan et al. 2008).

The cipher has a recursive Feistel structure with 8 rounds, as can be seen by fig. A.1, where each round is composed of 2 functions: the so-called *FO* function that in itself consists of a 3-round 32-bit Feistel construction and the so-called *FL* function, that mixes a 32-bit subkey with the data in a linear way.

The order in which the two functions are applied depends on the round: in even rounds, the *FO* function is applied firstly, and in odd rounds the *FL* function is applied first.

The *FO* function has a recursive structure of it's own. It uses a function called *FI*, that in itself is a four-round Feistel construction. This function uses two types of non-linear S-boxes, *S*7 and *S*9, representing a 7 bit-to-bit permutation and a 9 bit-to-bit permutation respectively. It also accepts a 16-bit subkey, which is used for mixing with the data.
Totally a 96-bit subkey enters the *FO* function in each round – 48 bits are used in the *FI* function and the rest 48 bits are used in key mixing.

The *FL* function uses a 32-bit input and two 16-bit subkey words. One of the subkey words is used to affect the data using the OR operation and the second affects the data using the AND operation.

The structure of KASUMI is outlined in fig. A.1.

The subkeys used in KASUMI are linearly derived from the key. The key, K, is divided into 16-bit words, $K_1, K_2, \cdots$. Each of these is used to compute $K'_i = K_i \oplus C_i$, where C_i is a constant (for a further explanation of these C_i refer to (3GPP 2001)).
In the standard 128-bit key version of KASUMI this gives a total of 8 subkeys and thus 8 K_i for $i = 1, \cdots, 8$.

In each round of KASUMI eight words are used as the round subkey (up to some in-word rotations). This means that each 128-bit round subkey is a linearly modified version of the secret key. (Dunkelman et al. 2010) (3GPP 2001)

The details of the key schedule is shown in table A.1.

Round	$KL_{i,1}$	$KL_{i,2}$	$KO_{i,1}$	$KO_{i,2}$	$KO_{i,3}$	$KI_{i,1}$	$KI_{i,2}$	$KI_{i,3}$
1	$K_1 \lll 1$	K'_3	$K_2 \lll 5$	$K_6 \lll 8$	$K_7 \lll 13$	K'_5	K'_4	K'_8
2	$K_2 \lll 1$	K'_4	$K_3 \lll 5$	$K_7 \lll 8$	$K_8 \lll 13$	K'_6	K'_5	K'_1
3	$K_3 \lll 1$	K'_5	$K_4 \lll 5$	$K_8 \lll 8$	$K_1 \lll 13$	K'_7	K'_6	K'_2
4	$K_4 \lll 1$	K'_6	$K_5 \lll 5$	$K_1 \lll 8$	$K_2 \lll 13$	K'_8	K'_7	K'_3
5	$K_5 \lll 1$	K'_7	$K_6 \lll 5$	$K_2 \lll 8$	$K_3 \lll 13$	K'_1	K'_8	K'_4
6	$K_6 \lll 1$	K'_8	$K_7 \lll 5$	$K_3 \lll 8$	$K_4 \lll 13$	K'_2	K'_1	K'_5
7	$K_7 \lll 1$	K'_1	$K_8 \lll 5$	$K_4 \lll 8$	$K_5 \lll 13$	K'_3	K'_2	K'_6
8	$K_8 \lll 1$	K'_2	$K_1 \lll 5$	$K_5 \lll 8$	$K_6 \lll 13$	K'_4	K'_3	K'_7

$(X \lll i)$ — X rotated to the left by i bits

TABLE A.1: KASUMI's key schedule taken from (Dunkelman et al. 2010, Table 1).

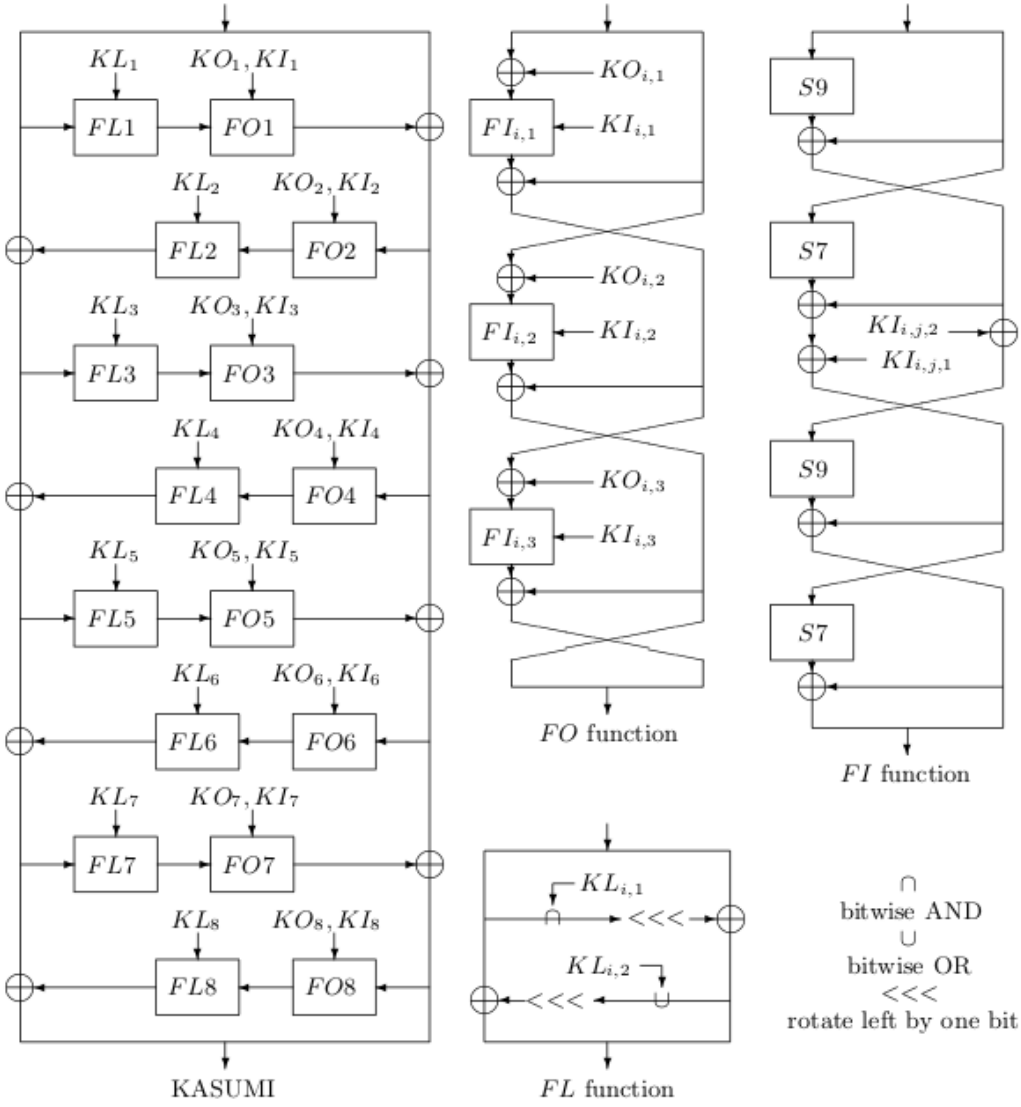

Figure A.1: Outline of KASUMI taken from (Dunkelman et al. 2010, figure 3).

B Overview of GSM

This appendix provides an overview of all aspects of GSM that are important to our project. If further investigation is needed, we recommend reading (Barkan et al. 2008, Appendix B).

In GSM each physical channel can support up to 8 different mobile devices. Each mobile using the same channel is given a time slot in which to transmit in a round-robin fashion.

Each frame consists of 8 such time slots and are identified by their respective number, which is incremented before a new frame starts.

The total amount of transmitted information available is 114 bits per time slot.

We will focus on a single phone and thus limit ourselves to looking at data from a single phone in any given frame.

GSM uses a lot of different types of channels and we will not go into all of them here. (Barkan et al. 2008, Appendix B)

B.1 Call establishment

Call establishment in GSM is done using the following procedure (the first step is only used in case the call is initiated by the network):

1. The network pages the phone with a `PAGING REQUEST` on a specific paging channel
2. Immediate assignment procedure
    a) The phone send a `CHANNEL REQUEST` on the random access channel. This request only includes 8 bits – 5 are used for a random discriminator and the remaining 3 are used for declaring the cause of establishment
    b) The network returns an `IMMEDIATE ASSIGNMENT` containing the random discriminator, details of the allocated channel, the frame number in which the `CHANNEL REQUEST` was received and some other technical information. The mobile immediately tunes to the given channel specified in this message.
3. Service request and contention resolution
    a) The mobile sends a service request message containing a potential paging response or request of some specific service alongside with a local location-specific identification number, called TMSI, and information about the phone (such as the A5 versions supported) and a ciphering key sequence number $(0, \cdots, 6)$
    b) The network acknowledges the service request and sends back the TMSI
4. Authentication
    a) The network sends an authentication request, that includes a random 128-bit value and a ciphering key sequence number, in which the key, K_c, should be stored
    b) The mobile answers with the computed signal response in an authentication response message
    c) The network then sends a message to the mobile, asking the mobile to start encrypting. The network can ask for a specific encryption algorithm and it specifies the encryption key, by a ciphering sequence number $(0, \cdots, 6)$. The network itself switches to deciphering incoming communication

   d) The mobile starts to encrypt/decrypt and responds with an encrypted *cipher* mod *complete message*

5. The call is now in session, but further changes can be sent from the network, specifying things like shifting from the current to another channel – in this case the new channel will also be encrypted

(Barkan et al. 2008, Appendix B.1)

C Code

C.1 A5.h

```
#pragma once

#include <inttypes.h>
#include <immintrin.h>

#include "types.h"

#define FRAMESIZE        114
#define FRAMESIZE_BYTES (FRAMESIZE / 8 + 1)

typedef struct {
  u32_4x registers;
  u32 frame;
  u64 key;
} A5Context;
```

