Refined Cryptanalysis of the GPRS Ciphers GEA-1 and GEA-2

Dor Amzaleg and Itai Dinur

Department of Computer Science, Ben-Gurion University, Israel

Abstract. At EUROCRYPT 2021, Beierle et al. presented the first public analysis of the GPRS ciphers GEA-1 and GEA-2. They showed that although GEA-1 uses a 64-bit session key, it can be recovered with the knowledge of only 65 bits of keystream in time 2^{40} using 44 GiB of memory. The attack exploits a weakness in the initialization process of the cipher that was presumably hidden intentionally by the designers to reduce its security.

While no such weakness was found for GEA-2, the authors presented an attack on this cipher with time complexity of about 2^{45}. The main practical obstacle is the required knowledge of 12800 bits of keystream used to encrypt a full GPRS frame. Variants of the attack are applicable (but more expensive) when given less consecutive keystream bits, or when the available keystream is fragmented (it contains no long consecutive block).

In this paper, we improve and complement the previous analysis of GEA-1 and GEA-2. For GEA-1, we devise an attack in which the memory complexity is reduced by a factor of about $2^{13} = 8192$ from 44 GiB to about 4 MiB, while the time complexity remains 2^{40}. Our implementation recovers the GEA-1 session key in average time of 2.5 hours on a modern laptop.

For GEA-2, we describe two attacks that complement the analysis of Beierle et al. The first attack obtains a linear tradeoff between the number of consecutive keystream bits available to the attacker (denoted by ℓ) and the time complexity. It improves upon the previous attack in the range of (roughly) $\ell \leq 7000$. Specifically, for $\ell = 1100$ the complexity of our attack is about 2^{54}, while the previous one is not faster than the 2^{64} brute force complexity. In case the available keystream is fragmented, our second attack reduces the memory complexity of the previous attack by a factor of 512 from 32 GiB to 64 MiB with no time complexity penalty.

Our attacks are based on new combinations of stream cipher cryptanalytic techniques and algorithmic techniques used in other contexts (such as solving the k-XOR problem).

1 Introduction

GPRS (General Packet Radio Service) is a mobile data standard that was widely deployed in the early 2000s. The standard is based on the GSM (2G) technology established by the European Telecommunications Standards Institute (ETSI). Encryption is used to protect against eavesdropping between the phone and the base station, and two proprietary stream ciphers GEA-1 and GEA-2 were initially designed and used for this purpose.

1.1 First Public Analysis of GEA-1 and GEA-2

Recently, Beierle et al. presented the first public analysis of GEA-1 and GEA-2, which should ideally provide 64-bit security [2]. Remarkably, the authors described a weakness in the initialization process of GEA-1, showing that two of its three internal linear feedback shift registers (LFSRs) can only assume 2^{40} values out of the 2^{64} possible. This led to a practical meet-in-the-middle (MITM) attack in time complexity 2^{40} and memory complexity 44.5 GiB. The attack only needs 65 bits of known keystream (24 from the same frame), which can be easily deduced from the ciphertext assuming knowledge of 65 plaintext bits (that can be obtained from metadata such as headers). The attack is therefore completely practical, as demonstrated by the authors. Since the attack recovers the 64-bit session key, it allows to decrypt the entire GPRS session.

The weakness of GEA-1 is believed to have been intentionally introduced and hidden by the designers, presumably due to strict export regulations on cryptography that were in effect in 1998 when the cipher was designed. To support this hypothesis, [2] carried out extensive experiments on random LFSRs which showed that it is very unlikely that the weakness occurred by chance. In the followup work [3], Beierle, Felke and Leander showed how to construct such a weak cipher efficiently.

The initialization weakness of GEA-1 is not present in the stronger cipher GEA-2 (which also uses a fourth register to produce the output). Yet, the authors of [2] presented an attack on GEA-2 which showed that it does not provide the ideal 64-bit security. Specifically, given 12800 bits of keystream used to encrypt a full GPRS frame, the complexity of the attack is about 2^{45} GEA-2 evaluations and it requires 32 GiB of memory. It is based on a combination of algebraic and MITM attacks.

The main challenge in practice is in obtaining the 12800-bit consecutive keystream, which may require social engineering or additional ad-hoc methods. Therefore, the authors presented a data-time tradeoff curve showing that the crossover point for beating exhaustive search is about 1468 consecutive keystream bits.

A variant of the attack is also applicable in case the known available keystream used to encrypt a frame is fragmented and contains no long consecutive block. In particular, given 11300 bits of fragmented keystream, the time complexity of the attack becomes roughly 2^{55}, while the memory complexity remains 32 GiB.

Impact of Attacks. The ETSI prohibited the implementation of GEA-1 in mobile phones in 2013. On the other hand, it is still mandatory to implement GEA-2 today [11].

Surprisingly, the authors of [2] noticed that modern mobile phones still supported GEA-1, deviating from the specification. As described in [2], this could have severe implications as it opens the door for various types of downgrade attacks. Consequently, after disclosing this vulnerability, test cases were added to verify that the support of GEA-1 is disabled by devices before entering the market.

In contrast, ETSI followed the mid-term goal to remove the support of GEA-2 from the specification. Yet, specification changes require consent of several parties and may take a long time.

1.2 Our Results

In this paper, we describe several attacks on GEA-1 and GEA-2 that improve and complement the ones of [2]. Our attacks are summarized in Table 1.

Attack G1. Attack G1 reduces the memory complexity of the previous attack on GEA-1 by a factor of about $2^{13} = 8192$ to 4 MiB, while the time complexity remains 2^{40} GEA-1 evaluations.[1] We implemented the attack and executed it on a modern laptop. Averaged over 5 runs, it recovers the GEA-1 session key in average time of 2.5 hours. In comparison, as it is difficult to run the attack of [2] on a laptop due to its high memory consumption, it was executed on a cluster.

For GEA-2, we present two attacks that focus on scenarios where the attacker obtains limited data which may be easier to acquire in practice. In general, the feasibility of assumptions on the available data depend on the exact attack scenario, and our goal is to describe attacks that optimally utilize this data.

Attack G2-1. Attack G2-1 assumes the attacker obtains ℓ bits of consecutive keystream. The complexity of this attack is about $2^{64}/(\ell - 62)$ GEA-2 evaluations. For example, given $\ell = 126$ (a keystream of moderate length), it already has a non-negligible advantage by a factor of 64 over exhaustive search. For $\ell = 1100$ the complexity of our attack is roughly 2^{54}, while the previous attack is not faster than the 2^{64} brute force complexity. In the range $\ell > 7000$, the attack of [2] is more efficient. Our attacks consume a moderately larger amount of memory than those of [2] (by a factor between 2 and 5, depending on the variant).

Attack G2-2. Attack G2-2 is mostly interesting when the available keystream is fragmented. This may occur if (for example) the eavesdropping communication channel is noisy or not stable, or the attacker only knows parts of the plaintext. In this scenario, our attack reduces the memory complexity of the previous attack by a factor of *at least* $2^9 = 512$ from 2^{35} bytes (32 GiB) to at most 2^{26} bytes (64 MiB) with no penalty in time complexity. For example, given 11300 bits of fragmented keystream in a frame, the complexity of the previous attack is about

[1] As in [2], we define a GEA-1 evaluation as the number of bit operations required to generate a 128-bit keystream.

2^{55} and it requires about 32 GiB of memory. We reduce the memory complexity to 32 MiB (by a factor of 2^{10}). Since the cost of the attack is largely influenced by its memory complexity, such a reduction is clearly favorable.

Cipher	Attack	Time	Data (bits)	Memory	Main technique	Section
GEA-1	G1	2^{40}	65	4 MiB	3-XOR	3.4
GEA-2	G2-1	$2^{64}/(\ell - 62)$	ℓ consecutive	64 GiB	4-XOR	4.3
GEA-2	G2-2[†]	2^{55}	11320 fragmented	32 MiB	Algebraic + MITM	4.4

[†] Specific parameter set for the attack with 11320 bits of fragmented keystream.

Table 1. Summery of our attacks.

Impact of new attacks. Unlike the work of [2], our work does not have immediate practical implications. Supposedly, after the measures taken following the work of [2], GEA-1 should no longer be supported by modern mobile phones. Regardless, the attack of [2] on GEA-1 is already practical and there is little more to be gained on this front.

On the other hand, our memory-optimized attack on GEA-1 is still interesting since it shows that the cost of eavesdropping to communication at a large scale (i.e., simultaneously eavesdropping to several GPRS sessions) is even lower than predicted by [2]. Indeed, implementing such an attack that requires several dozens of GiB was not trivial in the early 2000's, when GEA-1 was in wide use. With significantly reduced memory consumption, it is much easier to distribute the attack's workload among many cheap low-end devices.

As for GEA-2, our attacks provide new and interesting scenarios in which the cipher can be broken more efficiently than before. These attacks may have longer-term impact in expediting the removal of GEA-2 from the specification.

Regardless of this work's practical impact, we view its main contribution as technical and summarize it below. Analyzing ciphers that have been in wide use provides additional motivation for this work, yet it is not the only motivation.

1.3 Technical Contributions

GEA-1 and GEA-2 have interesting designs and there is additional insight to be gained from their analysis. Our techniques build on work that was published well after GEA-1 and GEA-2 were designed. However, this does not rule out the possibility that (variants of) these techniques were used (e.g., by intelligence agencies) to break the ciphers in practice. We now overview some of our techniques.

Optimization and adaptation of *k*-XOR algorithms. In the k-XOR problem, we are given access to k random functions $f_1, \ldots, f_k$ and a target value t, and the goal is to find a k-tuple of inputs $(x^{(1)}, \ldots, x^{(k)})$ such that $f_1(x^{(1)}) \oplus \ldots \oplus f_k(x^{(k)}) = t$. Since the outputs of GEA-1 and GEA-2 are calculated by XOR-ing the outputs of their internal registers, using techniques for solving k-XOR in their cryptanalysis is natural (indeed, the MITM attacks of [2] essentially solve a 2-XOR problem). However, in our specific case, we wish to apply additional techniques which are not directly applicable. Consequently, we optimize and adapt them to obtain our attacks.

Attack G1. In cryptanalysis of GEA-1, we use the *clamping through precomputation* technique, proposed to reduce the memory complexity of k-XOR algorithms in [4] by Bernstein. Applying the technique naively results in a penalty in time complexity. Our main observation is that the $k = 3$ functions in the corresponding 3-XOR problem are not random, and we show how to exploit a property of the GEA-1 internal registers to apply the technique with no penalty. Essentially, the property is that it is possible to efficiently enumerate all internal states of a register that output a given prefix string.[2]

Attack G2-1. In Attack G2-1, we attempt to apply Wagner's *k-tree algorithm* [19]. For $k = 4$ it improves upon standard 4-XOR algorithms provided that the domains of f_1, f_2, f_3, f_4 are sufficiently large and many 4-XOR solutions exist. The algorithm exploits this to efficiently find only one of them. However, the k-tree algorithm is not directly applicable to GEA-2, as a standard attack based on 4-XOR can only target a single internal state of GEA-2. Nevertheless, we show how to adapt a technique developed in [2] (and used in another attack) which allows to simultaneously target several internal states of the stream cipher. In our case, this artificially creates more solutions to the 4-XOR problem, and therefore a variant of the k-tree algorithm is applicable.

Application to the stream cipher XOR combiner. Interestingly, unlike the other attacks on GEA-2 (including the ones of [2], which exploit the low algebraic degree of its output), Attack G2-1 does not assume any special property of the 4 internal GEA-2 registers, whose outputs are XORed to produce the keystream. The attack is therefore applicable to a generic XOR combiner of 4 stream ciphers with an arbitrary internal structure. It shows that in order to provide ideal 64-bit security, the internal state size of GEA-2 must be increased, and changing the feedback and output functions of its registers does not suffice (assuming it remains a XOR combiner).