C.2 A5_2.h

```
#pragma once

#include <stdlib.h>
#include <immintrin.h>

#include "A5.h"

// Mask for taps
static u32_4x __tap_masks = {
  .N[0] = 0x072000,
  .N[1] = 0x300000,
  .N[2] = 0x700080,
  .N[3] = 0x010800
};

// Mask for outputs
static u32_4x __output_masks = {
  .N[0] = 0x040000,
  .N[1] = 0x200000,
  .N[2] = 0x400000,
  .N[3] = 0
};
// How many times to shift the outputs to make them the LSB
static u32_4x __output_masks_shift = {
  .N[0] = 18,
  .N[1] = 21,
  .N[2] = 22,
  .N[3] = 0
};

// Masks for what bits of the registers we care about
static u32_4x __register_masks = {
  .N[0] = 0x07FFFF,
  .N[1] = 0x3FFFFF,
  .N[2] = 0x7FFFFF,
  .N[3] = 0x01FFFF
};

// Bits that are forced to at end of setup
static u32_4x __loaded_mask = {
  .N[0] = 0x8000,
  .N[1] = 0x10000,
  .N[2] = 0x40000,
  .N[3] = 0x400,
};
```

```
46
47 #define R4TAP1  0x000400 /* bit 10 */
48 #define R4TAP2  0x000008 /* bit 3 */
49 #define R4TAP3  0x000080 /* bit 7 */
50
51 static inline void
52 set_feedback_bits(A5Context *ctx, u32_4x *parity) {
53   // set every bit not in a feedback tap position to 0.
54   parity->V = _mm_and_si128(ctx->registers.V, __tap_masks.V);
55   // counts the 1s in each set of feedback taps
56   parity->N[0] = _mm_popcnt_u32(parity->N[0]);
57   parity->N[1] = _mm_popcnt_u32(parity->N[1]);
58   parity->N[2] = _mm_popcnt_u32(parity->N[2]);
59   parity->N[3] = _mm_popcnt_u32(parity->N[3]);
60   // sets the feedback bit: if the number of 1's in a tap is uneven the
     ↪  feedback bit is 1, else 0
61   parity->V = _mm_and_si128(parity->V, __one.V);
62 }
63
64 static inline void
65 set_next_register_state(A5Context *ctx, u32_4x *registers, u32_4x *feedback) {
66   // registers <<= 1
67   registers->V = _mm_sllv_epi32(ctx->registers.V, __one.V);
68   // registers &= masks, this discards the most significant bit
69   registers->V = _mm_and_si128(registers->V, __register_masks.V);
70   // registers |= parity, this inserts the new feedback bit in position Ri[0]
71   registers->V = _mm_or_si128(registers->V, feedback->V);
72 }
73
74 static inline void
75 A52_clock_all(A5Context *ctx) {
76   u32_4x parity = __zero;
77
78   set_feedback_bits(ctx, &parity);
79   set_next_register_state(ctx, &(ctx->registers), &parity);
80 }
81
82 static inline void
83 A52_clock_masked(A5Context *ctx, u32_4x *mask) {
84   u32_4x parity = __zero;
85   set_feedback_bits(ctx, &parity);
86
87   u32_4x value = __zero;
88   set_next_register_state(ctx, &value, &parity);
89   // Apply mask to get states of only those registers that should be clocked
90   value.V = _mm_and_si128(mask->V, value.V);
91
```

```
92   // Reverse the mask to get a mask for registers that should not be clocked
93   u32_4x mask_old = { .V = _mm_cmpeq_epi32(mask->V, __zero.V) };
94   // Keep registers corresponding to mask_old intact, insert new register
     ↪  values in registers corresponding to mask
95   ctx->registers.V = _mm_and_si128(ctx->registers.V, mask_old.V);
96   ctx->registers.V = _mm_or_si128(ctx->registers.V, value.V);
97 }
98
99 static inline u08
100 u32_majority(u32 a, u32 b, u32 c) {
101   return a * b ^ a * c ^ b * c;
102 }
103
104 static u32_4x  __control_taps = {
105   .N[0] = 0x000400,
106   .N[1] = 0x000008,
107   .N[2] = 0x000080,
108   .N[3] = 0,
109 };
110
111 static inline void
112 A52_clock(A5Context *ctx) {
113   u32 R4 = ctx->registers.N[3];
114   // Calculate the majority of the control bits in R4
115   u32 majority = u32_majority((R4 & R4TAP1) > 0, (R4 & R4TAP2) > 0, (R4 &
     ↪  R4TAP3) > 0 );
116
117   // Build a clocking mask for R1,R2,R3
118   u32_4x mask;
119   mask.V = _mm_set1_epi32(ctx->registers.N[3]);
120   mask.V = _mm_and_si128(mask.V, __control_taps.V);
121   mask.V = _mm_cmpgt_epi32(mask.V, __zero.V);
122   mask.V = _mm_cmpeq_epi32(mask.V, majority ? __max.V : __zero.V);
123
124   // R4 is set to always be clocked
125   mask.N[3] = 0xFFFFFFFF;
126   A52_clock_masked(ctx, &mask);
127 }
128
129 static inline u08
130 A52_output(A5Context *ctx) {
131   u32_4x value;
132   // Apply output mask to get only bit Ri[n-1] and shift it to LSB position
     ↪  Ri[0]
133   value.V = _mm_and_si128(ctx->registers.V, __output_masks.V);
134   value.V = _mm_srlv_epi32(value.V, __output_masks_shift.V);
135   u08 bit = value.N[0] ^ value.N[1] ^ value.N[2];
```

```
136
137   // Calculate majority bits for all other output taps
138   u08 majority1 = u32_majority((ctx->registers.N[0] & 0x08000) > 0,
      ↪  ((~ctx->registers.N[0]) & 0x04000) > 0,  (ctx->registers.N[0] & 0x1000)
      ↪  > 0);
139   u08 majority2 = u32_majority(((~ctx->registers.N[1]) & 0x10000)
      ↪  >0,(ctx->registers.N[1] & 0x02000) > 0,  (ctx->registers.N[1] & 0x0200)
      ↪  > 0);
140   u08 majority3 = u32_majority((ctx->registers.N[2] & 0x40000) > 0,
      ↪  (ctx->registers.N[2] & 0x10000) > 0, ((~ctx->registers.N[2]) & 0x2000) >
      ↪  0);
141
142   return bit ^ majority1 ^ majority2 ^ majority3;
143 }
144
145 #define FRAME_BITS 22
146 #define KEY_BITS 64
147
148 #define DISCARDED_BITS 99
149
150 static inline void
151 A52_setup(A5Context *ctx, u64 key, u32 frame) {
152   ctx->registers.V = __zero.V;
153   ctx->frame = frame;
154   ctx->key = key;
155
156   // Load key
157   u08 bit, i;
158   for (i = 0; i < KEY_BITS; ++i) {
159     A52_clock_all(ctx);
160
161     bit = ctx->key >> i & 1u;
162     ctx->registers.V = _mm_xor_si128(ctx->registers.V, _mm_set1_epi32(bit));
163   }
164
165   // Load frame
166   for (i = 0; i < FRAME_BITS; ++i) {
167     A52_clock_all(ctx);
168
169     bit = ctx->frame >> i & 1u;
170     ctx->registers.V = _mm_xor_si128(ctx->registers.V, _mm_set1_epi32(bit));
171   }
172   ctx->registers.V = _mm_or_si128(ctx->registers.V, __loaded_mask.V);
173
174   // 99 clocks where the output is discarded.
175   // NOTE: This is the same as running for 100 clocks and using the output
      ↪  with a delay
```

```
176   for(i = 0; i < DISCARDED_BITS; ++i)
177     A52_clock(ctx);
178 }
179
180 static inline void
181 A52_setup_without_clocking_key(A5Context* ctx, u32 frame) {
182   ctx->frame = frame;
183
184   // Load frame
185   u08 bit;
186   for (int i = 0; i < FRAME_BITS; ++i) {
187     A52_clock_all(ctx);
188
189     bit = ctx->frame >> i & 1u;
190     ctx->registers.V = _mm_xor_si128(ctx->registers.V, _mm_set1_epi32(bit));
191   }
192   ctx->registers.V = _mm_or_si128(ctx->registers.V, __loaded_mask.V);
193
194   // 99 clocks where the output is discarded.
195   // NOTE: This is the same as running for 100 clocks and using the output
      ↪  with a delay
196   for (int i = 0; i < DISCARDED_BITS; ++i)
197     A52_clock(ctx);
198 }
199
200 #define INCOMING_KEYSTREAM_BITS FRAMESIZE
201 #define OUTGOING_KEYSTREAM_BITS FRAMESIZE
202 #define INCOMING_KEYSTREAM_BYTES FRAMESIZE_BYTES
203 #define OUTGOING_KEYSTREAM_BYTES FRAMESIZE_BYTES
204
205 static inline void
206 A52(A5Context *ctx, u08 *incoming, u08* outgoing) {
207   // Incoming = Network -> Phone, outgoing = Phone -> Network
208   u08 bit, i;
209
210   // The first 114 keystream bits of output are used for encrypting the
      ↪  incoming link from the Network to the Phone
211   for (i = 0; i < INCOMING_KEYSTREAM_BITS; ++i) {
212     A52_clock(ctx);
213     bit = A52_output(ctx);
214
215     incoming[i / 8] |= bit << (7 - (i & 7));
216   }
217
218   // The next 114 keystream bits of output are used for encrypting the
      ↪  outgoing link from the Phone to the Network
219   for (i = 0; i < OUTGOING_KEYSTREAM_BITS; ++i) {
```

```
220     A52_clock(ctx);
221     bit = A52_output(ctx);
222
223     outgoing[i / 8] |= bit << (7 - (i & 7));
224   }
225 }
```

C.3 ciphertext.c

```c
#include <stdio.h>

#define DEBUG
#define GUESS_RANGE 300

#define CIPHERTEXT_EXTRA_MESSAGES 5

#include "A5_2.h"
#include "ciphertext.h"
#include "matrix.h"
#include "stretchy_buffer.h"
#include "plaintext.h"

int
main(int argc, char **argv) {
  const u32 firstFrame = 0x68947356;
  const u64 key = 0x82EFD6456DAF;
  const u32 correctR4 = 0xC07F;

  // Generate 3 * 184 bits of plaintext to encode
  char plaintextStrings[CIPHERTEXT_MESSAGES_REQUIRED][CIPHERTEXT_P_SIZE / 8] =
  ↪ {
    "Something clever here.\0",
    "What about in this one\0",
    "This is a message str.\0"
  };
  u08 plaintexts[CIPHERTEXT_MESSAGES_REQUIRED][CIPHERTEXT_P_SIZE];
  for(int i = 0; i < CIPHERTEXT_P_SIZE; ++i) {
    const int byte = i / 8;
    const int bit = i % 8;

    for(int j = 0; j < CIPHERTEXT_MESSAGES_REQUIRED; ++j) {
      plaintexts[j][i] = (plaintextStrings[j][byte] >> bit) & 1;
    }
  }

  // Encoding the plaintext using our parity matrix
  u08 *messages[CIPHERTEXT_MESSAGES_REQUIRED];
  u08 *GT = parity_generate_gt(2);
  u08 *H = parity_generate_h(GT);
  for (int i = 0; i < CIPHERTEXT_MESSAGES_REQUIRED; ++i) {
    messages[i] = matrix_multiply(GT, G_WIDTH, G_HEIGHT, plaintexts[i], 1,
    ↪  G_WIDTH);
  }

```

```
44  // Generate the extra messages
45  char extraPlaintextStrings[CIPHERTEXT_EXTRA_MESSAGES][CIPHERTEXT_P_SIZE / 8]
    ↪  = {
46    "Lorem ipsum dolor sit \0",
47    "amet, consectetur adip\0",
48    "ctus orci, justo et le\0",
49    "justo a, tempus tempor\0",
50    "iscing elit. Aenean le\0",
51  };
52  u08 extraPlaintexts[CIPHERTEXT_EXTRA_MESSAGES][CIPHERTEXT_P_SIZE];
53  for (int i = 0; i < CIPHERTEXT_P_SIZE; ++i) {
54    const int byte = i / 8;
55    const int bit = i % 8;
56
57    for (int j = 0; j < CIPHERTEXT_EXTRA_MESSAGES; ++j) {
58      extraPlaintexts[j][i] = (extraPlaintextStrings[j][byte] >> bit) & 1;
59    }
60  }
61
62  // Encode the extra messages
63  u08 *extraMessages[CIPHERTEXT_EXTRA_MESSAGES];
64  for (int i = 0; i < CIPHERTEXT_EXTRA_MESSAGES; ++i) {
65    extraMessages[i] = matrix_multiply(GT, G_WIDTH, G_HEIGHT,
    ↪  extraPlaintexts[i], 1, G_WIDTH);
66  }
67
68  // Setup the A5 context
69  //A5Context cipher;
70  A5Context cipher;
71  u32 frameNumber = firstFrame;
72  u08 incomingKeyStream[FRAMESIZE_BYTES], outgoingKeyStream[FRAMESIZE_BYTES];
    ↪  // NOTE: The function needs both arrays, but we only use one of them
73
74  // Setup the ciphertext attack
75  CiphertextAttack attack;
76  ciphertext_init(&attack);
77  ciphertext_add_paritycheck_matrix(&attack, H);
78
79  // Split each message into 4 frames, encrypt them using A5/2
80  int nextCiphertext = 0;
81  u08 ciphertexts[CIPHERTEXT_STATES_REQUIRED][FRAMESIZE];
82  for(int i = 0; i < CIPHERTEXT_MESSAGES_REQUIRED; ++i) {
83    u08 *message = messages[i];
84
85    // Split into 4 frames
86    for(int j = 0; j < FRAMES_PER_MESSAGE; ++j) {
87      memset(incomingKeyStream, 0, FRAMESIZE_BYTES);
```

```
88
89        // A5/2 Encryption
90        A52_setup(&cipher, key, frameNumber);
91        A52(&cipher, incomingKeyStream, outgoingKeyStream);
92
93        for(int k = 0; k < FRAMESIZE; ++k) {
94          const int byte = k / 8;
95          const int bit = 7 - k % 8;
96          const u08 keyBit = (incomingKeyStream[byte] >> bit) & 1;
97
98          ciphertexts[nextCiphertext][k] = message[k] ^ keyBit;
99        }
100
101       ciphertext_add_frame(&attack, frameNumber, ciphertexts[nextCiphertext]);
102
103       ++nextCiphertext;
104       ++frameNumber;
105
106       message += FRAMESIZE;
107     }
108
109     //free(messages[i]);
110   }
111
112   // Split each extra message into 4 frames and encrypt
113   nextCiphertext = 0;
114   u32 extraFrameNumbers[CIPHERTEXT_EXTRA_MESSAGES * FRAMES_PER_MESSAGE];
115   u08 extraCiphertexts[CIPHERTEXT_EXTRA_MESSAGES *
      ↪  FRAMES_PER_MESSAGE][FRAMESIZE];
116   for (int m = 0; m < CIPHERTEXT_EXTRA_MESSAGES; ++m) {
117     u08 *message = extraMessages[m];
118
119     // Split into 4 frames
120     for (int f = 0; f < FRAMES_PER_MESSAGE; ++f) {
121       memset(incomingKeyStream, 0, FRAMESIZE_BYTES);
122
123       // A5/2 Encryption
124       A52_setup(&cipher, key, frameNumber);
125       A52(&cipher, incomingKeyStream, outgoingKeyStream);
126
127       extraFrameNumbers[nextCiphertext] = frameNumber;
128       for (int k = 0; k < FRAMESIZE; ++k) {
129         const int byte = k / 8;
130         const int bit = 7 - k % 8;
131         const u08 keyBit = (incomingKeyStream[byte] >> bit) & 1;
132
133         extraCiphertexts[nextCiphertext][k] = message[k] ^ keyBit;
```

```
134       }
135
136       ++frameNumber;
137       ++nextCiphertext;
138             message += FRAMESIZE;
139     }
140
141     //free(messages);
142   }
143
144   // Try and guess the correct value for R4
145   u32 *validGuesses = NULL;
146   Registers *validSolutions = NULL;
147   for (u32 guess = correctR4 - GUESS_RANGE; guess <= correctR4 + GUESS_RANGE;
      ↪ ++guess) {
148     Registers solution;
149     const int result = ciphertext_guess_r4(&attack, guess, &solution);
150     if (result) {
151       printf("\033[32mValid\033[0m guess: R4 = 0x%08X\n", guess);
152       sb_push(validGuesses, guess);
153       sb_push(validSolutions, solution);
154     }
155     else {
156       printf("\033[31mInvalid\033[0m guess: R4 = 0x%08X\n", guess);
157     }
158   }
159
160   printfn;
161   printf("Found %i valid guess(es) for R4:\n", sb_count(validGuesses));
162   for (int i = 0; i < sb_count(validGuesses); ++i) {
163     printf("R4 = 0x%08X\n", validGuesses[i]);
164   }
165   printfn;
166
167   // For each valid guess, use the cipher to generate the keystream for the
      ↪ guess
168   // compare with the known keystream to find the right guess
169   union
170   {
171     u08 frames[FRAMES_PER_MESSAGE][FRAMESIZE];
172     u08 message[CIPHERTEXT_M_SIZE];
173   } decrypted[CIPHERTEXT_MESSAGES_REQUIRED + CIPHERTEXT_EXTRA_MESSAGES];
174   u08 keystream[FRAMESIZE_BYTES];
175
176   printf("Comparing keystreams for guesses with known good:\n");
177   for (int i = 0; i < sb_count(validGuesses); ++i) {
178     const u32 guess = validGuesses[i];
```

```
179     const Registers solution = validSolutions[i];
180
181     // Reverse clock the frame numbers
182     const u32 r1 = plaintext_reverse_clock_framenumber(solution.r1,
        ↪  __tap_masks.N[0], __register_masks.N[0], firstFrame);
183     const u32 r2 = plaintext_reverse_clock_framenumber(solution.r2,
        ↪  __tap_masks.N[1], __register_masks.N[1], firstFrame);
184     const u32 r3 = plaintext_reverse_clock_framenumber(solution.r3,
        ↪  __tap_masks.N[2], __register_masks.N[2], firstFrame);
185
186     // Decrypt each of the 4 frames in each of the 3 messages
187     for(int m = 0; m < CIPHERTEXT_MESSAGES_REQUIRED; ++m) {
188       KnownMessage *message = attack.messages + m;
189
190           for(int f = 0; f < FRAMES_PER_MESSAGE; ++f) {
191         cipher.registers.N[0] = r1;
192         cipher.registers.N[1] = r2;
193         cipher.registers.N[2] = r3;
194         cipher.registers.N[3] = guess;
195         A52_setup_without_clocking_key(&cipher, message->frameNumbers[f]);
196
197         memset(keystream, 0, FRAMESIZE_BYTES);
198         A52(&cipher, keystream, outgoingKeyStream);
199
200         // Decrypt the frame
201         for (int k = 0; k < FRAMESIZE; ++k) {
202           const int b = k / 8;
203           const int bit = 7 - k % 8;
204           const u08 keyBit = (keystream[b] >> bit) & 1;
205
206           decrypted[m].frames[f][k] = message->frames[f][k] ^ keyBit;
207         }
208             }
209     }
210
211     // Decrypt the extra messages
212     nextCiphertext = 0;
213     for (int m = 0; m < CIPHERTEXT_EXTRA_MESSAGES; ++m) {
214       for (int f = 0; f < FRAMES_PER_MESSAGE; ++f) {
215         u32 frame = extraFrameNumbers[m * FRAMES_PER_MESSAGE + f];
216         cipher.registers.N[0] = r1;
217         cipher.registers.N[1] = r2;
218         cipher.registers.N[2] = r3;
219         cipher.registers.N[3] = guess;
220         A52_setup_without_clocking_key(&cipher, frame);
221
222         memset(keystream, 0, FRAMESIZE_BYTES);
```

```
223         A52(&cipher, keystream, outgoingKeyStream);
224
225         // Decrypt the frame
226         for (int k = 0; k < FRAMESIZE; ++k) {
227           const int b = k / 8;
228           const int bit = 7 - k % 8;
229           const u08 keyBit = (keystream[b] >> bit) & 1;
230
231           decrypted[CIPHERTEXT_MESSAGES_REQUIRED + m].frames[f][k] =
              ↪  extraCiphertexts[nextCiphertext][k] ^ keyBit;
232         }
233
234         ++nextCiphertext;
235       }
236     }
237
238     // Check the parity for each of the messages
239     u08 parity = 0;
240     for(int m = 0; m < CIPHERTEXT_MESSAGES_REQUIRED +
        ↪  CIPHERTEXT_EXTRA_MESSAGES; ++m) {
241       u08 *parityResult = matrix_multiply(H, H_WIDTH, H_HEIGHT,
          ↪  decrypted[m].message, 1, H_WIDTH);
242       u08 *parity_result_copy = parityResult;
243       for (int h = 0; h < H_HEIGHT; h++) {
244         parity += *parity_result_copy;
245         parity_result_copy++;
246       }
247
248       free(parityResult);
249     }
250
251     // Compare the actual keystream with the expected keystream
252     if (parity == 0) {
253       printf("\033[32mCorrect\033[0m guess R4 = 0x%08X \n", guess);
254       printf("R1 = 0x%08X\n", r1);
255       printf("R2 = 0x%08X\n", r2);
256       printf("R3 = 0x%08X\n", r3);
257       printf("R4 = 0x%08X\n", guess);
258       //return 0;
259
260       continue;
261     }
262
263     printf("\033[31mIncorrect\033[0m guess R4 = 0x%08X \n", guess);
264   }
265
266   return 1;
267 }
```

C.4 ciphertext.h

```
#pragma once

#include "state.h"
#include "types.h"
#include "solve.h"
#include "parity.h"

#define CIPHERTEXT_P_SIZE      184
#define CIPHERTEXT_M_SIZE      456
#define FRAMES_PER_MESSAGE       (CIPHERTEXT_M_SIZE / FRAMESIZE)
#define CIPHERTEXT_MESSAGES_REQUIRED 3
#define CIPHERTEXT_STATES_REQUIRED   (CIPHERTEXT_MESSAGES_REQUIRED *
↪  FRAMES_PER_MESSAGE)
#define CIPHERTEXT_EQUATIONS     (CIPHERTEXT_MESSAGES_REQUIRED * 272)

typedef struct
{
        State states[FRAMES_PER_MESSAGE];
        u32 frameNumbers[FRAMES_PER_MESSAGE];

        union
        {
                u08 message[CIPHERTEXT_M_SIZE];
                u08 frames[FRAMES_PER_MESSAGE][FRAMESIZE];
        };

        int nextFrame;
} KnownMessage;

typedef struct
{
        u08 H[H_SIZE];

        KnownMessage messages[CIPHERTEXT_MESSAGES_REQUIRED];
        int nextMessage;

        u08 *matrixUnalligned;
        Row *matrix;

        Vector *outputs; // lhs of the system of equations
        u08 *values;   // rhs of the system of equations
} CiphertextAttack;

static void
ciphertext_init(CiphertextAttack *attack) {
```

```
45          attack->nextMessage = 0;
46          for(size_t i = 0; i < ARRAY_SIZE(attack->messages); ++i) {
47                  attack->messages[i].nextFrame = 0;
48          }
49
50          // Allocate memory and allign to 32 bytes
51          attack->matrixUnalligned = malloc(sizeof(Row) * CIPHERTEXT_EQUATIONS +
            ↪  32);
52          attack->matrix = (Row *) (attack->matrixUnalligned + (size_t)
            ↪  attack->matrixUnalligned % 32);
53          attack->values = malloc(sizeof(u08) * CIPHERTEXT_EQUATIONS);
54          attack->outputs = malloc(sizeof(Vector) * CIPHERTEXT_EQUATIONS);
55  }
56
57  static void
58  ciphertext_add_frame(CiphertextAttack *attack, u32 frame, u08 *ciphertext) {
59          if(attack->messages[attack->nextMessage].nextFrame >=
            ↪  FRAMES_PER_MESSAGE) {
60                  ++attack->nextMessage;
61          }
62
63          KnownMessage *message = attack->messages + attack->nextMessage;
64          memcpy(message->frames[message->nextFrame], ciphertext, FRAMESIZE);
65          message->frameNumbers[message->nextFrame] = frame;
66          ++message->nextFrame;
67  }
68
69  static void
70  ciphertext_add_paritycheck_matrix(CiphertextAttack *attack, u08 *matrix) {
71          memcpy(attack->H, matrix, H_SIZE);
72  }
73
74  static Vector *
75  ciphertext_get_vector_for_bit(KnownMessage *message, int idx) {
76          // Index is in the first frame
77          if (idx < FRAMESIZE * 1) {
78                  return message->states[0].output[idx];
79          }
80
81          // Index is in the second frame
82          if(idx < FRAMESIZE * 2) {
83                  const int offset = idx - FRAMESIZE * 1;
84                  return message->states[1].output[offset];
85          }
86
87          // Index is in the third frame
88          if(idx < FRAMESIZE * 3) {
```

```
89                  const int offset = idx - FRAMESIZE * 2;
90                  return message->states[2].output[offset];
91          }
92
93          // Index is in the fourth frame
94          const int offset = idx - FRAMESIZE * 3;
95          return message->states[3].output[offset];
96  }
97
98  static void
99  ciphertext_populate_matrix(CiphertextAttack *attack) {
100         for (int i = 0; i < CIPHERTEXT_EQUATIONS; ++i) {
101                 Row *row = attack->matrix + i;
102                 row->value = attack->values[i];
103
104                 Vector *vector = attack->outputs + i;
105                 row->constant = vector->constant;
106
107                 for (u32 c = 0; c < ARRAY_SIZE(vector->data); ++c) {
108                         row->coefficients[c] = vector->data[c];
109                 }
110         }
111 }
112
113 static int
114 ciphertext_guess_r4(CiphertextAttack *attack, const u32 r4, Registers
    ↪  *solution) {
115         Vector *output = attack->outputs;
116         u08 *outputValue = attack->values;
117
118         int count = 0;
119         State *initialState = &(attack->messages[0].states[0]);
120         for(int i = 0; i < CIPHERTEXT_MESSAGES_REQUIRED; ++i) {
121                 KnownMessage *message = attack->messages + i;
122                 u08 *parity = matrix_multiply(attack->H, H_WIDTH, H_HEIGHT,
                    ↪  message->message, 1, H_WIDTH);
123
124                 u08 *parityBit = parity;
125                 u08 *parityRow = attack->H;
126
127                 for(int j = 0; j < FRAMES_PER_MESSAGE; ++j) {
128                         State *state = message->states + j;
129                         state_setup(r4, message->frameNumbers[j], state,
                            ↪  initialState);
130                         state_cipher(state);
131                 }
132
```

```
133                 // From the second frame in the message we get 114 - (184 -
                    ↪  114) = 44 equations
134                 // From the 3rd and 4th everything gives us an equation
135                 // Build the vector for the output bit from the parity matrix
136                 for(int f = 1; f < 4; ++f) {
137                         for(int k = f == 1 ? 70 : 0; k < FRAMESIZE; ++k) {
138                                 *outputValue++ = *parityBit++;
139
140                                 vector_set(output, 0, 0);
141                                 for (int h = 0; h < H_WIDTH; ++h) {
142                                         if (parityRow[h]) {
143                                                 Vector *vector =
                                                    ↪  ciphertext_get_vector_for_bit(message,
                                                    ↪  h);
144                                                 vector_xor(output, vector);
145                                         }
146                                 }
147
148                                 parityRow += H_WIDTH;
149                                 ++output;
150                                 ++count;
151                         }
152                 }
153
154                 free(parity);
155         }
156
157         // Attempt to solve the matrix
158         ciphertext_populate_matrix(attack);
159         for(int m = 0; m < CIPHERTEXT_MESSAGES_REQUIRED; ++m) {
160                 KnownMessage *message = attack->messages + m;
161
162                 for(int f = 0; f < FRAMES_PER_MESSAGE; ++f) {
163                         State *state = message->states + f;
164                         state_free(state);
165                 }
166         }
167
168         return solve(attack->matrix, CIPHERTEXT_EQUATIONS, solution);
169 }
```