Optimizing meet-in-the-middle attacks by subspace decompositions. A MITM attack is composed of two parts, each iterating over a subspace of

[2] This property is somewhat related to the *sampling resistance* property defined in the context of time-memory tradeoffs for stream ciphers with precomputation [5,6]. However, sampling resistance deals with the complexity of efficiently generating a *single* state (specified by some index) that produces an output prefix. On the other hand, we need to efficiently generate *all* states with a different efficiency measure.

vectors. If the vectors of the two subspaces are linearly dependent, we can decompose them and iterate over their common dependent part in a loop. Each iteration consists of a MITM attack on smaller independent subspaces, reducing the memory complexity. This technique is relatively standard (see [1,7,14]), although typically applied in different settings such as hash function cryptanalysis.

Our attacks use subspace decompositions several times. In a few of these cases, they are not initially applicable and only made possible in combination with additional techniques. Specifically, for GEA-1 we use two decompositions and the second one is made possible by exploiting specific properties of its internal registers. Attack G2-2 is based on the combined algebraic and MITM (or 2-XOR) attack of [2]. Subspace decomposition is made possible after guessing the values of carefully chosen linear combinations of variables. Interestingly, we benefit from this guessing strategy *twice*: once for its original purpose in [2] of reducing the number of variables in an algebraic linearization attack, and a second time in reducing the memory complexity of the subsequent MITM attack.

As we show, this guessing strategy also optimizes additional attacks on GEA-2 which previously seemed very inefficient, and makes them competitive. Therefore, an interesting open problem is to further optimize them and improve the state-of-the-art.

1.4 Structure of the Paper

The rest of this paper is structured as follows. Next, in Section 2, we give some preliminaries. Our attack on GEA-1 is described in Section 3, while our attacks on GEA-2 are given in Section 4.

2 Preliminaries

2.1 Description of GEA-1 and GEA-2

We give a short description of the GPRS ciphers GEA-1 and GEA-2, as specified in [2] (which is currently the only public source for their specification). We only describe the relevant components for our analysis.

The input to the encryption process of both ciphers consists of a 12800-bit plaintext (GPRS frame), a 64-bit session key, a direction bit (uplink/downlink), and a 32-bit IV which is a counter incremented for each frame.

GEA-1. GEA-1 uses three linear feedback shift registers (LFSRs) over $\mathbb{F}_2$, named A, B and C of lengths 31, 32 and 33, respectively. The registers operate in Galois mode, namely the bit that is shifted out of a register is XORed to the bits in a specified set of positions. The output of each register is computed by a non-linear Boolean function $f : \mathbb{F}_2^7 \rightarrow \mathbb{F}_2$ which has an algebraic degree of 4 (see [2] for its specification).

Initialization. The inputs to the GEA-1 initialization process consist of a 64-bit secret key, a public direction bit, and a 32-bit public IV. The initialization uses a non-linear feedback shift register (NLFSR) of length 64 to which the inputs are loaded while clocking the register (refer to [2] for more details).

The NLFSR's final state is a 64-bit seed. The seed is used to initialize the registers A, B and C via a linear mapping. The exact details of this mapping are irrelevant to this paper. However, the weakness of GEA-1 is based on a crucial property of this mapping (discovered in [2]): the joint 64-bit initial state of the registers A and C can only attain 2^{40} values (out of the 2^{64} possible).

We further note that in the event that one of the registers is set to 0 after initialization, it is reset to a non-zero state. For simplicity, throughout this paper, we will ignore this unlikely event.

Finally, another property of the initialization that we will use (shown in [2]), is that given a 96-bit initial state of the registers and the public IV and direction bits, there is a very simple algorithm that inverts the initialization process and recovers the session key. This implies that recovering the 96-bit initial state in the encryption process of a single plaintext (frame) allows to decrypt the entire session.

Keystream generation. After initialization, the cipher starts generating keystream. The output of each register is calculated by applying f to 7 bits in specified positions. A keystream bit is computed by XORing the 3 register outputs. After calculating a keystream bit, each register is clocked once before producing the next keystream bit.

The feedback positions of each register and the positions which serve as inputs to f are given in Figure 1 (taken from [2]).

GEA-2. GEA-2 is built similarly to GEA-1, hence we focus on the differences. Besides the registers A, B, C, the GEA-2 state consists of a fourth 29-bit register D (which also uses f to produce the output), as shown in Figure 1. The GEA-2 keystream is generated by XORing the outputs of the 4 registers.

The initialization process of GEA-2 is similar to that of GEA-1, but it makes use of a longer 97-bit NLFSR which produces a 97-bit seed. The seed is then used to initialize the state of the 4 registers via a linear mapping. Unlike the initialization mapping of GEA-1, the mapping of GEA-2 does not seem to have any noticeable weakness (in particular, one can verify that any pair of registers can assume all possible states). As for GEA-1, given an initial state and the public inputs, it is possible to efficiently recover the session key.

2.2 Notation

We describe the notation used throughout this paper.

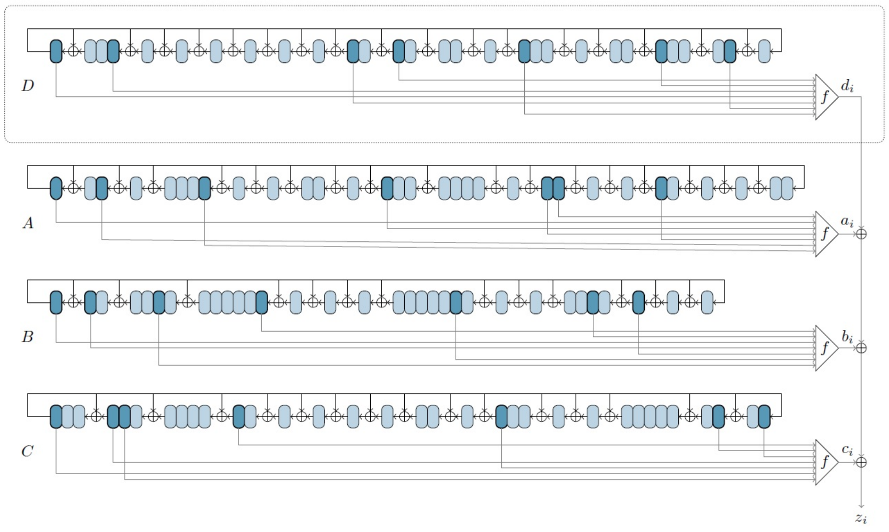

Fig. 1. Keystream generation of GEA-1 and GEA-2. Register D is only present in GEA-2. Credit: [2].

For an integer $n > 0$, let $[n] = \{1, \dots, n\}$. For a vector $x \in \mathbb{F}_2^n$ and $m \in [n]$, $x_{[m]} \in \mathbb{F}_2^m$ denotes the vector composed of the first m bits of x. For $m_1, m_2 \in [n]$ such that $m_1 \leq m_2$, $x_{[m_1,m_2]} \in \mathbb{F}_2^{m_2-m_1+1}$ denotes the vector composed of the bits of x in position m_1 up to m_2 (inclusive).

For a linear transformation T, we denote by $\mathrm{Im}(T)$ its image and by $\ker(T)$ its kernel. For a linear subspace V, we denote by $\dim(V)$ its dimension.

GEA-related notation. For the register A, we denote by $\hat{A} \in \mathbb{F}_2^{31}$ its internal state, and by $f_A : \mathbb{F}_2^{31} \to \{0,1\}^*$ the output of A starting from the given internal state. Typically, we will refer to specific bits of this function. In particular, for $m \in \mathbb{N}$, $f_A(\hat{A})_{[m]} \in \mathbb{F}_2^m$ denotes the first m output bits. Analogous notation is defined for the remaining registers B, C, D.

For $v \in \mathbb{F}_2^{96}$ (which represents an internal state of GEA-1), denote by $v_{[B]} \in \mathbb{F}_2^{32}$ its projection on the register B and by $v_{[AC]} \in \mathbb{F}_2^{64}$ its projection on the registers A and C. We use similar notations for the other GEA-1 registers and for GEA-2.

2.3 Computation Model and Data Structures

Consistently with [2], the complexity of the attacks on GEA-1 and GEA-2 is measured in terms of the number of operations required to generate a keystream of 128 bits.

The algorithms we describe use various lookup tables that support the operations of inserting and searching for elements. We assume that each such operation takes unit time (which is a standard assumption when using hash tables). This complexity of lookup table operations will typically be ignored in the total time complexity calculation, as for most attacks, it is proportional to the number of basic operations of evaluating the outputs of GEA registers.[3] We note that [2] used a slightly different computational model, but it does not have a significant impact on the final complexity estimations in our case.

2.4 3-XOR Problem

We define a variant of the well-known 3-XOR problem that is relevant for this paper. For simplicity, we assume the parameter n is divisible by 3.

Definition 1 (3-XOR). *Given access to 3 random functions $f_1, f_2, f_3 : \mathbb{F}_2^{n/3} \to \mathbb{F}_2^n$ and a target $t \in \mathbb{F}_2^n$, find $(x^{(1)}, x^{(2)}, x^{(3)}) \in (\mathbb{F}_2^{n/3})^3$ such that $f_1(x^{(1)}) \oplus f_2(x^{(2)}) \oplus f_3(x^{(3)}) = t$.*

We note that in a random function the output of every input is chosen uniformly at random from the range, independently of the other inputs. Since the 3-XOR problem places an n-bit condition on each triplet $(x^{(1)}, x^{(2)}, x^{(3)})$, the average number of solutions is $2^{3 \cdot n/3} \cdot 2^{-n} = 1$.

The naive 3-XOR algorithm based on sort-and-match (or meet-in-the-middle) has time complexity of roughly $2^{2n/3}$. It is a major open problem to improve this complexity significantly.[4] The naive 3-XOR algorithm also requires $2^{n/3}$ words of memory (of length $O(n)$ bits). However, unlike time complexity, we can significantly improve the memory complexity.

Proposition 1 (3-XOR algorithm using enumeration). *Let $\tau \in \{0, \ldots, n/3\}$ be a parameter. Assume there is an (enumeration) algorithm that, given $t' \in \mathbb{F}_2^\tau$, enumerates all the (expected number of) $2^{2n/3-\tau}$ pairs $(x^{(2)}, x^{(3)}) \in (\mathbb{F}_2^{n/3})^2$ such that $(f_2(x^{(2)}) \oplus f_3(x^{(3)}))_{[\tau]} = t'$ in time complexity $O(2^{2n/3-\tau})$ and memory complexity $O(2^{n/3-\tau})$. Then, there in an algorithm that solves 3-XOR in time $O(2^{2n/3})$ and memory $O(2^{n/3-\tau})$.*

Here, the memory complexity is measured in terms of the number of words of length $O(n)$ bits.

The 3-XOR algorithm is based on the *clamping through precomputation* technique that was proposed to reduce the memory complexity of k-XOR algorithms in [4] by Bernstein (and subsequently used in several works such as [9,15]). For 3-XOR, the idea is to build a (partial) table for f_1 that fixes its output prefix to $u \in \mathbb{F}_2^\tau$ (XORed with $t_{[\tau]}$), and loop over all prefixes. Specifically, the algorithm below establishes the proposition.

[3] An exception is Attack G2-2, where most calculations involve different operations. For this attack we mainly reuse the analysis of [2].

[4] There are algorithms that save factors polynomial in n for some variants of the problem (e.g. [9,13,15,16]), but these are generally inapplicable in our setting.

```
1. For all $u \in \mathbb{F}_2^{\tau}$:
   (a) – Initialize a table $\mathcal{T}_1$, storing elements in $\mathbb{F}_2^{n/3}$.
       – For all $x^{(1)} \in \mathbb{F}_2^{n/3}$, if $f_1(x^{(1)})_{[\tau]} \oplus t_{[\tau]} = u$, store $x^{(1)}$ at index[a]
         $f_1(x^{(1)}) \oplus t$ in $\mathcal{T}_1$.
   (b) Run the algorithm of Proposition 1 on input $t' = u$. For each pair
       $(x^{(2)}, x^{(3)})$ returned:
       – Search $\mathcal{T}_1$ for $f_2(x^{(2)}) \oplus f_3(x^{(3)})$. If a match $x^{(1)}$ exists, return
         $(x^{(1)}, x^{(2)}, x^{(3)})$ as a solution to 3-XOR.
```

[a] The index $f_1(x^{(1)}) \oplus t$ is the input to the hash function of $\mathcal{T}_1$.

Analysis.

Correctness. A 3-XOR solution satisfies $f_1(x^{(1)}) \oplus t = f_2(x^{(2)}) \oplus f_3(x^{(3)})$, and therefore if $f_1(x^{(1)})_{[\tau]} \oplus t_{[\tau]} = u$, then $(f_2(x^{(2)}) \oplus f_3(x^{(3)}))_{[\tau]} = u$. Thus, for $u = f_1(x^{(1)})_{[\tau]} \oplus t_{[\tau]}$, $(x^{(2)}, x^{(3)})$ is returned by the enumeration algorithm and the solution is output.

Complexity. For each of the 2^{τ} iterations, Step 1.(a) requires $O(2^{n/3})$ time, while (by the assumption of Proposition 1) Step 1.(b) requires time $O(2^{2n/3-\tau})$. The total time complexity is thus $O(2^{2n/3})$ as claimed. The probability that $f_1(x^{(1)})_{[\tau]} \oplus t_{[\tau]} = u$ is $2^{-\tau}$. The number of elements stored in $\mathcal{T}_1$ in each iteration is therefore $O(2^{n/3-\tau})$ with high probability, and by the assumption of Proposition 1, this dominates the memory complexity of the algorithm.

Enumeration algorithm for Proposition 1. Below we describe a simple enumeration algorithm[5] for Proposition 1. We do not use this algorithm and it is only described for the sake of completeness.

```
1. For all $u' \in \mathbb{F}_2^{\tau}$:
   (a) – Initialize a table $\mathcal{T}_2$, storing elements in $\mathbb{F}_2^{n/3}$.
       – For all $x^{(2)} \in \mathbb{F}_2^{n/3}$, if $f_2(x^{(2)})_{[\tau]} \oplus t' = u'$, store $x^{(2)}$ in $\mathcal{T}_2$.
   (b) For all $x^{(3)} \in \mathbb{F}_2^{n/3}$, if $f_3(x^{(3)})_{[\tau]} = u'$:
       – For all $x^{(2)}$ in $\mathcal{T}_2$, output $(x^{(2)}, x^{(3)})$.
```

Complexity analysis. The total time complexity is $O(\max(2^{n/3+\tau}, 2^{2n/3-\tau}))$, where $2^{2n/3-\tau}$ represents the expected number of output pairs. The memory complexity is $O(2^{n/3-\tau})$.

[5] The full 3-XOR algorithm is similar to the 3-SUM algorithm of Wang [20].

Setting $\tau = n/6$ optimizes the time complexity of the algorithm. Combined with Proposition 1, this gives a 3-XOR algorithm with time and memory complexities of $O(2^{2n/3})$ and $O(2^{n/6})$, respectively.

2.5 4-XOR Problem

We consider the following variant of the 4-XOR problem. For simplicity, assume the parameter n is divisible by 4.

Definition 2 (4-XOR). *Given access to 4 random functions $f_1, f_2, f_3, f_4 : \mathbb{F}_2^{n/4} \to \mathbb{F}_2^n$ and a target $t \in \mathbb{F}_2^n$, find $(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}) \in (\mathbb{F}_2^{n/4})^4$ such that $f_1(x^{(1)}) \oplus f_2(x^{(2)}) \oplus f_3(x^{(3)}) \oplus f_4(x^{(4)}) = t$.*

As we have an n-bit condition on each quartet $(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)})$, the average number of solutions is $2^{4 \cdot n/4} \cdot 2^{-n} = 1$.

A naive meet-in-the-middle algorithm has time complexity of about $2^{n/2}$ and requires $2^{n/2}$ words of memory. It is not known how to substantially improve its time complexity. On the other hand, the memory complexity of the naive algorithm can be significantly reduced to $2^{n/4}$ using a variant of the Schroeppel-Shamir algorithm [18], which is described in [12] (for the subset-sum problem). The idea is to enumerate over all $u \in \mathbb{F}_2^{n/4}$, representing the values

$$f_1(x^{(1)})_{[n/4]} \oplus f_2(x^{(2)})_{[n/4]} \oplus t_{[\tau]} \text{ and } f_3(x^{(3)})_{[n/4]} \oplus f_4(x^{(4)})_{[n/4]},$$

which are equal for a 4-XOR solution. This allows to split the 4-XOR problem into two 2-XOR problems, each solved by a MITM procedure. The solutions of the two 2-XOR problems are then merged to give a solution to the original 4-XOR problem.

Wagner's k-tree algorithm [19] provides an improvement for k-XOR when the domains of the functions are larger. For $k = 4$, if $f_1, f_2, f_3, f_4 : \mathbb{F}_2^{n/3} \to \mathbb{F}_2^n$, then the number of expected solutions is $2^{4 \cdot n/3} \cdot 2^{-n} = 2^{n/3}$ and the k-tree algorithm finds one of them in time and memory complexities of $O(2^{n/3})$. The high-level idea is that we only need to enumerate over a single $u \in \mathbb{F}_2^{n/3}$ to find a solution with high probability.

In the general case where $f_1, f_2, f_3, f_4 : \mathbb{F}_2^{\kappa} \to \mathbb{F}_2^n$ for $n/4 \leq \kappa \leq n/3$, a full tradeoff algorithm was devised in [12]. Its time complexity is $O(2^{n-2\kappa})$, while its memory complexity is $O(2^{\kappa})$.

More details about these algorithms are given in Appendix A.

3 Memory-Optimized Attack on GEA-1

In this section we describe our memory-optimized attack on GEA-1. We begin by describing the findings of [2] regarding the initialization process of GEA-1 in

Section 3.1, and the corresponding attack in Section 3.2. We then optimize the memory complexity in two steps. The first step is based on a simple observation and reduces the memory complexity by a factor of about $2^8 = 256$ to 128 MiB. The second step further reduces the memory complexity by a factor of about $2^5 = 32$ to 4 MiB. While the additional reduction is only by a factor of 32, it is clearly non-negligible and technically more interesting. Furthermore, some of the ideas will be reused in Attack G2-2 on GEA-2.

3.1 Weakness in the GEA-1 Initialization Process

The initialization process of GEA-1 defines an injective mapping $M : \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{96}$ which maps the seed to an initial state $(\hat{A}, \hat{B}, \hat{C})$. We can decompose the mapping according to its projections on the different registers:

$$M_A : \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{31}, M_B : \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{32}, M_C : \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{33}.$$

Further define

$$M_{AC} : \mathbb{F}_2^{64} \rightarrow \mathbb{F}_2^{64}$$

as the projection of M onto $(\hat{A}, \hat{C})$.

Crucially, it was observed in [2] that $\dim(\ker(M_{AC})) = 24$, where ideally it should be 0. This implies that $\dim(\mathrm{Im}(M_{AC})) = 64 - 24 = 40$ and thus the state $(\hat{A}, \hat{C})$ obtained after initialization can only assume 2^{40} values.

Decomposition of the initialization mapping. We have $\dim(\mathrm{Im}(M_B)) = 32$ and therefore $\dim(\ker(M_B)) = 64 - 32 = 32$ (and also $\dim(\ker(M_{AC})) = 24$). Furthermore, $\dim(\ker(M_B) \cap \ker(M_{AC})) = 0$.

Hence, $\mathbb{F}_2^{64}$ can be decomposed as a direct sum into

$$\mathbb{F}_2^{64} = W^{(1)} \boxplus \ker(M_{AC}) \boxplus \ker(M_B),$$

where $\dim(W^{(1)}) = 64 - \dim(\ker(M_{AC})) - \dim(\ker(M_B)) = 8$.

It will be more convenient to work directly over $\mathrm{Im}(M)$ rather than over $\mathbb{F}_2^{64}$ (here, we slightly deviate from [2]). Thus, let

$U^{(B)} = \{(0, x, 0) \in \mathbb{F}_2^{31} \times \mathbb{F}_2^{32} \times \mathbb{F}_2^{33}\}$ and $U^{(AC)} = \{(x, 0, y) \in \mathbb{F}_2^{31} \times \mathbb{F}_2^{32} \times \mathbb{F}_2^{33}\}$.

Define $V^{(1)}$ as the image of $W^{(1)}$ under M, $V^{(2)} \subset U^{(B)}$ as the image of $\ker(M_{AC})$ under M and $V^{(3)} \subset U^{(AC)}$ as the image of $\ker(M_B)$ under M. We have

$$\mathrm{Im}(M) = V^{(1)} \boxplus V^{(2)} \boxplus V^{(3)}, \tag{1}$$

where

$$\dim(V^{(1)}) = 8, \dim(V^{(2)}) = 24, \dim(V^{(3)}) = 32.$$

The decomposition above implies that every state $(\hat{A}, \hat{B}, \hat{C}) \in \mathrm{Im}(M)$ (obtained after initialization) can be uniquely represented by a triplet

$$(v^{(1)}, v^{(2)}, v^{(3)}) \in V^{(1)} \times V^{(2)} \times V^{(3)}$$

such that

$$(\hat{A}, \hat{B}, \hat{C}) = v^{(1)} \oplus v^{(2)} \oplus v^{(3)}.$$

Since $v^{(2)}_{[AC]} = 0$ and $v^{(3)}_{[B]} = 0$, then

$$\begin{aligned} \hat{B} &= (v^{(1)} \oplus v^{(2)} \oplus v^{(3)})_{[B]} = (v^{(1)} \oplus v^{(2)})_{[B]} \text{ and} \\ (\hat{A}, \hat{C}) &= (v^{(1)} \oplus v^{(2)} \oplus v^{(3)})_{[AC]} = (v^{(1)} \oplus v^{(3)})_{[AC]}. \end{aligned} \tag{2}$$

3.2 Basic Meet-in-the-Middle Attack

Below we describe the basic attack of [2] with minor differences and using somewhat different notation. We assume for simplicity that the algorithm is given as input the consecutive keystream $z_{[32]}$, and additional keystream that allows verifying that the initial state (or key) is correctly recovered. However, as noted in [2], it can be easily adjusted to use only 24 bits from the same frame.

```
1. For all $v^{(1)} \in V^{(1)}$:
   (a) Initialize a table $\mathcal{T}_B^{v^{(1)}}$, storing elements in $\mathbb{F}_2^{32}$.
   (b) For all $v^{(2)} \in V^{(2)}$, let $\hat{B} = (v^{(1)} \oplus v^{(2)})_{[B]}$. Store $\hat{B}$ in $\mathcal{T}_B^{v^{(1)}}$ at index
       $f_B(\hat{B})_{[32]}$.
2. For all $v^{(1)} \in V^{(1)}$:
   (a) For all $v^{(3)} \in V^{(3)}$, let $(\hat{A}, \hat{C}) = (v^{(1)} \oplus v^{(3)})_{[AC]}$. Search $\mathcal{T}_B^{v^{(1)}}$ for
       $f_A(\hat{A})_{[32]} \oplus f_C(\hat{C})_{[32]} \oplus z_{[32]}$. For each match $\hat{B}$:
       – Test the state $(\hat{A}, \hat{B}, \hat{C})$, and if the test succeeds, recover and
         output the key.
```

Since the first step is independent of the keystream, in [2] it was performed in preprocessing.

Testing states. A state $(\hat{A}, \hat{B}, \hat{C})$ is tested by using it to produce more output and comparing with the (additional) available keystream. Since there are 2^{64} possible initial states and the attack directly exploits 32 bits of available keystream, the expected number of states to test is $2^{64-32} = 2^{32}$.