C.5 matrix.h

```
#pragma once

#include <stdlib.h>
#include <assert.h>
#include "types.h"

static u08 *
matrix_transpose(u08 *matrix, int width, int height) {
  u08 *result = (u08 *) malloc(width * height);
  u08 *result_copy = result;
  u08 *input_row = matrix;
  for (size_t i = 0; i < width; i++)
  {
    u08 *row_copy = input_row;

    for (size_t j = 0; j < height; j++)
    {
      *result_copy = *row_copy;
      result_copy++;
      row_copy += width;
    }

    input_row++;
  }

  return result;
}

static u08 *
matrix_multiply(u08 *left, int l_width, int l_height, u08 *right, int r_width,
↪  int r_height) {
  assert(r_height == l_width);
  u08 *result = (u08 *) malloc(r_width * l_height);
  u08 *result_copy = result;
  u08 *lrow_copy = left;
  for (size_t lrow = 0; lrow < l_height; lrow++)
  {
    u08 *rcolumn_copy = right;
    for (size_t rcolumn = 0; rcolumn < r_width; rcolumn++)
    {
      u08 ij_value = 0;
      for (size_t k = 0; k < r_height; k++)
      {
        //printf("k: %i, j: %i --> %i\n", k, rrow, k * r_width + rrow);
        //printf("j: %i, k: %i --> %i\n", rrow, k, (rrow+1)*l_width + k);
```

```
45          ij_value ^= (*(lrow_copy + k)) * (*(rcolumn_copy + k *
             ↪  r_width));//left[rrow*l_width + k]*right[k*r_width + rrow];
46        }
47
48        *result_copy = ij_value;
49        result_copy++;
50
51        rcolumn_copy++;
52      }
53      lrow_copy += l_width;
54    }
55    return result;
56  }
```

C.6 parity.h

```
#pragma once

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "types.h"
#include "matrix.h"

#define G_WIDTH  184
#define G_HEIGHT 456
#define H_WIDTH   G_HEIGHT
#define H_HEIGHT (G_HEIGHT - G_WIDTH)
#define H_SIZE   (H_WIDTH * H_HEIGHT)

static u08*
parity_generate_gt(int seed) {
        srand(seed);
        u08 *matrix = (u08*) calloc(G_WIDTH * G_HEIGHT, sizeof(u08));
        u08 *row = matrix;
        //build the identity matrix part
        for (size_t i = 0; i < G_WIDTH; i++) {
                row[i] = 1;
                //row += G_HEIGHT;
                //for (size_t j = 0; j < G_WIDTH - G_HEIGHT; j++) {
                //        row[j] = rand() % 2;
                //}
                row += G_WIDTH;
        }
        //build the random matrix part
        for        (size_t j = 0; j < (G_HEIGHT - G_WIDTH) * G_WIDTH; j++) {
                row[j] = rand() % 2;
        }

        return matrix;
}

static u08*
parity_generate_h(u08 *g) {
        u08 *matrix = (u08*) calloc(H_SIZE, sizeof(u08));

        u08 *g_copy = g;
        g_copy += G_WIDTH * G_WIDTH;
        u08 *row_random = matrix;

```

```
        for (size_t i = 0; i < H_HEIGHT; ++i) {
                for (size_t j = 0; j < H_WIDTH; ++j) {
                        if (j < 184) {
                                *row_random++ = *g_copy++;
                        }
                        else {
                                *row_random++ = (j - 184) == i;
                        }
                }
        }

        return matrix;
}


static void
save_matrix(char *fileName, u08 *matrix, int height, int width) {
  FILE *file = fopen(fileName, "wb");
  for (int r = 0; r < height; r++) {
    u08 *row = matrix + r*width;
    for (int c = 0; c < width; c++) {
      fprintf(file, "%c ", row[c] ? '1' : '0');
    }
    //fprintf(file, "%c", *row);

    fprintf(file, "\n");
  }

  fflush(file);
  fclose(file);
}
```

C.7 plaintext.c

```c
#include <stdio.h>

#define DEBUG
#define GUESS_RANGE 10

#include "A5_2.h"
#include "plaintext.h"
#include "stretchy_buffer.h"

int
main(int argc, char **argv) {
  const u32 firstFrame = 0x68947356;
  const u64 key = 0x82efd6456daf;
  const u32 correctR4 = 0xC07F;

  PlaintextAttack attack;
  plaintext_init(&attack);

  // Run the cipher to generate the key streams we need
  A5Context cipher;
  u08 incomingKeyStream[15], outgoingKeyStream[15];
  for (int i = 0; i < PLAINTEXT_FRAMES_REQUIRED; ++i) {
    const u32 frame = firstFrame + i;

    memset(incomingKeyStream, 0, FRAMESIZE_BYTES);
    memset(outgoingKeyStream, 0, FRAMESIZE_BYTES);

    A52_setup(&cipher, key, frame);
    A52(&cipher, incomingKeyStream, outgoingKeyStream);

    // Add the known frame to the attack
    plaintext_add_frame(&attack, frame, incomingKeyStream, outgoingKeyStream);
  }

  // Try and guess the correct value for R4
  u32 *validGuesses = NULL;
  Registers *validSolutions = NULL;
  for(u32 guess = correctR4 - GUESS_RANGE; guess <= correctR4 + GUESS_RANGE;
  ↪ ++guess) {
    Registers solution;
    const int result = plaintext_guess_r4(&attack, guess, &solution);
    if (result) {
      printf("\033[32mValid\033[0m guess: R4 = 0x%08X\n", guess);
      sb_push(validGuesses, guess);
      sb_push(validSolutions, solution);
```

```
45      } else {
46        printf("\033[31mInvalid\033[0m guess: R4 = 0x%08X\n", guess);
47      }
48    }
49
50    printfn;
51    printf("Found %i valid guess(es) for R4:\n", sb_count(validGuesses));
52    for(int i = 0; i < sb_count(validGuesses); ++i) {
53      printf("R4 = 0x%08X\n", validGuesses[i]);
54    }
55    printfn;
56
57    // For each valid guess, use the cipher to generate the keystream for the
      ↪  guess
58    // compare with the known keystream to find the right guess
59    printf("Comparing keystreams for guesses with known good:\n");
60    u08 *firstFrameKeyStream = attack.frames[0].incoming;
61    for (int i = 0; i < sb_count(validGuesses); ++i) {
62      const u32 guess = validGuesses[i];
63      const Registers solution = validSolutions[i];
64
65      // Unclock the frame number
66      const u32 r1 = plaintext_reverse_clock_framenumber(solution.r1,
        ↪  __tap_masks.N[0], __register_masks.N[0], firstFrame);
67      const u32 r2 = plaintext_reverse_clock_framenumber(solution.r2,
        ↪  __tap_masks.N[1], __register_masks.N[1], firstFrame);
68      const u32 r3 = plaintext_reverse_clock_framenumber(solution.r3,
        ↪  __tap_masks.N[2], __register_masks.N[2], firstFrame);
69
70      // Set the register values to what they would be after clocking in the
        ↪  key
71      cipher.registers.N[0] = r1;
72      cipher.registers.N[1] = r2;
73      cipher.registers.N[2] = r3;
74      cipher.registers.N[3] = guess;
75      A52_setup_without_clocking_key(&cipher, firstFrame);
76
77      memset(incomingKeyStream, 0, FRAMESIZE_BYTES);
78      A52(&cipher, incomingKeyStream, outgoingKeyStream);
79
80      // Compare the actual keystream with the expected keystream
81      if(memcmp(firstFrameKeyStream, incomingKeyStream, FRAMESIZE_BYTES) == 0)
        ↪  {
82        printf("\033[32mCorrect\033[0m guess R4 = 0x%08X \n", guess);
83        printfn;
84
85        printf("Internal state after clocking in key: \n");
```

```
86        printf("R1 = 0x%08X\n", r1);
87        printf("R2 = 0x%08X\n", r2);
88        printf("R3 = 0x%08X\n", r3);
89        printf("R4 = 0x%08X\n", guess);
90        return 0;
91      }
92
93      printf("\033[31mIncorrect\033[0m guess R4 = 0x%08X \n", guess);
94    }
95
96    return 1;
97  }
```

C.8 plaintext.h

```c
#pragma once

#include "state.h"
#include "types.h"
#include "solve.h"

#define PLAINTEXT_FRAMES_REQUIRED 4
#define PLAINTEXT_MATRIX_HEIGHT   (PLAINTEXT_FRAMES_REQUIRED * FRAMESIZE * 2)

typedef struct {
  u32 frameNumber;
  u08 incoming[15];
  u08 outgoing[15];
} KnownFrame;

typedef struct {
  KnownFrame frames[PLAINTEXT_FRAMES_REQUIRED];
  State states[PLAINTEXT_FRAMES_REQUIRED];
  int nextFrame;

  u08 *matrixUnalligned;
  Row *matrix;
} PlaintextAttack;

static void
plaintext_init(PlaintextAttack *ctx) {
  ctx->nextFrame = 0;

  // Allocate memory and allign to 32 bytes
  ctx->matrixUnalligned = malloc(sizeof(Row) * PLAINTEXT_MATRIX_HEIGHT + 32);
  ctx->matrix = (Row *) (ctx->matrixUnalligned + (size_t)
  ↪  ctx->matrixUnalligned % 32);
}

static void
plaintext_add_frame(PlaintextAttack *ctx, const u32 frameNumber, const u08
↪  *incoming, const u08 *outgoing) {
  ctx->frames[ctx->nextFrame].frameNumber = frameNumber;
  for (int i = 0; i < 15; ++i) {
    ctx->frames[ctx->nextFrame].incoming[i] = incoming[i];
    ctx->frames[ctx->nextFrame].outgoing[i] = outgoing[i];
  }

  ++ctx->nextFrame;
}
```

```
44
45 static void
46 plaintext_populate_matrix(PlaintextAttack *ctx) {
47   // Create the matrix
48   // NOTE: Matrix on form [A|c|b], where A is the matrix of coefficients, c
     ↪  the constants and b the vector of the rhs.
49   int workingRow = -1;
50   for (int s = 0; s < PLAINTEXT_FRAMES_REQUIRED; ++s) {
51     const State state = ctx->states[s];
52     u08 *inc = ctx->frames[s].incoming;
53     u08 *out = ctx->frames[s].outgoing;
54
55     for (int r = 0; r < FRAMESIZE * 2; ++r) {
56       Vector *vector = state.output[r];
57       Row *row = ctx->matrix + (++workingRow);
58
59       for (u32 c = 0; c < ARRAY_SIZE(vector->data); ++c) {
60         row->coefficients[c] = vector->data[c];
61       }
62
63       // NOTE: The first output vector will be for the first bit of the
         ↪  keystream
64       //     IE: the last bit in our keystream array
65       u08 keystreamValue;
66       if (r < FRAMESIZE) {
67         keystreamValue = (inc[r / 8] >> (7 - (r & 7))) & 1;
68       }
69       else {
70         const int i = r - 114;
71         keystreamValue = (out[i / 8] >> (7 - (i & 7))) & 1;
72       }
73
74       row->constant = vector->constant;
75       row->value = keystreamValue;
76     }
77   }
78 }
79
80 static int
81 plaintext_guess_r4(PlaintextAttack *ctx, u32 r4, Registers *solution) {
82   // Run through the tracking for each of the frames
83   for (int i = 0; i < PLAINTEXT_FRAMES_REQUIRED; ++i) {
84     KnownFrame *frame = ctx->frames + i;
85     state_setup(r4, frame->frameNumber, ctx->states + i, ctx->states);
86     state_cipher(ctx->states + i);
87   }
88
```

```
89    // Generate the linear system of equations
90    plaintext_populate_matrix(ctx);
91    for (int f = 0; f < PLAINTEXT_FRAMES_REQUIRED; ++f) {
92      state_free(&ctx->states[f]);
93    }
94
95    // Attempt to solve the system
96    return solve(ctx->matrix, PLAINTEXT_MATRIX_HEIGHT, solution);
97  }
98
99  static u32
100 plaintext_reverse_clock_framenumber(u32 reg, const u32 taps, const u32 mask,
    ↪  const u32 frame) {
101   const u32 clockedTapMask = (taps << 1) & mask;
102   const u32 lastBit = 1 << (_mm_popcnt_u32(mask) - 1);
103
104   for (int i = 21; i >= 0; --i) {
105     const u32 tap = reg & clockedTapMask;
106     const u32 sum = _mm_popcnt_u32(tap);
107
108     const u08 frameBit = (frame >> i) & 1;
109     const u08 parity = (reg & 1) ^ frameBit; // Last bit of the frame number
110
111     // Figure out the value of the tap (t1) which was clocked out
112     // Sum is odd => t1 = !parity otherwise t1 = parity
113     const u08 t1 = (sum & 1) ? !parity : parity;
114
115     // Reverse clock
116     reg >>= 1;
117     reg |= lastBit * t1;
118   }
119
120   return reg;
121 }
```

C.9 solve.h

```
#pragma once

#include <immintrin.h>
#include <stdio.h>

#include "A5.h"
#include "types.h"
#include "vector.h"
#include "state.h"

#define NUMBER_OF_UNKNOWNS     720 // (the same as columns - 1)
#define MATRIX_CONSTANT_COLUMN   719

#define VARIABLES_FROM_REGISTERS (R1_LEN + R2_LEN + R3_LEN)
#define VARIABLES_FROM_PRODUCTS  (R1P_LEN + R2P_LEN + R3P_LEN)

typedef struct
{
  u32 r1, r2, r3;
} Registers;

typedef enum
{
  zero      = 0,
  one       = 1,
  unknown     = 2,
  free_variable = 3
} Variable;

// Represents a row in the matrix
typedef union
{
  // 719 coefficients (64 variables, the rest are products), a constant and a
   ↪  value (right hand side)
  // representing an equation from the linear system
  struct
  {
    u08 coefficients[719];
    u08 constant;
    u08 value;
  };

  u08 columns[721];

```

```
  // We can also treat this as an array of vectors for bytes packed into a __m128i (see types.h for u08_x32)
  // This allows us to optimize the elimination code
  u08_x32 V[23];
} Row;

static inline void
row_xor(Row *left, Row *right) {
  for (int i = 0; i < 23; ++i) {
    u08_x32 leftVi = left->V[i];
    u08_x32 rightVi = right->V[i];
    u08_x32* leftPointer = &(left->V[i]);
    leftPointer->V = _mm256_xor_si256(leftVi.V, rightVi.V);
  }
}

static inline void
row_swap(Row *left, Row *right) {
  row_xor(left, right);
  row_xor(right, left);
  row_xor(left, right);
}

static void
simplify_products(Variable *variables, int vLen, Variable *products, int pLen) {
  int lhs, rhs;
  for (int i = 0; i < pLen; ++i) {
    product_2d_index(vLen, i, &lhs, &rhs);
    const Variable left = variables[lhs];
    const Variable right = variables[rhs];

    // We know both variables, but not the product, we can compute it
    if((products[i] == unknown  || products[i] == free_variable) && left < unknown && right < unknown) {
      products[i] = left * right;
      debug_printf("  Computed RP_%03i = R_%02i * R_%02i = %i * %i = %i\n", i, lhs, rhs, left, right, products[i]);
    }
    // If either variable is 0, the product must also be 0
    else if (left == 0 || right == 0) {
            products[i] = 0;
      debug_printf("  RP_%03i = R_%02i * R_%02i = %i * %i => RP_%03i = %i\n", i, lhs, rhs, left, right, i, products[i]);
    }
    // The product is 1, so both variables must also be 1
    else if (products[i] == 1) {
```

```
86        variables[lhs] = 1;
87        variables[rhs] = 1;
88        debug_printf("  RP_%03i = 1 => R_%02i = 1, R_%02i = 1 \n", i, lhs,
          ↪  rhs);
89      }
90      // If the product is 0 and one of the variables 1, we know that the other
        ↪  variable must be 0
91      else if (products[i] == 0 && ((left == 1 && right == unknown) || (left ==
        ↪  unknown && right == 1))) {
92        const int variable = left == 1 ? rhs : lhs;
93        variables[variable] = 0;
94
95        debug_printf("  RP_%03i = R_%02i * R_%02i = 1 * a => a = 0 = R_%02i\n",
          ↪  i, left == 1 ? lhs : rhs, variable, variable);
96
97        // Since we now know the variable is 0, we can update all the products
          ↪  it is in to also be 0
98        for (int otherVariable = 0; otherVariable < vLen; ++otherVariable) {
99          if (otherVariable == variable) continue;
100
101         const int otherProduct = product_1d_index(vLen, variable,
            ↪  otherVariable);
102         products[otherProduct] = 0;
103
104         debug_printf("  Computed RP_%03i = R_%02i * R_%02i = 0 * a = 0\n",
            ↪  otherProduct, variable, otherVariable);
105       }
106     }
107   }
108 }
109
110 static void
111 select_values_from_products(Variable *v, int vLen, Variable *p, int pLen) {
112   for (int freeVariable = 0; freeVariable < vLen; ++freeVariable) {
113     const Variable solution = p[freeVariable];
114     if (solution < unknown) continue;
115
116     int l, r;
117     product_2d_index(vLen, freeVariable, &l, &r);
118     const Variable left = v[l], right = v[r];
119
120     debug_printf("RP_%03i is free, product of R_%02i and R_%02i (%i * %i)\n",
        ↪  freeVariable, l, r, left, right);
121
122     // If left and right are both unknown picking a value for the products,
        ↪  wont allow us to solve any other variables
123     // In this case wee just pick the value and advance to next free variable
```

```
124     if (left == unknown && right == unknown) {
125       debug_printf("Both variables unknown, skipping\n");
126       printfn;
127       continue;
128     }
129
130     // At this point the known variable is 1 (otherwise we would have solved
        ↪  it during simplification)
131     // This means that the value of the unknown will be the same as the value
        ↪  we choose for the product
132     const int unknownVariable = left == unknown ? l : r;
133
134     // Pick a value for the free variable, in
135     int solutions = 0, lastSolution = -1;
136     for (u08 value = 0; value <= 1; ++value) {
137       debug_printf("-->Checking value R_%2i = %i\n", unknownVariable, value);
138
139       // Check that assigning the free variable this value (and thus assigning
          ↪  it to the unknown as well)
140       // still satisfies all the products
141       int satisfiesAll = 1;
142       for (int other = 0; other < vLen; ++other) {
143         if (other == unknownVariable) continue;
144
145         const int otherProductIndex = product_1d_index(vLen, unknownVariable,
            ↪  other);
146         const Variable otherProduct = p[otherProductIndex];
147         if (otherProduct == unknown || otherProduct == free_variable) {
148           debug_printf("Skipped RP_%03i = R_%2i * R_%2i = %i\n",
              ↪  otherProductIndex, unknownVariable, other, otherProduct);
149           continue;
150         }
151
152         // The value now must match that of the product
153         const Variable chosenProduct = value * v[other];
154         if (chosenProduct != otherProduct) {
155           satisfiesAll = 0;
156           debug_printf("Value does not satisfy RP_%03i = R_%2i * R_%2i =
              ↪  %i\n", otherProductIndex, unknownVariable, other, otherProduct);
157         }
158         else {
159           debug_printf("Satifies RP_%03i = R_%2i * R_%2i = %i\n",
              ↪  otherProductIndex, unknownVariable, other, otherProduct);
160         }
161       }
162
163       if (satisfiesAll) {
```

```
164         debug_printf("Posible value RP_%3i = %i => R_%2i = %i\n",
            ↪  freeVariable, value, unknownVariable, value);
165
166         ++solutions;
167         lastSolution = value;
168       }
169
170       printfn;
171     }
172
173     // Check whether or not we would be able to choose one or both values
174     // If we can pick either, don't do anything
175     if(solutions == 1) {
176       debug_printf("Chose value RP_%3i = %i => R_%2i = %i\n", freeVariable,
          ↪  lastSolution, unknownVariable, lastSolution);
177       v[unknownVariable] = p[freeVariable] = lastSolution;
178
179       printfn;
180       simplify_products(v, vLen, p, pLen);
181     }
182     else if(solutions == 2) {
183       debug_printf("Both 0 and 1 would work as solutions for R2P_%3i, cannot
          ↪  choose either\n", freeVariable);
184     }
185   }
186 }
187
188 static inline void
189 save_matrix_to_file(char *fileName, Row *matrix, int height, int width) {
190   FILE *file = fopen(fileName, "w");
191   for (int r = 0; r < height; ++r) {
192     Row *row = matrix + r;
193     for (int c = 0; c < width; ++c) {
194       fprintf(file, "%c", row->columns[c] ? '1' : '-');
195     }
196     fprintf(file, "%c", row->value ? '1' : '-');
197
198     fprintf(file, "\n");
199   }
200
201   fflush(file);
202   fclose(file);
203 }
204
205 static int
206 solve(Row *rows, const int height, Registers *result) {
207   const int width = sizeof(Row);
```

```
208
209   // Gauss-Jordan elimination on the matrix to find solutions
210   int workingRow = 0;
211   Variable solutions[NUMBER_OF_UNKNOWNS];
212   for (int c = 0; c < NUMBER_OF_UNKNOWNS; ++c) {
213     solutions[c] = unknown;
214
215     Row *pivot = NULL;
216     for (int r = workingRow; r < height; ++r) {
217       if (rows[r].coefficients[c]) {
218         if (r != workingRow) {
219           row_swap(rows + r, rows + workingRow);
220         }
221
222         pivot = rows + workingRow;
223         break;
224       }
225     }
226
227     if (!pivot) {
228       solutions[c] = free_variable;
229
230       ++workingRow;
231       continue;
232     }
233
234     for (int r = 0; r < height; ++r) {
235       Row *target = rows + r;
236       if (!target->coefficients[c] || r == workingRow) continue;
237
238       row_xor(target, pivot);
239     }
240
241     ++workingRow;
242   }
243
244   // Go through the matrix, get the solutions
245   for (int r = 0; r < height; ++r) {
246     Row row = rows[r];
247
248     int firstColumn = -1;
249     for (int c = r; c < NUMBER_OF_UNKNOWNS; ++c) {
250       if (firstColumn == -1 && row.coefficients[c]) {
251         firstColumn = c;
252         solutions[c] = row.value;
253       }
254       else if (firstColumn != -1 && row.coefficients[c]) {
```

```
255         solutions[firstColumn] = unknown;
256         break;
257       }
258     }
259   }
260
261   // Check that everything appears valid, we check:
262   // 0. The constant must be solved to 1
263   if (solutions[MATRIX_CONSTANT_COLUMN] != 1) {
264     debug_printf("Constant does not solve to 1, invalid solution\n");
265     return 0;
266   }
267
268   // 1. Variables corresponding to the loaded bits should be free (this is a
      ↪ quirk of how we track the loaded bits)
269   const int freeLoadedBits = (solutions[15] == free_variable) &&
      ↪ (solutions[35] == free_variable) &&(solutions[59]);
270   if (!freeLoadedBits) {
271     debug_printf("Expecting all 3 loaded bits to solve to free variables,
      ↪ invalid solution\n");
272     return 0;
273   }
274
275   // 2. All other variables for the registers should be solved
276   int solvedVariables = 0;
277   for (int i = 0; i < VARIABLES_FROM_REGISTERS; ++i) {
278           if (solutions[i] < unknown) {
279       ++solvedVariables;
280           }
281   }
282   if(solvedVariables != VARIABLES_FROM_REGISTERS - 3) { // All bits, except
      ↪ the 3 loaded bits
283     debug_printf("Unable to obtain solutions for all variables corresponding
      ↪ to bits, invalid solution\n");
284     return 0;
285   }
286
287   // Update the solutions with the value of the loaded bits (which we know are
      ↪ 1)
288   solutions[15] = 1;
289   solutions[35] = 1;
290   solutions[59] = 1;
291
292   // 3. The products which we have solved should match with the variables we
      ↪ have solved
293   int solutionIndex = 64; // The 64th index (65th) solution is the first
      ↪ product from R1P
```

```
294   for (int i = 0; i < R1P_LEN; ++i, ++solutionIndex) {
295     if (solutions[solutionIndex] > 1) continue;
296
297     // At this point we know that we have solutions for all the variables, so
        ↪  the computed product
298     // will always be either 0 or 1
299     int lhs, rhs;
300     product_2d_index(R1_LEN, i, &lhs, &rhs);
301     const int actualProduct = solutions[lhs] * solutions[rhs];
302
303     if (actualProduct != solutions[solutionIndex]) {
304       debug_printf("Found mismatch in R1 products, invalid solution: R1P_%03i
          ↪  = R1_%02i * R1_%02i != %i\n", i, lhs, rhs,
          ↪  solutions[solutionIndex]);
305       return 0;
306     }
307   }
308   for (int i = 0; i < R2P_LEN; ++i, ++solutionIndex) {
309     if (solutions[solutionIndex] > 1) continue;
310
311     int lhs, rhs;
312     product_2d_index(R2_LEN, i, &lhs, &rhs);
313     const int actualProduct = solutions[R1_LEN + lhs] * solutions[R1_LEN +
        ↪  rhs];
314
315     if (actualProduct != solutions[solutionIndex]) {
316       debug_printf("Found mismatch in R2 products, invalid solution: R2P_%03i
          ↪  = R2_%02i * R2_%02i = %i * %i != %i\n", i, lhs, rhs,
          ↪  solutions[R1_LEN + lhs], solutions[R1_LEN + rhs],
          ↪  solutions[solutionIndex]);
317       return 0;
318     }
319   }
320   for (int i = 0; i < R3P_LEN; ++i, ++solutionIndex) {
321     if (solutions[solutionIndex] > 1) continue;
322
323     int lhs, rhs;
324     product_2d_index(R3_LEN, i, &lhs, &rhs);
325     const int actualProduct = solutions[R1_LEN + R2_LEN + lhs] *
        ↪  solutions[R1_LEN + R2_LEN + rhs];
326
327     if (actualProduct != solutions[solutionIndex]) {
328       debug_printf("Found mismatch in R3 products, invalid solution: R3P_%03i
          ↪  = R3_%02i * R3_%02i = %i * %i != %i\n", i, lhs, rhs,
          ↪  solutions[R1_LEN + R2_LEN + lhs], solutions[R1_LEN + R2_LEN + rhs],
          ↪  solutions[solutionIndex]);
329       return 0;
```

```
330       }
331     }
332
333     // Rebuild the registers from the solutions
334     result->r1 = 0;
335     result->r2 = 0;
336     result->r3 = 0;
337
338     solutionIndex = 0;
339     for (int i = 0; i < R1_LEN; ++solutionIndex, ++i) {
340       result->r1 |= solutions[solutionIndex] * 0x80000000;
341       result->r1 >>= 1;
342     }
343     for (int i = 0; i < R2_LEN; ++solutionIndex, ++i) {
344       result->r2 |= solutions[solutionIndex] * 0x80000000;
345       result->r2 >>= 1;
346     }
347     for (int i = 0; i < R3_LEN; ++solutionIndex, ++i) {
348       result->r3 |= solutions[solutionIndex] * 0x80000000;
349       result->r3 >>= 1;
350     }
351
352     result->r1 >>= 32 - R1_LEN - 1;
353     result->r2 >>= 32 - R2_LEN - 1;
354     result->r3 >>= 32 - R3_LEN - 1;
355
356     return 1;
357 }
```