Complexity analysis. The memory complexity is $2^8 \cdot 2^{24} = 2^{32}$ words (dominated by the 2^8 tables $\mathcal{T}_B^{v^{(1)}}$, each of size 2^{24}) and the time complexity is 2^{40}, dominated by the second step. It is assumed to dominate the complexity of testing the 2^{32} states.

3.3 Basic Memory-Optimized Attack

In the previous attack the decomposition is only used to obtain a post-filtering condition. Specifically, all vectors in $V^{(1)}$ are iterated over independently in both steps, and $v^{(1)} \in V^{(1)}$ determines which small table to access in the second step. We construct an outer loop over the elements of the common subspace $V^{(1)}$. This allows to divide the computation of the previous attack into 2^8 independent parts, each using a single small table. We remark that unlike the previous attack, the small tables are no longer computed during preprocessing. Nevertheless, the memory-optimized attack seems favorable, as the online complexity is similar to the previous one, while the memory complexity is reduced. The details of the algorithm are provided below. It is given as input the keystream $z_{[32]}$.

```
1. For all $v^{(1)} \in V^{(1)}$:
   (a) – Initialize a table $\mathcal{T}_B$, storing elements in $\mathbb{F}_2^{32}$.
       – For all $v^{(2)} \in V^{(2)}$, let $\hat{B} = (v^{(1)} \oplus v^{(2)})_{[B]}$. Store $\hat{B}$ at index
         $f_B(\hat{B})_{[32]} \oplus z_{[32]}$ in $\mathcal{T}_B$.
   (b) For all $v^{(3)} \in V^{(3)}$, let $(\hat{A}, \hat{C}) = (v^{(1)} \oplus v^{(3)})_{[AC]}$. Search $\mathcal{T}_B$ for
       $f_A(\hat{A})_{[32]} \oplus f_C(\hat{C})_{[32]}$. For each match $\hat{B}$:
       – Test the state $(\hat{A}, \hat{B}, \hat{C})$, and if the test succeeds, recover and
         output the key.
```

Analysis.

Correctness. Let $(\hat{A}, \hat{B}, \hat{C})$ be the internal state used to produce the keystream. In particular, it satisfies

$$f_B(\hat{B})_{[32]} \oplus z_{[32]} = f_A(\hat{A})_{[32]} \oplus f_C(\hat{C})_{[32]}. \tag{3}$$

Consider its decomposition $(v^{(1)}, v^{(2)}, v^{(3)}) \in V^{(1)} \times V^{(2)} \times V^{(3)}$ such that $\hat{B} = (v^{(1)} \oplus v^{(2)})_{[B]}$ and $(\hat{A}, \hat{C}) = (v^{(1)} \oplus v^{(3)})_{[AC]}$. By (3), this state is tested in Step 1.(b) for the corresponding value of $v^{(1)}$ and the correct key is output.

Complexity. The time complexity of the attack remains $2^8 \cdot 2^{32} = 2^{40}$, dominated by the 2^8 executions of Step 1.(b). The memory complexity (dominated by each $\mathcal{T}_B$) is 2^{24} words.

3.4 Attack G1 – Improved Memory-Optimized Attack

We now revisit the previous attack on GEA-1, with the aim of further improving its memory complexity with only a minor effect on time complexity. Specifically, similarly to 3-XOR algorithms, given a prefix string, we would like to devise an efficient enumeration algorithm for internal states $(\hat{A}, \hat{C})$ that output this prefix (f_A and f_C replace f_2 and f_3 in Definition 1).

For GEA-1 we are only interested in a small fraction of states $(\hat{A}, \hat{C})$ that can be produced by the initialization process. On the other hand, the standard enumeration algorithm used in Section 2.4 for the 3-XOR problem does not impose such restrictions and therefore mostly outputs states that are irrelevant for us, rendering it inefficient for our purpose. Therefore, we need to devise a more dedicated algorithm.

High-level overview of the attack. The attack is based on an enumeration algorithm similarly to the 3-XOR algorithm of Section 2.4. Specifically, Proposition 2 below is analogous to Proposition 1 for 3-XOR. It isolates the challenge in improving the memory complexity and allows to design the algorithm in a modular way.

Let $V^{(3)}_{[AC]} \subset \mathbb{F}_2^{64}$ be the projection of $V^{(3)}$ in (1) on the registers A and C (since $v^{(3)}_{[B]} = 0$ for all $v^{(3)} \in V^{(3)}$, the projection does not reduce its dimension). Essentially, the challenge is to enumerate all states $(\hat{A}, \hat{C})$ in the 32-dimensional coset $v^{(1)}_{[AC]} \oplus V^{(3)}_{[AC]}$ that produce a given output prefix efficiently with limited memory.

Proposition 2. *Let $\tau \in [8]$ be a parameter. Assume there is a state enumeration algorithm that given a target $u \in \mathbb{F}_2^{\tau}$ and a vector $v^{(1)} \in V^{(1)}$, enumerates all the (expected number of) $2^{32-\tau}$ states $(\hat{A}, \hat{C})$ such that $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[\tau]} \oplus f_C(\hat{C})_{[\tau]} = u$ in time complexity $2^{32-\tau}$ and memory complexity 2^m words of 32 bits. Then, there is a key-recovery attack on GEA-1 in time complexity 2^{40} and memory complexity about $2^{24-\tau} + 2^m$ words of 32 bits.*

Obviously, we would like to have $2^m \ll 2^{24-\tau}$ so the overall memory complexity is about $2^{24-\tau}$.

Note that if $(\hat{A}, \hat{C}) = (v^{(1)} \oplus v^{(3)})_{[AC]}$ as in the previous attack, then $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ as in the above proposition.

We now describe the key-recovery attack that establishes the proposition. It is based on the clamping through precomputation technique similarly to the 3-XOR algorithm of Proposition 1. Yet, it uses the additional constraint on the states (similarly to the basic GEA-1 attack above). As previously, the attack directly utilizes a keystream $z_{[32]}$.

```
Attack G1
  1. For all $v^{(1)} \in V^{(1)}$ and all $u \in \mathbb{F}_2^{\tau}$:
      (a)  – Initialize a table $\mathcal{T}_B$, storing elements in $\mathbb{F}_2^{32}$.
           – For all $v^{(2)} \in V^{(2)}$, let $\hat{B} = (v^{(1)} \oplus v^{(2)})_{[B]}$. If $f_B(\hat{B})_{[\tau]} \oplus z_{[\tau]} = u$,
             store $\hat{B}$ at index $f_B(\hat{B})_{[32]} \oplus z_{[32]}$ in $\mathcal{T}_B$.
```

(b) – Run the algorithm of Proposition 2 on inputs $v^{(1)}$ and u.

– For each state $(\hat{A},\hat{C})$ returned (satisfying $(\hat{A},\hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[\tau]} \oplus f_C(\hat{C})_{[\tau]} = u$), search $\mathcal{T}_B$ for $f_A(\hat{A})_{[32]} \oplus f_C(\hat{C})_{[32]}$. For each match $\hat{B}$:

- Test the state $(\hat{A},\hat{B},\hat{C})$, and if the test succeeds, recover and output the key.

Analysis.

Correctness. Let $(\hat{A},\hat{B},\hat{C})$ be the internal state used to produce the keystream. In particular

$$f_B(\hat{B})_{[32]} \oplus z_{[32]} = f_A(\hat{A})_{[32]} \oplus f_C(\hat{C})_{[32]}.$$

We show that when iterating over $v^{(1)}$ and u satisfying $u = f_B(\hat{B})_{[\tau]} \oplus z_{[\tau]} = f_A(\hat{A})_{[\tau]} \oplus f_C(\hat{C})_{[\tau]}$, this state is tested and thus the key is returned.

Consider the state's decomposition $(v^{(1)}, v^{(2)}, v^{(3)}) \in V^{(1)} \times V^{(2)} \times V^{(3)}$ such that $\hat{B} = (v^{(1)} \oplus v^{(2)})_{[B]}$ and $(\hat{A},\hat{C}) = (v^{(1)} \oplus v^{(3)})_{[AC]}$. For $u = f_B(\hat{B})_{[\tau]} \oplus z_{[\tau]}$, $\hat{B}$ is stored at index $f_B(\hat{B})_{[32]} \oplus z_{[32]}$ in $\mathcal{T}_B$.

Since $(\hat{A},\hat{C}) \oplus v^{(1)}_{[AC]} = v^{(3)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[\tau]} \oplus f_C(\hat{C})_{[\tau]} = u$, the enumeration algorithm returns $(\hat{A},\hat{C})$ and $(\hat{A},\hat{B},\hat{C})$ is tested as claimed.

Complexity. The complexity of all $2^{8+\tau}$ executions of Step 1.(a) is $2^{8+\tau} \cdot 2^{24} = 2^{32+\tau} \leq 2^{40}$ evaluations of (32 bits of) f_B. By Proposition 2, the complexity of all $2^{8+\tau}$ executions of Step 1.(b) is $2^{8+\tau} \cdot 2^{32-\tau} = 2^{40}$ (evaluations of f_A and f_C) and it dominates the complexity of the attack. The memory complexity is dominated by $\mathcal{T}_B$ in addition to 2^m of the enumeration algorithm and is $2^{24-\tau} + 2^m$ words of 32 bits, as claimed.

Devising a state enumeration algorithm. We have reduced the goal to devising a state enumeration algorithm. If we assume that f_A, f_C are random functions, then clearly we cannot produce all solutions required by Proposition 2 in $2^{32-\tau} < 2^{32}$ time (regardless of the memory complexity), since the size of the domain of $\hat{C}$ is 2^{33} (and the number of vectors that satisfy $(\hat{A},\hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ is 2^{32}). Our main observation is that the functions f_A, f_C are not random and we can utilize their specific properties to devise a dedicated algorithm for GEA-1.

Proposition 3 (State enumeration algorithm for GEA-1). *For $\tau = 5$ and $m = 7$, there is a state enumeration algorithm for GEA-1. Specifically, given inputs $v^{(1)} \in V^{(1)}$ and $u \in \mathbb{F}_2^5$, there is an algorithm that enumerates all the $2^{40-8-5} = 2^{27}$ states $(\hat{A},\hat{C})$ such that $(\hat{A},\hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[5]} \oplus f_C(\hat{C})_{[5]} = u$ in time complexity 2^{27} using $2^7 \ll 2^{19}$ memory words of 32-bits.*

Therefore, Proposition 2 implies that we can recover the key of GEA-1 in time complexity 2^{40} and memory complexity (slightly more than) $2^{19} + 2^7$ words of 32 bits.[6] Below we describe the details of the algorithm.

Influence of the state on the output. We observe that for all registers, only a subset of the internal state bits influence the first output bits. Specifically, we will exploit the following property, which is easily deduced from Figure 1.

Property 1 (Influence of the state on the output).

- $f_A(\hat{A})_{[5]}$ only depends on $31 - 5 = 26$ bits of $\hat{A}$.
- $f_B(\hat{B})_{[5]}$ only depends on $32 - 7 = 25$ bits of $\hat{B}$.
- $f_B(\hat{C})_{[5]}$ only depends on $33 - 11 = 22$ bits of $\hat{C}$.

Denote these 26 (resp. $25, 22$) state bit indices of A (resp. B, C) by J_A (resp. J_B, J_C). We note that we use the above property only for registers A and C.

Initial attempt. An initial idea that exploits Property 1 is to prepare a table for all possible 2^{22} values of J_C. Then, enumerate over the 2^{26} bits of J_A and merge the (partial) states according to the linear constraints imposed by $V^{(3)}$ via the relation $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and the output constraint $f_A(\hat{A})_{[5]} \oplus f_C(\hat{C})_{[5]} = u$. While this algorithm satisfies the required time complexity, it does not give the desired memory saving.

Decomposition by influential bits. Let $V^{(3)}_{[J_A J_C]} \subset \mathbb{F}_2^{48}$ denote the projection of $V^{(3)}$ on the 48 influential bits $J_A \cup J_C$. Using a computer program (see Appendix C.2), we calculated $\dim(V^{(3)}_{[J_A J_C]}) = \dim(V^{(3)}) = 32$.