C.10 state.h

```
#pragma once

#include "A5.h"
#include "A5_2.h"
#include "types.h"
#include "vector.h"
#include "stretchy_buffer.h"

typedef struct
{
  Vector *R1[R1_LEN];
  Vector *R2[R2_LEN];
  Vector *R3[R3_LEN];

  Vector **output;

  u32 R4;

  struct
  {
    u08 R1[R1_LEN];
    u08 R2[R2_LEN];
    u08 R3[R3_LEN];
    u08 R4[R4_LEN];
  } constants;
} State;

static void
state_free(State *state) {
  for (int i = 0; i < R1_LEN; ++i)
    free(state->R1[i]);
  for (int i = 0; i < R2_LEN; ++i)
    free(state->R2[i]);
  for (int i = 0; i < R3_LEN; ++i)
    free(state->R3[i]);

  const int length = sb_count(state->output);
  for (int i = 0; i < length; ++i)
    free(state->output[i]);

  sb_free(state->output);
}

static void
state_init(State *state, const u32 r4) {
```

```
46   vector_set_starting_state(state->R1, ARRAY_SIZE(state->R1), 0);
47   vector_set_starting_state(state->R2, ARRAY_SIZE(state->R2),
     ↪  ARRAY_SIZE(state->R1));
48   vector_set_starting_state(state->R3, ARRAY_SIZE(state->R3),
     ↪  ARRAY_SIZE(state->R1) + ARRAY_SIZE(state->R2));
49
50   state->output = NULL;
51   state->R4 = r4;
52 }
53
54 static void
55 state_start(State *state, State *initialState) {
56   vector_reset_state(state->R1, ARRAY_SIZE(state->R1), 0);
57   vector_reset_state(state->R2, ARRAY_SIZE(state->R2), ARRAY_SIZE(state->R1));
58   vector_reset_state(state->R3, ARRAY_SIZE(state->R3), ARRAY_SIZE(state->R1) +
     ↪  ARRAY_SIZE(state->R2));
59
60   // Set the constants
61   for (int i = 0; i < R1_LEN; ++i) {
62     state->constants.R1[i] = state->R1[i]->constant;
63   }
64   for (int i = 0; i < R2_LEN; ++i) {
65     state->constants.R2[i] = state->R2[i]->constant;
66   }
67   for (int i = 0; i < R3_LEN; ++i) {
68     state->constants.R3[i] = state->R3[i]->constant;
69   }
70
71   // XOR Constants from frame number
72   if (initialState) {
73     for (int i = 0; i < R1_LEN; ++i) {
74       state->R1[i]->constant ^= initialState->constants.R1[i];
75     }
76
77     for (int i = 0; i < R2_LEN; ++i) {
78       state->R2[i]->constant ^= initialState->constants.R2[i];
79     }
80
81     for (int i = 0; i < R3_LEN; ++i) {
82       state->R3[i]->constant ^= initialState->constants.R3[i];
83     }
84   }
85 }
86
87 static void
88 state_track_loaded_bit(State *state) {
89   vector_set(state->R1[15], 0, 1);
```

```
90   vector_set(state->R2[16], 0, 1);
91   vector_set(state->R3[18], 0, 1);
92 }
93
94 static void
95 state_track_frame_xor(State *state, u08 bit) {
96   state->R1[0]->constant ^= bit;
97   state->R2[0]->constant ^= bit;
98   state->R3[0]->constant ^= bit;
99 }
100
101 static void
102 state_track_output_majority(Vector *dest, Vector *product, Vector *a, Vector
    ↪  *b, Vector *c) {
103   vector_multiply(product, a, b);
104   vector_xor(dest, product);
105
106   vector_multiply(product, b, c);
107   vector_xor(dest, product);
108
109   vector_multiply(product, a, c);
110   vector_xor(dest, product);
111 }
112
113 static void
114 state_track_clock_R1(State *state) {
115   const u32 size = ARRAY_SIZE(state->R1);
116
117   Vector *first = vector_alloc(0);
118   vector_xor(first, state->R1[13]);
119   vector_xor(first, state->R1[16]);
120   vector_xor(first, state->R1[17]);
121   vector_xor(first, state->R1[18]);
122
123   for (u32 i = size - 1; i > 0; --i) {
124     vector_copy(state->R1[i], state->R1[i - 1]);
125   }
126
127   vector_copy(state->R1[0], first);
128   free(first);
129 }
130
131 static void
132 state_track_clock_R2(State *state) {
133   const u32 size = ARRAY_SIZE(state->R2);
134
135   Vector *first = vector_alloc(0);
```

```
136   vector_xor(first, state->R2[20]);
137   vector_xor(first, state->R2[21]);
138
139   for (u32 i = size - 1; i > 0; --i) {
140     vector_copy(state->R2[i], state->R2[i - 1]);
141   }
142
143   vector_copy(state->R2[0], first);
144   free(first);
145 }
146
147 static void
148 state_track_clock_R3(State *state) {
149   const u32 size = ARRAY_SIZE(state->R3);
150
151   Vector *first = vector_alloc(0);
152   vector_xor(first, state->R3[7]);
153   vector_xor(first, state->R3[20]);
154   vector_xor(first, state->R3[21]);
155   vector_xor(first, state->R3[22]);
156
157   for (u32 i = size - 1; i > 0; --i) {
158     vector_copy(state->R3[i], state->R3[i - 1]);
159   }
160
161   vector_copy(state->R3[0], first);
162   free(first);
163 }
164
165 static void
166 state_clock_r4(State *state) {
167   const u32 parity = _mm_popcnt_u32(state->R4 & __tap_masks.N[3]) & 1;
168   state->R4 <<= 1;
169   state->R4 &= __register_masks.N[3];
170   state->R4 |= parity;
171 }
172
173 static void
174 state_clock_all(State *state, int track) {
175   state_clock_r4(state);
176
177   if (track) {
178     state_track_clock_R1(state);
179     state_track_clock_R2(state);
180     state_track_clock_R3(state);
181   }
182 }
```

```
183
184 static void
185 state_clock(State *state) {
186   const u32 control = state->R4;
187   const u32 majority = u32_majority((control & R4TAP1) > 0, (control & R4TAP2)
      ↪  > 0, (control & R4TAP3) > 0);
188
189   u32_4x mask;
190   mask.V = _mm_set1_epi32(state->R4);
191   mask.V = _mm_and_si128(mask.V, __control_taps.V);
192   mask.V = _mm_cmpgt_epi32(mask.V, __zero.V);
193   mask.V = _mm_cmpeq_epi32(mask.V, majority ? __max.V : __zero.V);
194
195   // mask = R4
196   // mask &= R4 control taps
197   // mask = mask > 0
198   // mask = mask == majority
199
200   state_clock_r4(state);
201
202   if (!state) return;
203   if (mask.N[0]) state_track_clock_R1(state);
204   if (mask.N[1]) state_track_clock_R2(state);
205   if (mask.N[2]) state_track_clock_R3(state);
206 }
207
208 static void
209 state_output(State *state) {
210   // XOR the leaving bits from each register
211   Vector *output = vector_alloc(0);
212   Vector *product = vector_alloc(0);
213   vector_xor(output, state->R1[18]);
214   vector_xor(output, state->R2[21]);
215   vector_xor(output, state->R3[22]);
216
217   // Majority from R1
218   vector_invert(state->R1[14]);
219   state_track_output_majority(output, product, state->R1[15], state->R1[14],
      ↪  state->R1[12]);
220   vector_invert(state->R1[14]);
221
222   // Majority form R2
223   vector_invert(state->R2[16]);
224   state_track_output_majority(output, product, state->R2[16], state->R2[13],
      ↪  state->R2[9]);
225   vector_invert(state->R2[16]);
226
```

```
227   // Majority from R3
228   vector_invert(state->R3[13]);
229   state_track_output_majority(output, product, state->R3[18], state->R3[16],
      ↪  state->R3[13]);
230   vector_invert(state->R3[13]);
231
232   free(product);
233   sb_push(state->output, output);
234 }
235
236 static void
237 state_setup(const u32 r4, const u32 frame, State *state, State *firstState) {
238   state_init(state, r4);
239
240   // Load frame
241   for (u08 i = 0; i < 22; ++i) {
242     state_clock_all(state, 1);
243
244     const u08 value = frame >> i & 1u;
245
246     state->R4 ^= value;
247     state_track_frame_xor(state, value);
248   }
249
250   state_start(state, firstState);
251
252   // Loaded bit
253   state->R4 |= __loaded_mask.N[3];
254   state_track_loaded_bit(state);
255
256   for (u08 i = 0; i < 99; ++i)
257     state_clock(state);
258 }
259
260 static void
261 state_cipher(State *state) {
262   for (u08 i = 0; i < 114; ++i) {
263     state_clock(state);
264     state_output(state);
265   }
266
267   for (u08 i = 0; i < 114; ++i) {
268     state_clock(state);
269     state_output(state);
270   }
271 }
```