Recall that we are only interested in states $(\hat{A}, \hat{C})$ that satisfy $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$, namely contained in the 32-dimensional coset $(v^{(1)} \oplus V^{(3)})_{[AC]}$. Moreover, as we are only interested in the first 5 output bits, it is sufficient to consider only 48-bit partial states in the projected coset $(v^{(1)} \oplus V^{(3)})_{[J_A J_C]}$ and then complement them to full 64-bit states $(\hat{A}, \hat{C})$.

Since $\dim(V^{(3)}_{[J_A J_C]}) = 32$, it is not efficient to iterate over its elements directly, but the main observation is that we can decompose it according to the bits of J_A and J_C and perform a MITM procedure as in the initial attempt above, but with less memory.

[6] Based on this computation, the algorithm requires about 2 MiB of memory. However, if we use the data structure used in the GEA-1 attack of [2], the memory complexity would be 4 MiB (which is what we claim).

We restrict the discussion to the 48-bit subspace spanned by $J_A \cup J_C$ (viewed as unit vectors). Let $U^{(J_A)} \subset \mathbb{F}_2^{48}$ be the 26-dimensional subspace whose vectors are zero on the bits of J_C. Define the 22-dimensional subspace $U^{(J_C)}$ similarly.

We have

$$\dim(V^{(3)}_{[J_A J_C]} \cap U^{(J_A)}) \geq \dim(V^{(3)}_{[J_A J_C]}) + \dim(U^{(J_A)}) - 48 = 32 + 26 - 48 = 10,$$

and similarly, $\dim(V^{(3)}_{[J_A J_C]} \cap U^{(J_C)}) \geq 32 + 22 - 48 = 6$ (both hold with equality, as verified by our program in Appendix C.2). Since $\dim(U^{(J_A)} \cap U^{(J_C)}) = 0$, we can decompose

$$V^{(3)}_{[J_A J_C]} = V^{(4)} \boxplus V^{(A)} \boxplus V^{(C)}, \tag{4}$$

where $V^{(A)} \subset U^{(J_A)}$ and $\dim(V^{(A)}) = 10$, while $V^{(C)} \subset U^{(J_C)}$ and $\dim(V^{(C)}) = 6$. Therefore, $\dim(V^{(4)}) = 32 - 10 - 6 = 16$.

The additional decomposition allows to divide the computation of the MITM procedure in the initial attempt above into 2^{16} independent smaller procedures, one for each $v^{(4)} \in V^{(4)}$. Consequently, the size of the table for J_C is reduced to $2^{22-16} = 2^6$, while we need to enumerate over $2^{26-16} = 2^{10}$ values for the bits of J_A and match with the table on the 5-bit output u. The average number of matches in the table per $v^{(4)} \in V^{(4)}$ is $2^{6+10-5} = 2^{11}$, and this matching phase dominates the complexity (which is $2^{16} \cdot 2^{11} = 2^{27}$ as required by Proposition 3). We give the details below.

State enumeration algorithm for GEA-1. Based on the decomposition

$$V^{(3)}_{[J_A J_C]} = V^{(4)} \boxplus V^{(A)} \boxplus V^{(C)},$$

given in (4), any partial state $(x_A, y_C) \in V^{(3)}_{[J_A J_C]}$ is decomposed as

$$\begin{aligned} x_A &= v^{(4)}_{[J_A]} \oplus v^{(A)}_{[J_A]} \oplus v^{(C)}_{[J_A]} = v^{(4)}_{[J_A]} \oplus v^{(A)}_{[J_A]} \text{ and} \\ y_C &= v^{(4)}_{[J_C]} \oplus v^{(A)}_{[J_C]} \oplus v^{(C)}_{[J_C]} = v^{(4)}_{[J_C]} \oplus v^{(C)}_{[J_C]}. \end{aligned}$$

Partial states relevant for the MITM procedure in the coset $(\tilde{A}, \tilde{C}) \in (v^{(1)} \oplus V^{(3)})_{[J_A J_C]}$ are similarly decomposed as

$$\tilde{A} = v^{(1)}_{[J_A]} \oplus v^{(4)}_{[J_A]} \oplus v^{(A)}_{[J_A]} \text{ and } \tilde{C} = v^{(1)}_{[J_C]} \oplus v^{(4)}_{[J_C]} \oplus v^{(C)}_{[J_C]}.$$

This is the main decomposition used by the algorithm.

Yet, as the algorithm needs to return full 64-bit states and not partial states, it will be more convenient to directly work with 64-bit vectors and project them to partial states when needed. For this purpose, note that since $\dim(V^{(3)}_{[J_A J_C]}) = \dim(V^{(3)}) = 32$, then any 48-bit vector $v \in V^{(3)}_{[J_A J_C]}$ can be uniquely extended via linear algebra to a 64-bit vector $v' \in V^{(3)}_{[AC]}$ such that $v = v'_{[J_A J_C]}$.

Similarly, the subspaces $V^{(4)}, V^{(A)}, V^{(C)}$ (of $V^{(3)}_{[J_A J_C]}$) can be uniquely extended to subspaces $V^{(4')}, V^{(A')}, V^{(C')}$ (of $V^{(3)}_{[AC]}$) such that $V^{(4')}_{[J_A J_C]} = V^{(4)}, V^{(A')}_{[J_A J_C]} = V^{(A)}, V^{(C')}_{[J_A J_C]} = V^{(C)}$. Moreover, any $v^{(3')} \in V^{(3)}_{[AC]}$ can be uniquely written as

$$v^{(3')} = v^{(4')} \oplus v^{(A')} \oplus v^{(C')}, \tag{5}$$

where $(v^{(4')}, v^{(A')}, v^{(C')}) \in V^{(4')} \times V^{(A')} \times V^{(C')}$.

Details of the state enumeration algorithm. We extend the output functions $f_A(\hat{A})_{[5]}$ and $f_C(\hat{C})_{[5]}$ to work with partial states $\tilde{A} \in \mathbb{F}_2^{26}$ and $\tilde{C} \in \mathbb{F}_2^{22}$, respectively.

Recall that the state enumeration algorithm receives inputs $v^{(1)} \in V^{(1)}$ and $u \in \mathbb{F}_2^5$ and enumerates all $2^{40-8-5} = 2^{27}$ states $(\hat{A}, \hat{C})$ such that $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[5]} \oplus f_C(\hat{C})_{[5]} = u$. The algorithm is given below.

```
1. For all $v^{(4')} \in V^{(4')}$:
   (a) – Initialize a table $\mathcal{T}_C$, storing elements in $\mathbb{F}_2^{64}$.
       – For all $v^{(C')} \in V^{(C')}$, let $\tilde{C} = v^{(1)}_{[J_C]} \oplus v^{(4')}_{[J_C]} \oplus v^{(C')}_{[J_C]}$. Store $v^{(C')}$
         at index $f_C(\tilde{C})_{[5]} \oplus u$ in $\mathcal{T}_C$.
   (b) For all $v^{(A')} \in V^{(A')}$, let $\tilde{A} = v^{(1)}_{[J_A]} \oplus v^{(4')}_{[J_A]} \oplus v^{(A')}_{[J_A]}$. Search $\mathcal{T}_C$ for
       $f_A(\tilde{A})_{[5]}$. For each match $v^{(C')}$:
       – Let $v^{(3')} = v^{(4')} \oplus v^{(A')} \oplus v^{(C')}$ and return $(\hat{A}, \hat{C}) = v^{(1)}_{[AC]} \oplus v^{(3')}$.
```

Analysis.

Correctness. Let $(\hat{A}, \hat{C})$ be such that $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[5]} \oplus f_C(\hat{C})_{[5]} = u$. Then, we can write $(\hat{A}, \hat{C}) = v^{(1)}_{[AC]} \oplus v^{(3')}$, where $v^{(3')} \in V^{(3)}_{[AC]}$, and $v^{(3')} = v^{(4')} \oplus v^{(A')} \oplus v^{(C')}$ as in (5). Then, the partial state $(\tilde{A}, \tilde{C}) = (\hat{A}, \hat{C})_{[J_A J_C]}$ is considered when iterating over $v^{(4')}$, and $(\hat{A}, \hat{C})$ is returned as required.

Complexity. The heaviest step is 1.(b). For each $v^{(4')} \in V^{(4')}$, its complexity is 2^{10} for iterating over $v^{(A')} \in V^{(A')}$. The expected number of matches in $\mathcal{T}_C$ is $2^{10} \cdot 2^6 \cdot 2^{-5} = 2^{11}$ (it is a 5-bit matching). Hence, the total complexity of each iteration is about 2^{11}, while the total complexity is $2^{16} \cdot 2^{11} = 2^{27}$ as claimed in Proposition 3. In terms of memory, table $\mathcal{T}_C$ requires 2^6 words of 64 bits.

Optimality of parameters for the GEA-1 attack. The algorithm enumerates all states $(\hat{A}, \hat{C})$ such that $(\hat{A}, \hat{C}) \oplus v^{(1)}_{[AC]} \in V^{(3)}_{[AC]}$ and $f_A(\hat{A})_{[5]} \oplus f_C(\hat{C})_{[5]} = u$. The main parameter is the length of the output prefix, which is set to $\tau = 5$. A shorter output prefix results in a larger table $\mathcal{T}_B$, increasing the memory complexity. On the other hand, a longer output prefix (e.g., $\tau = 6$) implies that $f_A(\hat{A})_{[6]}$ and $f_C(\hat{C})_{[6]}$ depend on too many internal state bits and it is not clear how to enumerate the relevant $2^{40-8-6} = 2^{26}$ states in about 2^{26} time (and limited memory) to obtain total time complexity of 2^{40}.

Implementation. We implemented the attack in `C++` and experimentally verified it on a laptop with an AMD Ryzen-7 5800H processor. The program recovered the GEA-1 session key in 153 minutes, averaged over 5 runs. As the attack of [2] was implemented on a cluster, it cannot be directly compared to ours. Nevertheless, we give a rough comparison in terms of CPU time: our attack takes $6\times$ time using $32\times$ less cores which are $1.5\times$ faster. This seems favorable and is possibly a consequence of the reduced allocated memory fitting in cache.

4 Attacks on GEA-2

In this section we analyze the GEA-2 cipher. We begin by giving an overview of the attacks of [2], as our attacks reuse some of their techniques.

We then describe a simple attack that is based on the Schroeppel-Shamir variant for 4-XOR. This attack needs only a small amount of keystream. Its time complexity is about 2^{63} and it requires roughly 32 GiB of memory. We subsequently describe Attack G2-1 that improves the simple attack in a scenario where a longer keystream sequence is available: given a consecutive keystream of ℓ bits, the time complexity is about $2^{64}/(\ell - 62)$, while the memory complexity is about 64 GiB accessed randomly (and additional 96 GiB of storage accessed sequentially, which can be eliminated at a small cost).

Finally, we describe several attacks that target the initialization of GEA-2, including Attack G2-2. As we explain, for technical reasons the current results are mostly interesting in case the attacker obtains a long yet fragmented keystream (not containing a long window of consecutive known bits). Compared to [2], Attack G2-2 provides an improvement by a factor of (at least) $2^9 = 512$ in memory complexity in the considered scenario.

4.1 Previous Attacks on GEA-2

Let $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ be an internal state. Since the algebraic degree of the filter function f is 4, any consequent output bit can be symbolically represented as a polynomial of algebraic degree 4 over $\mathbb{F}_2$ in terms of the 125 bits of $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$, treated as variables.

Assume we receive the encryption of a fully known GEA-2 frame, thus obtaining 12800 keystream bits. Hence, we can construct a system of 12800 polynomial equations of degree 4 in 125 variables. Since the registers are independent, the number of monomials that appear in the polynomials is upper bounded by

$$1+\sum_{i=1}^{4}\binom{29}{i}+\binom{31}{i}+\binom{32}{i}+\binom{33}{i}=152682.$$

Attempting to apply a linearization attack, we replace every monomial in each polynomial equation with an independent variable and try to eliminate variables by Gaussian elimination on the 12800 linearized polynomial representations of the keystream bits. Unfortunately, the number of variables is much larger than the 12800 available equations, rendering this straightforward approach useless.