C.11 types.h

```c
#pragma once

#include <inttypes.h>

#define ARRAY_SIZE(arr) (sizeof((arr)) / sizeof((arr)[0]))

#define printfn printf("\n")

#ifdef DEBUG
#define debug_printf(f_, ...) printf("\033[34mDEBUG\033[0m: "); \
                        printf((f_), ##__VA_ARGS__)
#else
#define debug_printf(f_, ...) 0
#endif

typedef uint8_t  u08;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;
typedef u08 u8;

typedef int8_t  i08;
typedef int16_t i16;
typedef int32_t i32;
typedef int64_t i64;
typedef i08 i8;

typedef float  f32;
typedef double f64;

typedef f32 r32;
typedef f64 r64;

typedef union
{
  __m256i V;
  u08 N[32];
} u08_x32;

typedef union {
  __m128i V;
  u32 N[4];
  u08 B[16];
} u32_4x;

```

```
46 static u32_4x __max = { .N[0] = 0xFFFFFFFF, .N[1] = 0xFFFFFFFF, .N[2] =
   ↪  0xFFFFFFFF, .N[3] = 0xFFFFFFFF };
47 static u32_4x __one = { .N[0] = 1, .N[1] = 1, .N[2] = 1, .N[3] = 1 };
48 static u32_4x __zero = { .N[0] = 0, .N[1] = 0, .N[2] = 0, .N[3] = 0 };
49
50 static inline
51 __m128i _wrapped_popcnt_epi32(__m128i v) {
52 #ifdef AVX512
53   return _mm_popcnt_epi32(v);
54 #else
55   u32_4x r = {
56     .V = v
57   };
58
59   r.N[0] = _mm_popcnt_u32(r.N[0]);
60   r.N[1] = _mm_popcnt_u32(r.N[1]);
61   r.N[2] = _mm_popcnt_u32(r.N[2]);
62   r.N[3] = _mm_popcnt_u32(r.N[3]);
63
64   return r.V;
65 #endif
66 }
```

C.12 vector.h

```c
#pragma once

#include <string.h>
#include "types.h"

#define R1_LEN 19
#define R2_LEN 22
#define R3_LEN 23
#define R4_LEN 17

#define R1P_LEN 171
#define R2P_LEN 231
#define R3P_LEN 253

#pragma push(pack, 1)
typedef struct
{
  u08 constant;

  union
  {
    u08 data[R1_LEN + R2_LEN + R3_LEN + R1P_LEN + R2P_LEN + R3P_LEN];

    struct
    {
      u08 R1[R1_LEN];
      u08 R2[R2_LEN];
      u08 R3[R3_LEN];

      u08 R1P[R1P_LEN];
      u08 R2P[R2P_LEN];
      u08 R3P[R3P_LEN];
    };
  };
} Vector;
#pragma pop(pack)

static Vector *
vector_alloc(const u08 constant) {
  Vector *vector = (Vector*) calloc(sizeof(Vector), 1);
  vector->constant = constant;

  return vector;
}