Therefore, [2] considers a hybrid approach in which we guess some variables in order to reduce the number of monomials. However reducing the number of monomials to 12800 seems to require guessing at least 58 variables.[7] Each such guess requires additional linear algebra computations which make the attack slower than exhaustive search.

Hybrid with meet-in-the-middle. The main idea of [2] is to combine the hybrid approach with a MITM procedure. More specifically, the idea is to guess some bits of the internal states of the shorter registers A and D and eliminate their contribution from the keystream by linearization. Then, perform a MITM procedure on the registers B and C.

We give a high-level overview of this attack. Let $(\hat{A},\hat{B},\hat{C},\hat{D})$ be an unknown internal state that produces $z_{[12800]}$. Guess 11 bits of $\hat{A}$ and 9 bits of $\hat{D}$. This reduces the number of monomials in the remaining $20+20$ unknown variables in these registers to $\sum_{i=1}^{4}\binom{20}{i}+\binom{20}{i}=12390$. By Gaussian elimination, find $12800-12390=410$ linear expressions (masks) of length 12800, each eliminating the contributions of $\hat{A}$ and $\hat{D}$ from the keystream. The attack essentially only needs 64 of these masks.

Next, apply the 64 masks to the keystream to derive a 64-bit masked keystream (that should depend only on $\hat{B}$ and $\hat{C}$ if the initial guess is correct). Finally, perform a MITM procedure: for each possible value of $\hat{B}$, compute $f_B(\hat{B})_{[12800]}$ and apply the 64 masks. Store $\hat{B}$ indexed by the 64-bit results (after XORing with the masked keystream) in a table. Then, for each possible value of $\hat{C}$, compute $f_C(\hat{C})_{[12800]}$, apply the 64 masks and search the table for the 64-bit value. After additional tests, a match allows to easily construct the full state of GEA-2 and to recover the key if the state is correct. In order to perform all these $2^{32}+2^{33}$ computations of 64 bits efficiently (without expanding the full 12800-bit output and applying the 64 masks), the attack first interpolates the symbolic representations of the 64 masked outputs of $\hat{B}$ and $\hat{C}$ (which are

[7] The authors of [2] showed how to reduce the number of monomials to 12800 by guessing 59 variables. We have found a way to do it by guessing only 58 variables, but this does not have a substantial effect on the attack.

Boolean functions of degree 4). Then, the fast polynomial evaluation algorithm of [8] is used.

There are two optimizations applied to the attack. The first optimization uses the observation that degree 4 monomials produced by the 20 + 20 eliminated variables of $\hat{A}$ and $\hat{D}$ are unchanged by the guesses (as they are not multiplied by any other variable in the original polynomial representations that involve the guessed variables). This allows to perform the Gaussian elimination only once on these $\binom{20}{4} + \binom{20}{4} = 9690$ linearized variables and reduces the complexity of the remaining work for computing the 410 masks.

Overall, the 2^{9+11} performed MITM procedures dominate the time and memory complexities of the attack, which the authors estimate as (about) 2^{54} GEA-2 evaluations, and roughly 2^{32} words, respectively.

Shifted keystreams. The second optimization produces 753 internal state targets for the attack at different clocks. This allows to reduce the number of guesses by a factor of (roughly) 753 (after $2^{20}/753$ guesses, we expect to hit one of the internal state targets). Specifically, the idea is to produce from the 12800-bit keystream 753 shifted consecutive keysteams of length 12047 (keystream i starts from position i). Then, by linear algebra, compute $12047 - 753 + 1 = 11295$ masks (linear expressions), each having a constant value on all 753 keystreams. These 11295 constant bits serve as the keystream input to the previous attack and allows to simultaneously target all 753 shifted keystreams. Since the effective keystream size is reduced to 11295, we now have to guess 21 variables instead of 20 to perform linearization, but we are expected to hit one of the targets much faster. The authors estimate the complexity of this attack by about 2^{45} GEA-2 evaluations. The memory complexity remains roughly 2^{32} words (32 GiB).

The authors also calculated the complexity of the optimized attack when given less data and estimated that it beats exhaustive search given at least 1468 consecutive keystream bits.

Attack on fragmented keystream. We note that the final optimization can only be applied if the attacker obtains a long sequence of consecutive keystream bits. On the other hand, assume the attacker obtains a frame in which 11300 bits of keystream at arbitrary locations are known. In this case, the best attack is the previous one (without the final optimization) that can be adjusted to work in slightly higher complexity of 2^{55} GEA-2 evaluations (instead of 2^{54}), and 2^{32} words of memory.

4.2 Basic 4-XOR Attack

Our first attack adapts the Schroeppel-Shamir variant for 4-XOR (summarized in Section 2.5) to an attack on GEA-2. As in the Schroeppel-Shamir variant, we partition the functions f_A, f_B, f_C, f_D into pairs during the merging process. The time complexity will be dominated by the pair of registers that has the maximal number of possible states. In order to optimize the attack, we consider the pairs (f_C, f_D) and (f_A, f_B) to obtain time complexity of about $2^{31+32} = 2^{63}$ (the number of internal states of registers A and B). This complexity is very close to exhaustive search, and we describe it below mainly as an exposition to Attack G2-1 that follows.

Let $\tau \leq 64$ be a parameter. We enumerate over all $u \in \mathbb{F}_2^{\tau}$, representing the values of

$$f_C(\hat{C})_{[\tau]} \oplus f_D(\hat{D})_{[\tau]} \oplus z_{[\tau]} \text{ and } f_A(\hat{A})_{[\tau]} \oplus f_B(\hat{B})_{[\tau]}.$$

These values are equal for the correct state $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$.

We assume that we have a 64-bit keystream $z_{[64]}$ (although the attack can be applied in additional scenarios).

```
1.  – Initialize a table $\mathcal{T}_D$, storing elements in $\mathbb{F}_2^{29}$.
    – For all $\hat{D} \in \mathbb{F}_2^{29}$, store $\hat{D}$ at index $f_D(\hat{D})_{[\tau]}$ in $\mathcal{T}_D$.
2.  – Initialize a table $\mathcal{T}_A$, storing elements in $\mathbb{F}_2^{31}$.
    – For all $\hat{A} \in \mathbb{F}_2^{31}$, store $\hat{A}$ at index $f_A(\hat{A})_{[\tau]}$ in $\mathcal{T}_A$.
3. For each $u \in \mathbb{F}_2^{\tau}$:
   (a) – Initialize a table $\mathcal{T}_{CD}$, storing pairs in $\mathbb{F}_2^{33} \times \mathbb{F}_2^{29}$.
       – For all $\hat{C} \in \mathbb{F}_2^{33}$, search $\mathcal{T}_D$ for $f_C(\hat{C})_{[\tau]} \oplus u \oplus z_{[\tau]}$. For each match $\hat{D}$:
           • Let $v_{CD} = f_C(\hat{C})_{[\tau+1,64]} \oplus f_D(\hat{D})_{[\tau+1,64]} \oplus z_{[\tau+1,64]}$. Store $(\hat{C}, \hat{D})$ at index $v_{CD}$ in $\mathcal{T}_{CD}$.
   (b) For all $\hat{B} \in \mathbb{F}_2^{32}$, search $\mathcal{T}_A$ for $f_B(\hat{B})_{[\tau]} \oplus u$. For each match $\hat{A}$:
       – Let $v_{AB} = f_A(\hat{A})_{[\tau+1,64]} \oplus f_B(\hat{B})_{[\tau+1,64]}$. Search $v_{AB}$ in $\mathcal{T}_{CD}$. For each match $(\hat{C}, \hat{D})$:
           • Test the state $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ and if the test succeeds, recover and output the corresponding key.
```

Testing states. There are several ways to test a 125-bit state $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$. One way is to compute more output bits and compare them against additional available keystream bits (a total of 125 bits suffice on average). A different way is to exploit the fact that the image of the mapping from the 97-bit seed to the 125-bit state is of dimension 97. This implies that any valid state obtained after initialization must satisfy $125 - 97 = 28$ linear equations, which gives another test that does not require additional keystream bits (in fact, these conditions can be directly incorporated into the final merge). Even if we do not exploit additional linear equations, since we impose a 64-bit conditional on the 125-bit internal state, the expected number of states to test is $2^{125-64} = 2^{61}$. Since the total complexity will be about 2^{63}, we consider the testing time as negligible.

Analysis.

Correctness. Fix any $(\hat{C}, \hat{D})$. Then, for

$$u = f_C(\hat{C})_{[\tau]} \oplus f_D(\hat{D})_{[\tau]} \oplus z_{[\tau]},$$

$f_C(\hat{C})_{[\tau]} \oplus u \oplus z_{[\tau]} = f_D(\hat{D})_{[\tau]}$ is searched in $\mathcal{T}_D$ and $\hat{D}$ is retrieved. Therefore, $(\hat{C}, \hat{D})$ is stored at index v_{CD} in $\mathcal{T}_{CD}$. If $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ is the correct state, then

$$u = f_A(\hat{A})_{[\tau]} \oplus f_B(\hat{B})_{[\tau]}$$

holds as well, implying that when searching $\mathcal{T}_A$ for $f_B(\hat{B})_{[\tau]} \oplus u$, the state $\hat{A}$ is retrieved and $v_{AB} = f_A(\hat{A})_{[\tau+1,64]} \oplus f_B(\hat{B})_{[\tau+1,64]}$ is searched in $\mathcal{T}_{CD}$. Finally, since $v_{AB} = v_{CD}$ for the correct state $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$, then it is tested.

Complexity analysis. The complexity of generating the outputs $\hat{D}$ in the first step and building the table is about 2^{29} (in terms of τ-bit computations of f_D). Similarly, the complexity of the second step for A is about 2^{31}.

For each of the 2^τ iterations of Step 3, the complexity of generating the outputs $\hat{C}$ in Step 3.(a) is about 2^{33}. The expected number of matches in $\mathcal{T}_D$ is $2^{29} \cdot 2^{33} \cdot 2^{-\tau} = 2^{62-\tau}$ (as we match on τ bits), which gives the expected number of entries in $\mathcal{T}_{CD}$. For Step 3.(b), the complexity of generating the outputs $f_B(\hat{B})$ is about 2^{32}. The expected number of matches in $\mathcal{T}_A$ is $2^{31} \cdot 2^{32} \cdot 2^{-\tau} = 2^{63-\tau}$. Overall, we estimate the total time complexity per $u \in \mathbb{F}_2^\tau$ by about $\max(2^{33}, 2^{63-\tau})$ GEA-2 evaluations (producing a 128-bit keystream). To optimize the time complexity (and minimize memory complexity for this choice), we choose $\tau = 30$. This gives total time complexity of $2^{30+33} = 2^{63}$.

The memory complexity of all the 3 tables is $2^{29} + 2^{31} + 2^{32} < 2^{33}$ words.

4.3 Attack G2-1 – Extended 4-XOR Attack

The basic attack requires a short keystream and we would like to optimize it in case additional keystream data is available to the attacker.

We show how to apply a variant of Wagner's k-tree algorithm that solves 4-XOR more efficiently than the Schroeppel-Shamir variant in case there are many solutions. For this purpose, we use the idea of [2] and combine several (shifted) keystreams by computing common masks. This allows to combine multiple targets (internal states at different clocks) for the attack, and has an analogous (although not identical) effect to enlarging the domains of the functions in the original 4-XOR problem.

In this attack, the value $u \in \mathbb{F}_2^\tau$ that we iterate over in the loop will represent the values of the linear masks applied to $f_C(\hat{C}) \oplus f_D(\hat{D}) \oplus z$.

Linear masks. We assume that we have a keystream of length $\ell \geq 64$ bits denoted by $z_{[\ell]}$. For convenience, we assume that ℓ is even. Let $\ell' = (\ell - 62)/2$, and for $j \in [\ell']$ define shifted streams $z^{(j)} = z_{[j,j+\ell'+62]} \in \mathbb{F}_2^{\ell'+63}$. Note that the last index of $z^{(\ell')}$ is keystream bit number $\ell' + \ell' + 62 = \ell$, which is the last index of the stream.

We have ℓ' shifted sequences, each of length $\ell' + 63$ bits, and can compute 64 linearly independent masks $m^{(1)}, \ldots, m^{(64)}$ where $m^{(i)} \in \mathbb{F}_2^{\ell'+63}$ such that for each $i \in [64]$, $m^{(i)} \cdot z^{(j)} = c^{(i)}$ for all $j \in [\ell']$, where $c^{(i)} \in \mathbb{F}_2$ is a constant independent of j (the symbol $\cdot$ denotes inner product mod 2).

Concretely, define a $(\ell'-1) \times (\ell'+63)$-dimensional matrix (denoted by Z), where the j'th row is $z^{(j)} \oplus z^{(\ell')}$. The kernel of this matrix is of dimension (at least) $(\ell'+63) - (\ell'-1) = 64$. The masks are a basis of the kernel and can be computed by Gaussian elimination. The 64 constants $c^{(i)}$ are determined by application of the 64 masks to $z^{(\ell')}$.

Before describing the attack, we define some additional notation: given the masks $m^{(1)}, \ldots, m^{(64)}$ (as an implicit input), and a state $\hat{A}$, let

$$g_A(\hat{A}) = \{m^{(i)} \cdot f_A(\hat{A})_{[\ell'+63]}\}_{i \in [64]} \in \mathbb{F}_2^{64}$$

denote the concatenations of the applications of the 64 masks to the $(\ell' + 63)$-bit output prefix produced by $\hat{A}$. Similar notation is defined for the registers B, C, D. Finally, let $c \in \mathbb{F}_2^{64}$ denote the concatenation of all the constants $c^{(i)}$.

Details of the algorithm. The algorithm is given as input the keystream $z_{[\ell]}$. Let $\tau \leq 64$ be a parameter.

<u>Attack G2-1</u>

1. Given $z_{[\ell]}$, compute the matrix $Z \in \mathbb{F}_2^{(\ell'-1)\times(\ell'+63)}$ defined above. Then, derive the masks $m^{(1)}, \ldots, m^{(64)} \in \mathbb{F}_2^{\ell'+63}$ and $c \in \mathbb{F}_2^{64}$ by Gaussian elimination.
2. – Initialize a table $\mathcal{T}_D$, storing pairs in $\mathbb{F}_2^{29} \times \mathbb{F}_2^{64}$.
   – For all $\hat{D} \in \mathbb{F}_2^{29}$, store $(\hat{D}, g_D(\hat{D}))$ at index $g_D(\hat{D})_{[\tau]}$ in $\mathcal{T}_D$.
   – Build a similar table $\mathcal{T}_A$ for A, storing $(\hat{A}, g_A(\hat{A})) \in \mathbb{F}_2^{31} \times \mathbb{F}_2^{64}$ at index $g_A(\hat{A})_{[\tau]}$.
3. – Initialize a sequential table (array) $\mathcal{T}_B$, storing elements in $\mathbb{F}_2^{64}$.
   – For all $\hat{B} \in \mathbb{F}_2^{32}$, store $g_B(\hat{B})$ in $\mathcal{T}_B$ in entry $\hat{B}$.
   – Build a similar table $\mathcal{T}_C$ for C, storing $g_C(\hat{C}) \in \mathbb{F}_2^{64}$ in entry $\hat{C}$.
4. For each $u \in \mathbb{F}_2^{\tau}$:
   (a) – Initialize a table $\mathcal{T}_{CD}$, storing pairs in $\mathbb{F}_2^{33} \times \mathbb{F}_2^{29}$.

```
        – For all $\hat{C} \in \mathbb{F}_2^{33}$, retrieve $g_C(\hat{C})$ from $\mathcal{T}_C$. Search $\mathcal{T}_D$ for
          $g_C(\hat{C})_{[\tau]} \oplus u \oplus c_{[\tau]}$. For each match $(\hat{D}, g_D(\hat{D}))$:
            • Let $v_{CD} = g_C(\hat{C})_{[\tau+1,64]} \oplus g_D(\hat{D})_{[\tau+1,64]} \oplus c_{[\tau+1,64]}$. Store
              $(\hat{C}, \hat{D})$ at index $v_{CD}$ in $\mathcal{T}_{CD}$.
    (b) For all $\hat{B} \in \mathbb{F}_2^{32}$, retrieve $g_B(\hat{B})$ from $\mathcal{T}_B$. Search $\mathcal{T}_A$ for $g_B(\hat{B})_{[\tau]} \oplus u$.
        For each match $(\hat{A}, g_A(\hat{A}))$:
        – Let $v_{AB} = g_A(\hat{A})_{[\tau+1,64]} \oplus g_B(\hat{B})_{[\tau+1,64]}$. Search $v_{AB}$ in $\mathcal{T}_{CD}$.
          For each match $(\hat{C}, \hat{D})$:
            • Test the state $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$. If the test succeeds, recover and
              output the corresponding key.
```

Testing a state is done by computing output bits and comparing with $z_{[\ell]}$ at all indices $j \in [\ell']$ (on average, we need to compute about $\lceil \log \ell' \rceil < \lceil \log \ell \rceil \leq 14$ output bits). We note that the attack involves precomputation of additional tables $\mathcal{T}_B, \mathcal{T}_C$ in order to avoid recomputing the masks in each iteration.

Analysis.

Correctness. Fixing $(\hat{C}, \hat{D})$, for $u = g_C(\hat{C})_{[\tau]} \oplus g_D(\hat{D})_{[\tau]} \oplus c_{[\tau]}$, $(\hat{C}, \hat{D})$ is stored at index v_{CD} in $\mathcal{T}_{CD}$. If $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ is a state that produced the shifted keystream $z^{(j)}$ for $j \in [\ell']$, then for every $i \in [64]$,

$$\begin{aligned} (g_A(\hat{A}) \oplus g_B(\hat{B}) \oplus g_C(\hat{C}) \oplus g_D(\hat{D}))_i &= \\ m^{(i)} \cdot (f_A(\hat{A}) \oplus f_B(\hat{B}) \oplus f_C(\hat{C}) \oplus f_D(\hat{D}))_{[\ell'+63]} &= \\ m^{(i)} \cdot z^{(j)} &= c^{(i)}, \end{aligned}$$

where the final equality holds by the properties of the masks. Equivalently,

$$c = g_A(\hat{A}) \oplus g_B(\hat{B}) \oplus g_C(\hat{C}) \oplus g_D(\hat{D}).$$

Specifically,

$$g_C(\hat{C})_{[\tau]} \oplus g_D(\hat{D})_{[\tau]} \oplus c_{[\tau]} = g_A(\hat{A})_{[\tau]} \oplus g_B(\hat{B})_{[\tau]}.$$

This implies that if the algorithm iterates over $u = g_C(\hat{C})_{[\tau]} \oplus g_D(\hat{D})_{[\tau]} \oplus c_{[\tau]}$, then v_{AB} is searched in $\mathcal{T}_{CD}$ and the key is output.

Complexity. Since there are ℓ' shifted keystreams $z^{(j)}$, then the expected number of corresponding $u \in \mathbb{F}_2^{\tau}$ values is about ℓ' (assuming $\ell' < 2^{\tau}$) and hence the algorithm is expected to recover the key in about $2^{\tau}/\ell'$ iterations.

In terms of time complexity, computing the masks in Step 1 by naive Gaussian elimination requires time complexity of roughly ℓ^3 bit operations. Naively applying the masks to the outputs of all states of each register and building the tables $\mathcal{T}_A, \mathcal{T}_B, \mathcal{T}_C, \mathcal{T}_D$ requires about $64 \cdot (2^{29} + 2^{31} + 2^{32} + 2^{33}) \cdot \ell \leq 2^{40} \cdot \ell$ bit operations.

Since for GEA-2 we have $\ell \leq 12800 < 2^{14}$, then the linear algebra complexity is upper bounded by roughly $2^{14 \cdot 3} + 2^{54} \approx 2^{54}$ bit operations, which is 2^{47} operations on 128-bit words (an upper bound on the complexity in GEA-2 evaluations).

Choosing $\tau = 30$ as in the basic attack, the complexity of each iteration remains about 2^{33} and their total complexity is

$$2^{33} \cdot 2^{30}/\ell' = 2^{64}/(\ell - 62).$$

Since $\ell \leq 12800$, this term dominates the complexity of the attack.

The memory complexity of the attack is calculated as follows: the matrix Z requires about ℓ^2 bits of storage, but this will be negligible. The hash tables $\mathcal{T}_A$ and $\mathcal{T}_D$ require about $2^{29} + 2^{31}$ words of 96 bits. The hash table $\mathcal{T}_{CD}$ requires memory of about 2^{32} words of 64 bits. Altogether, the hash tables require memory of about 64 GiB. The sequential tables $\mathcal{T}_B, \mathcal{T}_C$ require storage of $2^{32} + 2^{33} = 3 \cdot 2^{32}$ words of 64 bits or 96 GiB.

Note that this attack does not exploit any special property of the internal GEA-2 shift registers, and is thus applicable to any construction that combines the outputs of 4 independent stream ciphers by a simple XOR operation. In Appendix B we describe how to optimize the linear algebra of this attack, which may become a bottleneck in case the length of the keystream available to the attacker is not limited as in the case of GEA-2.

Recomputation of masked outputs. It is possible to eliminate the sequential tables and recompute the masked outputs for B and C on-the-fly at a modest penalty in time complexity. For GEA-2, this can be done (for example), with the fast polynomial evaluation algorithm of [8] (as also used in [2]), exploiting the low degree representation of the output of its registers.

4.4 Attacks Targeting the GEA-2 Initialization

We consider attacks that target the GEA-2 initialization process. Although this process does not have a significant weakness as in GEA-1, it linearly maps a 97-bit seed to a 125-bit internal state. Therefore, this state resides in a 97-dimensional linear subspace. Our previous attacks (and the ones of [2]) do not exploit this property and it is interesting to investigate whether it leads to improved attacks. On the other hand, we note that attacks which target the initialization process cannot benefit from the optimization that allows targeting multiple states using a consecutive keystream. While such attacks can target multiple initial states obtained by different GEA-2 frames using similar ideas, this requires more data and is therefore less practical.

Our conclusion is that at this point attacks targeting the initialization do not generally improve attacks that use a consecutive keystream (although they provide interesting time-memory tradeoffs even given a consecutive keystream). Yet, such attacks do offer improvements in case the keystream is fragmented and simultaneously targeting multiple states does not seem possible.

Exploiting the GEA-2 Initialization. Our goal is to exploit the fact that the state obtained after initialization resides in a 97-dimensional linear subspace to optimize attacks of GEA-2. This seems difficult at first, as the linear relations among the registers are complex and each register (and pair of registers) can attain all possible values. However, a careful examination will allow optimizations, as described next.

Note that any valid state obtained after initialization must satisfy $125 - 97 = 28$ linear equations (masks). Denote these masks by $m^{(1)}, \ldots, m^{(28)}$, where $m^{(i)} \in \mathbb{F}_2^{125}$ for $i \in [28]$. Let $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$ be a state obtained after initialization. Then, for all $i \in [28]$, $m^{(i)} \cdot (\hat{A}, \hat{B}, \hat{C}, \hat{D}) = 0$.