```

```
46 static void
47 vector_invert(Vector *vector) {
48   vector->constant ^= 1;
49 }
50
51 static void
52 vector_xor(Vector *left, Vector *right) {
53   left->constant ^= right->constant;
54   for(u32 i = 0; i < ARRAY_SIZE(left->data); ++i) {
55     left->data[i] ^= right->data[i];
56   }
57 }
58
59 static u32
60 product_1d_index(const int size, int x, int y) {
61   if (x < y) {
62     x ^= y;
63     y ^= x;
64     x ^= y;
65   }
66
67   const int diagonalSize = y + 1;
68   const int areaBelowDiagonal = (diagonalSize * diagonalSize - diagonalSize) /
     ↪  2;
69   const int unusedEntries = areaBelowDiagonal + diagonalSize;
70   const int usedEntries = diagonalSize * size - unusedEntries;
71   return usedEntries - (size - x); // - the number of entries after the given
     ↪  x-index
72 }
73
74 static u16 __product_2d_map_r1[] = {
```

```
75   0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008, 0x0009,
     ↪  0x000A, 0x000B, 0x000C, 0x000D, 0x000E, 0x000F, 0x0010, 0x0011, 0x0012,
     ↪  0x0102, 0x0103, 0x0104, 0x0105, 0x0106, 0x0107, 0x0108, 0x0109, 0x010A,
     ↪  0x010B, 0x010C, 0x010D, 0x010E, 0x010F, 0x0110, 0x0111, 0x0112, 0x0203,
     ↪  0x0204, 0x0205, 0x0206, 0x0207, 0x0208, 0x0209, 0x020A, 0x020B, 0x020C,
     ↪  0x020D, 0x020E, 0x020F, 0x0210, 0x0211, 0x0212, 0x0304, 0x0305, 0x0306,
     ↪  0x0307, 0x0308, 0x0309, 0x030A, 0x030B, 0x030C, 0x030D, 0x030E, 0x030F,
     ↪  0x0310, 0x0311, 0x0312, 0x0405, 0x0406, 0x0407, 0x0408, 0x0409, 0x040A,
     ↪  0x040B, 0x040C, 0x040D, 0x040E, 0x040F, 0x0410, 0x0411, 0x0412, 0x0506,
     ↪  0x0507, 0x0508, 0x0509, 0x050A, 0x050B, 0x050C, 0x050D, 0x050E, 0x050F,
     ↪  0x0510, 0x0511, 0x0512, 0x0607, 0x0608, 0x0609, 0x060A, 0x060B, 0x060C,
     ↪  0x060D, 0x060E, 0x060F, 0x0610, 0x0611, 0x0612, 0x0708, 0x0709, 0x070A,
     ↪  0x070B, 0x070C, 0x070D, 0x070E, 0x070F, 0x0710, 0x0711, 0x0712, 0x0809,
     ↪  0x080A, 0x080B, 0x080C, 0x080D, 0x080E, 0x080F, 0x0810, 0x0811, 0x0812,
     ↪  0x090A, 0x090B, 0x090C, 0x090D, 0x090E, 0x090F, 0x0910, 0x0911, 0x0912,
     ↪  0x0A0B, 0x0A0C, 0x0A0D, 0x0A0E, 0x0A0F, 0x0A10, 0x0A11, 0x0A12, 0x0B0C,
     ↪  0x0B0D, 0x0B0E, 0x0B0F, 0x0B10, 0x0B11, 0x0B12, 0x0C0D, 0x0C0E, 0x0C0F,
     ↪  0x0C10, 0x0C11, 0x0C12, 0x0D0E, 0x0D0F, 0x0D10, 0x0D11, 0x0D12, 0x0E0F,
     ↪  0x0E10, 0x0E11, 0x0E12, 0x0F10, 0x0F11, 0x0F12, 0x1011, 0x1012, 0x1112
76 };
77
78 static u16 __product_2d_map_r2[] = {
79   0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008, 0x0009,
     ↪  0x000A, 0x000B, 0x000C, 0x000D, 0x000E, 0x000F, 0x0010, 0x0011, 0x0012,
     ↪  0x0013, 0x0014, 0x0015, 0x0102, 0x0103, 0x0104, 0x0105, 0x0106, 0x0107,
     ↪  0x0108, 0x0109, 0x010A, 0x010B, 0x010C, 0x010D, 0x010E, 0x010F, 0x0110,
     ↪  0x0111, 0x0112, 0x0113, 0x0114, 0x0115, 0x0203, 0x0204, 0x0205, 0x0206,
     ↪  0x0207, 0x0208, 0x0209, 0x020A, 0x020B, 0x020C, 0x020D, 0x020E, 0x020F,
     ↪  0x0210, 0x0211, 0x0212, 0x0213, 0x0214, 0x0215, 0x0304, 0x0305, 0x0306,
     ↪  0x0307, 0x0308, 0x0309, 0x030A, 0x030B, 0x030C, 0x030D, 0x030E, 0x030F,
     ↪  0x0310, 0x0311, 0x0312, 0x0313, 0x0314, 0x0315, 0x0405, 0x0406, 0x0407,
     ↪  0x0408, 0x0409, 0x040A, 0x040B, 0x040C, 0x040D, 0x040E, 0x040F, 0x0410,
     ↪  0x0411, 0x0412, 0x0413, 0x0414, 0x0415, 0x0506, 0x0507, 0x0508, 0x0509,
     ↪  0x050A, 0x050B, 0x050C, 0x050D, 0x050E, 0x050F, 0x0510, 0x0511, 0x0512,
     ↪  0x0513, 0x0514, 0x0515, 0x0607, 0x0608, 0x0609, 0x060A, 0x060B, 0x060C,
     ↪  0x060D, 0x060E, 0x060F, 0x0610, 0x0611, 0x0612, 0x0613, 0x0614, 0x0615,
     ↪  0x0708, 0x0709, 0x070A, 0x070B, 0x070C, 0x070D, 0x070E, 0x070F, 0x0710,
     ↪  0x0711, 0x0712, 0x0713, 0x0714, 0x0715, 0x0809, 0x080A, 0x080B, 0x080C,
     ↪  0x080D, 0x080E, 0x080F, 0x0810, 0x0811, 0x0812, 0x0813, 0x0814, 0x0815,
     ↪  0x090A, 0x090B, 0x090C, 0x090D, 0x090E, 0x090F, 0x0910, 0x0911, 0x0912,
     ↪  0x0913, 0x0914, 0x0915, 0x0A0B, 0x0A0C, 0x0A0D, 0x0A0E, 0x0A0F, 0x0A10,
     ↪  0x0A11, 0x0A12, 0x0A13, 0x0A14, 0x0A15, 0x0B0C, 0x0B0D, 0x0B0E, 0x0B0F,
     ↪  0x0B10, 0x0B11, 0x0B12, 0x0B13, 0x0B14, 0x0B15, 0x0C0D, 0x0C0E, 0x0C0F,
     ↪  0x0C10, 0x0C11, 0x0C12, 0x0C13, 0x0C14, 0x0C15, 0x0D0E, 0x0D0F, 0x0D10,
     ↪  0x0D11, 0x0D12, 0x0D13, 0x0D14, 0x0D15, 0x0E0F, 0x0E10, 0x0E11, 0x0E12,
     ↪  0x0E13, 0x0E14, 0x0E15, 0x0F10, 0x0F11, 0x0F12, 0x0F13, 0x0F14, 0x0F15,
     ↪  0x1011, 0x1012, 0x1013, 0x1014, 0x1015, 0x1112, 0x1113, 0x1114, 0x1115,
     ↪  0x1213, 0x1214, 0x1215, 0x1314, 0x1315, 0x1415
```

```
};

static u16 __product_2d_map_r3[] = {
  0x0001, 0x0002, 0x0003, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008, 0x0009,
    ↪ 0x000A, 0x000B, 0x000C, 0x000D, 0x000E, 0x000F, 0x0010, 0x0011, 0x0012,
    ↪ 0x0013, 0x0014, 0x0015, 0x0016, 0x0102, 0x0103, 0x0104, 0x0105, 0x0106,
    ↪ 0x0107, 0x0108, 0x0109, 0x010A, 0x010B, 0x010C, 0x010D, 0x010E, 0x010F,
    ↪ 0x0110, 0x0111, 0x0112, 0x0113, 0x0114, 0x0115, 0x0116, 0x0203, 0x0204,
    ↪ 0x0205, 0x0206, 0x0207, 0x0208, 0x0209, 0x020A, 0x020B, 0x020C, 0x020D,
    ↪ 0x020E, 0x020F, 0x0210, 0x0211, 0x0212, 0x0213, 0x0214, 0x0215, 0x0216,
    ↪ 0x0304, 0x0305, 0x0306, 0x0307, 0x0308, 0x0309, 0x030A, 0x030B, 0x030C,
    ↪ 0x030D, 0x030E, 0x030F, 0x0310, 0x0311, 0x0312, 0x0313, 0x0314, 0x0315,
    ↪ 0x0316, 0x0405, 0x0406, 0x0407, 0x0408, 0x0409, 0x040A, 0x040B, 0x040C,
    ↪ 0x040D, 0x040E, 0x040F, 0x0410, 0x0411, 0x0412, 0x0413, 0x0414, 0x0415,
    ↪ 0x0416, 0x0506, 0x0507, 0x0508, 0x0509, 0x050A, 0x050B, 0x050C, 0x050D,
    ↪ 0x050E, 0x050F, 0x0510, 0x0511, 0x0512, 0x0513, 0x0514, 0x0515, 0x0516,
    ↪ 0x0607, 0x0608, 0x0609, 0x060A, 0x060B, 0x060C, 0x060D, 0x060E, 0x060F,
    ↪ 0x0610, 0x0611, 0x0612, 0x0613, 0x0614, 0x0615, 0x0616, 0x0708, 0x0709,
    ↪ 0x070A, 0x070B, 0x070C, 0x070D, 0x070E, 0x070F, 0x0710, 0x0711, 0x0712,
    ↪ 0x0713, 0x0714, 0x0715, 0x0716, 0x0809, 0x080A, 0x080B, 0x080C, 0x080D,
    ↪ 0x080E, 0x080F, 0x0810, 0x0811, 0x0812, 0x0813, 0x0814, 0x0815, 0x0816,
    ↪ 0x090A, 0x090B, 0x090C, 0x090D, 0x090E, 0x090F, 0x0910, 0x0911, 0x0912,
    ↪ 0x0913, 0x0914, 0x0915, 0x0916, 0x0A0B, 0x0A0C, 0x0A0D, 0x0A0E, 0x0A0F,
    ↪ 0x0A10, 0x0A11, 0x0A12, 0x0A13, 0x0A14, 0x0A15, 0x0A16, 0x0B0C, 0x0B0D,
    ↪ 0x0B0E, 0x0B0F, 0x0B10, 0x0B11, 0x0B12, 0x0B13, 0x0B14, 0x0B15, 0x0B16,
    ↪ 0x0C0D, 0x0C0E, 0x0C0F, 0x0C10, 0x0C11, 0x0C12, 0x0C13, 0x0C14, 0x0C15,
    ↪ 0x0C16, 0x0D0E, 0x0D0F, 0x0D10, 0x0D11, 0x0D12, 0x0D13, 0x0D14, 0x0D15,
    ↪ 0x0D16, 0x0E0F, 0x0E10, 0x0E11, 0x0E12, 0x0E13, 0x0E14, 0x0E15, 0x0E16,
    ↪ 0x0F10, 0x0F11, 0x0F12, 0x0F13, 0x0F14, 0x0F15, 0x0F16, 0x1011, 0x1012,
    ↪ 0x1013, 0x1014, 0x1015, 0x1016, 0x1112, 0x1113, 0x1114, 0x1115, 0x1116,
    ↪ 0x1213, 0x1214, 0x1215, 0x1216, 0x1314, 0x1315, 0x1316, 0x1415, 0x1416,
    ↪ 0x1516
};

static void
product_2d_index(const int size, int idx, int* x, int *y) {
  u16 value;
  switch (size) {
  case R1_LEN:
    value = __product_2d_map_r1[idx];
    break;
  case R2_LEN:
    value = __product_2d_map_r2[idx];
    break;
  case R3_LEN:
    value = __product_2d_map_r3[idx];
    break;
```

```
  default:
    value = 0;
  }

  *x = value >> 8;
  *y = value & 0xFF;
}

static void
vector_multiply(Vector *dest, Vector *left, Vector *right) {
  // Constants
  dest->constant = left->constant * right->constant;
  for (u32 i = 0; i < ARRAY_SIZE(left->data); ++i) {
    const u08 lp = left->constant * right->data[i];
    const u08 rp = right->constant * left->data[i];

    dest->data[i] = lp ^ rp;
  }

  // Multiply R1
  u32 size = ARRAY_SIZE(dest->R1);
  for (u32 i = 0; i < size; ++i)
    for (u32 j = 0; j < size; ++j) {
      if (!(left->R1[i] * right->R1[j])) continue;

      if(i == j) {
        dest->R1[i] ^= 1;
        continue;
      }

      dest->R1P[product_1d_index(size, i, j)] ^= 1;
    }

  // Multiply R2
  size = ARRAY_SIZE(dest->R2);
  for (u32 i = 0; i < size; ++i)
    for (u32 j = 0; j < size; ++j) {
      if (!(left->R2[i] * right->R2[j])) continue;

      if (i == j) {
        dest->R2[i] ^= 1;
        continue;
      }

      dest->R2P[product_1d_index(size, i, j)] ^= 1;
    }

```

```
146   // Multiply R3
147   size = ARRAY_SIZE(dest->R3);
148   for (u32 i = 0; i < size; ++i)
149     for (u32 j = 0; j < size; ++j) {
150       if (!(left->R3[i] * right->R3[j])) continue;
151
152       if (i == j) {
153         dest->R3[i] ^= 1;
154         continue;
155       }
156
157       dest->R3P[product_1d_index(size, i, j)] ^= 1;
158     }
159 }
160
161 static void
162 vector_copy(Vector *left, Vector *right) {
163   left->constant = right->constant;
164   memcpy(left->data, right->data, ARRAY_SIZE(left->data));
165 }
166
167 static Vector*
168 vector_clone(Vector *vector) {
169   Vector *result = vector_alloc(vector->constant);
170   vector_copy(result, vector);
171
172   return result;
173 }
174
175 static void
176 vector_set_starting_state(Vector **state, const u32 size, const u32 offset) {
177   for (u32 i = 0; i < size; ++i) {
178     state[i] = vector_alloc(0);
179     state[i]->data[offset + i] = 1;
180   }
181 }
182
183 static void
184 vector_set(Vector *vector, const u08 value, const u08 constant) {
185   memset(vector->data, value, ARRAY_SIZE(vector->data));
186   vector->constant = constant;
187 }
188
189 static void
190 vector_set_coefficients(Vector *vector, const u08 value) {
191   memset(vector->data, value, ARRAY_SIZE(vector->data));
192 }
```

```
193
194 static void
195 vector_reset_state(Vector **state, const u32 size, const u32 offset) {
196   for (u32 i = 0; i < size; ++i) {
197     vector_set_coefficients(state[i], 0);
198     state[i]->data[offset + i] = 1;
199   }
200 }
```

C.13 test_a52.c

```c
#include <stdio.h>
#include <string.h>

#include "A5_2.h"

int
main(int argc, char **argv) {
  // Known good key for testing
  u64 key = 0xFFFFFFFFFFFFFC00; // note: Endianness
  u32 frame = 0x21;
  A5Context context;

  u08 incoming[INCOMING_KEYSTREAM_BYTES], outgoing[OUTGOING_KEYSTREAM_BYTES];
  memset(incoming, 0, INCOMING_KEYSTREAM_BYTES);
  memset(outgoing, 0, OUTGOING_KEYSTREAM_BYTES);

  // Sets the number of times to generate the keystream (for performance
  ↪  testing)
  #define COUNT 1

  float sum = 0;
  for(int i = 0; i < COUNT; ++i) {
    u64 start = __rdtsc();
    // Run the key setup
    A52_setup(&context, key, frame);
    // Run the 327 cycles after the key setup, this sets the incoming/outgoing
    ↪  to the respective 114 bits/15 bytes
    A52(&context, incoming, outgoing);
    u64 end = __rdtsc();

    sum += (end - start);
  }

  float avg = sum / COUNT;
  printf("Average cycles: %f.2\n", avg);

  // Known good test keystreams
  u08 good_incoming[INCOMING_KEYSTREAM_BYTES] = { 0xF4, 0x51, 0x2C, 0xAC,
  ↪  0x13, 0x59, 0x37, 0x64, 0x46, 0x0B, 0x72, 0x2D, 0xAD, 0xD5, 0x00 };
  u08 good_outgoing[OUTGOING_KEYSTREAM_BYTES] = { 0x48, 0x00, 0xD4, 0x32,
  ↪  0x8E, 0x16, 0xA1, 0x4D, 0xCD, 0x7B, 0x97, 0x22, 0x26, 0x51, 0x00 };

  printf("Output for frame %u with key %16lX:\n", frame, key);

  printf("Expected incoming: ");
```

```
42  for(int i = 0; i < INCOMING_KEYSTREAM_BYTES; ++i)
43    printf("%2X ", good_incoming[i]);
44  printf("\n");
45
46  printf("Actual: ");
47  for(int i = 0; i < INCOMING_KEYSTREAM_BYTES; ++i)
48    printf("%2X ", incoming[i]);
49  printf("\n");
50
51  printf("Expected outgoing: ");
52  for(int i = 0; i < OUTGOING_KEYSTREAM_BYTES; ++i)
53    printf("%2X ", good_outgoing[i]);
54  printf("\n");
55
56  printf("Actual: ");
57  for(int i = 0; i < OUTGOING_KEYSTREAM_BYTES; ++i)
58    printf("%2X ", outgoing[i]);
59  printf("\n");
60
61  // Checks that our generated keystreams match the expected
62  u08 check_incoming = 1, check_outgoing = 1;
63  for(int i = 0; i < INCOMING_KEYSTREAM_BYTES; ++i) {
64    if(incoming[i] != good_incoming[i])
65      check_incoming = 0;
66
67    if(outgoing[i] != good_outgoing[i])
68      check_outgoing = 0;
69  }
70
71  printf("\n");
72  printf("Test keystream Network -> Phone: %i, Test keystream Phone ->
    ↪  Network: %i\n", check_incoming, check_outgoing);
73  if(check_incoming && check_outgoing) {
74    printf("Both tests passed\n");
75  } else {
76    printf("Test(s) failed\n");
77  }
78
79  return 0;
80 }
```