Suppose we wish to eliminate $g \leq 28$ variables from each register of an unknown state $(\hat{A}, \hat{B}, \hat{C}, \hat{D})$. Consider $m^{(1)}, \ldots, m^{(g)}$ and for each $i \in [g]$, guess the 3 bits

$$m^{(i)}_{[A]} \cdot \hat{A},\ m^{(i)}_{[B]} \cdot \hat{B},\ m^{(i)}_{[C]} \cdot \hat{C}.$$

This immediately gives

$$m^{(i)}_{[D]} \cdot \hat{D} = m^{(i)}_{[A]} \cdot \hat{A} \oplus m^{(i)}_{[B]} \cdot \hat{B} \oplus m^{(i)}_{[C]} \cdot \hat{C}.$$

Therefore, we have g linear equations per register ($4g$ in total), and by guessing the values of $3g$ of them we reduce the dimension of the subspace spanned by any register by g. We can thus symbolically represent the value of any b-bit register with only $b - g$ variables, which has an identical effect to guessing g variables per register. Overall, we have eliminated $4g$ variables at the cost of guessing $3g$ bits.

Hybrid attack and combination with exhaustive search. We consider a hybrid attack that exploits the optimized guessing strategy above. Set $g = 14$, reducing the dimension of the subspace spanned by each register by 14. Furthermore, guess another arbitrary variable for registers B and C. The total number of guesses is therefore $3 \cdot 14 + 2 = 44$.

After the guesses, the number of non-constant monomials in a polynomial representation of the output is upper bounded by

$$\sum_{i=1}^{4} \binom{29-14}{i} + \binom{31-14}{i} + \binom{32-15}{i} + \binom{33-15}{i} = 12411,$$

which is sufficient to mount the attack. After some optimizations, we were able to reduce the work performed per guess to be roughly equivalent to Gaussian elimination on a $2^{10} \times 2^{10}$ matrix, or 2^{30} bit operations. This implies that the total complexity is about $2^{44+30} = 2^{74}$ bit operations. While this may be faster than exhaustive search, the advantage is not considerable. We note, however, that the memory complexity of the attack is small and its time complexity decreases sharply with the keystream size.

A more efficient attack for the relevant parameters eliminates the contributions of the 3 smaller registers A, B, D, which is possible using $g = 13$ (i.e. we guess the values of $3 \cdot 13 = 39$ mask applications to these registers), as the number of non-constant monomials in the variables of these registers is upper bounded by

$$\sum_{i=1}^{4} \binom{29-13}{i} + \binom{31-13}{i} + \binom{32-13}{i} = 11598.$$

Then, we perform exhaustive search on $\hat{C}$ (using a fast polynomial evaluation algorithm), where the remaining search space is reduced to $33 - 13 = 20$ bits by exploiting the known linear equation values. This attack is competitive with the (optimized) one we describe below, as it has somewhat higher time complexity but reduced memory complexity.

It is an interesting open problem to improve these attacks further.

Attack G2-2 – hybrid with meet-in-the-middle. We now show how to use the guessing strategy to improve the memory complexity of the hybrid with meet-in-the-middle attack of [2] with no penalty in time complexity. This results in the most efficient attack on GEA-2 given a fragmented keystream.

Recall that the goal in this attack is to eliminate the contributions of the two registers A and D from the keystream, and then perform a meet-in-the-middle attack on registers B and C.

Consider $m^{(1)}, \ldots, m^{(9)}$ as defined in the guessing strategy. For each $i \in [9]$ guess the 2 bits

$$m^{(i)}_{[A]} \cdot \hat{A},\ m^{(i)}_{[D]} \cdot \hat{D}.$$

Moreover, guess additional 2 arbitrary bits of $\hat{A}$. This has an identical effect to guessing 11 bits of $\hat{A}$ and 9 bits of $\hat{D}$, and now the attack of [2] described above (without exploiting shifted keystreams) is directly applicable.

Optimizing memory complexity using additional linear equations. For each $i \in [9]$, we have

$$m^{(i)}_{[B]} \cdot \hat{B} \oplus m^{(i)}_{[C]} \cdot \hat{C} = m^{(i)}_{[A]} \cdot \hat{A} \oplus m^{(i)}_{[D]} \cdot \hat{D}, \tag{6}$$

where the right hand side is known. These 9 linear equations reduce the dimension of the subspace of states $(\hat{B}, \hat{C})$ relevant to the MITM attack. The main observation is that we can exploit the reduced dimension of this subspace to save memory by decomposing it, similarly to the attacks on GEA-1.

Let

$$U^{(B)} = \{(x,0) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{33}\} \text{ and } U^{(C)} = \{(0,x) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{33}\}.$$

In addition, define

$$V^{(BC)} = \{(x,y) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{33} \mid \forall i \in [9] : m_{[B]}^{(i)} \cdot x \oplus m_{[C]}^{(i)} \cdot y = 0\}.$$

The states relevant for the attack form an affine subspace $w \oplus V^{(BC)}$, where $w \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{33}$ depends on the guesses on the right hand side of (6).

We have $\dim(V^{(BC)}) = 65 - 9 = 56$. Moreover, as all relevant subspaces are in a 65-dimensional subspace,

$$\dim(V^{(BC)} \cap U^{(B)}) \geq 56+32-65 = 23 \text{ and } \dim(V^{(BC)} \cap U^{(C)}) \geq 56+33-65 = 24$$

(we chose the masks so both hold with equality, as verified by our program in Appendix C.3). Since $\dim(U^{(B)} \cap U^{(C)}) = 0$, similarly to the attacks on GEA-1, we can decompose the 56-dimensional subspace $V^{(BC)}$ as a direct sum

$$V^{(BC)} = V^{(1)} \boxplus V^{(2)} \boxplus V^{(3)},$$

where $\dim(V^{(1)}) = 9$, $\dim(V^{(2)}) = 23$, $\dim(V^{(3)}) = 24$, such that for any $(\hat{B}, \hat{C}) \in V^{(BC)}$, we have $\hat{B} = (v^{(1)} \oplus v^{(2)})_{[B]}$ and $\hat{C} = (v^{(1)} \oplus v^{(3)})_{[C]}$.

By considering the affine subspace $w \oplus V^{(BC)}$, similarly to the attacks on GEA-1, this decomposition allows to reduce the memory complexity by a factor of $2^{\dim(V^{(1)})} = 2^9$ to about 2^{23} words (it still dominates the memory complexity of the attack).

Interestingly, our advantage in terms of memory complexity *increases* as the number of available keystream bits decreases. This is because more variables are guessed, implying that $\dim(V^{(1)})$ and $2^{\dim(V^{(1)})}$ (which is the advantage factor in memory complexity) increase. For example, given 11300 bits of fragmented keystream, the memory complexity is reduced by a factor of 2^{10}.

Adapting the MITM attack of [2]. We point out two issues regarding our adaptation of the MITM attack of [2].

First, in [2], the 64 polynomial expressions of outputs of registers B and C were computed by multiplying large matrices. Our attack interpolates these polynomials directly in order to save memory. Specifically, we evaluate these $64+64 = 128$ polynomials on all inputs of Hamming weight at most 4 by computing the register outputs and applying the corresponding 128 linear expressions. After obtaining the evaluations, we interpolate the polynomials using the Möbius transform. The required memory is proportional to the size of the polynomials, namely $64 \cdot \left(\sum_{i=0}^{4} \binom{32}{i} + \binom{33}{i}\right) < 2^{23}$ bits, which is negligible.

Second, as noted above, the MITM attack of [2] is not standard since it is performed on complex functions of $f_B(\hat{B})$ and $f_C(\hat{C})$. and the attack uses a fast polynomial evaluation algorithm (such as the one of [8]) to evaluate these functions efficiently on all inputs. However, this does not pose a technical obstacle for our optimization, as the polynomial evaluation algorithm of [8] evaluates the polynomials using a Gray code. Therefore, we order the variables of $\hat{B}$ and $\hat{C}$ in all the polynomials with their (at least) 9 common variables as most significant bits. This allows to interleave the polynomial evaluations with the smaller MITM attacks (each independent MITM attack is performed per value of the most significant bits).

Implementation. We implemented the attack in `SageMath`, assuming 11300 keystream bits are available. We executed several iterations, each with a different guess for the linear expressions described above. An iteration took about 50 minutes to execute on a laptop using a single thread. While our implementation can be significantly optimized, its main purpose was to verify correctness by checking that the attack indeed returns the correct state for the correct guess.

Acknowledgements. This work was supported by the Israel Science Foundation grant no. 1903/20 and by the European Research Council under the ERC starting grant agreement no. 757731 (LightCrypt).

References

1. Aoki, K., Sasaki, Y.: Meet-in-the-Middle Preimage Attacks Against Reduced SHA-0 and SHA-1. In: Halevi, S. (ed.) Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5677, pp. 70–89. Springer (2009)
2. Beierle, C., Derbez, P., Leander, G., Leurent, G., Raddum, H., Rotella, Y., Rupprecht, D., Stennes, L.: Cryptanalysis of the GPRS Encryption Algorithms GEA-1 and GEA-2. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12697, pp. 155–183. Springer (2021)
3. Beierle, C., Felke, P., Leander, G.: To Shift or Not to Shift: Understanding GEA-1. IACR Cryptol. ePrint Arch. p. 829 (2021), https://eprint.iacr.org/2021/829
4. Bernstein, D.J.: Better price-performance ratios for generalized birthday attacks (2007), https://cr.yp.to/rumba20/genbday-20070904.pdf
5. Biryukov, A., Shamir, A.: Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers. In: Okamoto, T. (ed.) Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1976, pp. 1–13. Springer (2000)
6. Biryukov, A., Shamir, A., Wagner, D.A.: Real Time Cryptanalysis of A5/1 on a PC. In: Schneier, B. (ed.) Fast Software Encryption, 7th International Workshop, FSE 2000, New York, NY, USA, April 10-12, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1978, pp. 1–18. Springer (2000)

7. Bogdanov, A., Rechberger, C.: A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) Selected Areas in Cryptography - 17th International Workshop, SAC 2010, Waterloo, Ontario, Canada, August 12-13, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6544, pp. 229–240. Springer (2010)
8. Bouillaguet, C., Chen, H., Cheng, C., Chou, T., Niederhagen, R., Shamir, A., Yang, B.: Fast Exhaustive Search for Polynomial Systems in F_2. In: Mangard, S., Standaert, F. (eds.) Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6225, pp. 203–218. Springer (2010)
9. Bouillaguet, C., Delaplace, C., Fouque, P.: Revisiting and Improving Algorithms for the 3XOR Problem. IACR Trans. Symmetric Cryptol. 2018(1), 254–276 (2018)
10. Camion, P., Patarin, J.: The Knapsack Hash Function proposed at Crypto'89 can be broken. In: Advances in Cryptology - EUROCRYPT '91. Lecture Notes in Computer Science, vol. 547, pp. 39–53. Springer (1991)
11. ETSI: Digital cellular telecommunications system (Phase 2+) (GSM); 3GPP TS 24.008 version 16.7.0 Release 16: (2021), https://www.etsi.org/deliver/etsi_ts/124000_124099/124008/16.07.00_60/ts_124008v160700p.pdf
12. Howgrave-Graham, N., Joux, A.: New Generic Algorithms for Hard Knapsacks. In: Gilbert, H. (ed.) Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6110, pp. 235–256. Springer (2010)
13. Joux, A.: Algorithmic Cryptanalysis. Chapman & Hall/CRC (2009)
14. Knellwolf, S., Khovratovich, D.: New Preimage Attacks against Reduced SHA-1. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7417, pp. 367–383. Springer (2012)
15. Leurent, G., Sibleyras, F.: Low-Memory Attacks Against Two-Round Even-Mansour Using the 3-XOR Problem. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11693, pp. 210–235. Springer (2019)
16. Nikolic, I., Sasaki, Y.: Refinements of the k-tree algorithm for the generalized birthday problem. In: Tetsu Iwata and Jung Hee Cheon (ed.) Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9453, pp. 683–703. Springer (2015)
17. Pan, V.Y.: Structured Matrices and Polynomials: Unified Superfast Algorithms. Springer-Verlag, Berlin, Heidelberg (2001)
18. Schroeppel, R., Shamir, A.: A T=O($2^{n/2}$), S=O($2^{n/4}$) Algorithm for Certain NP-Complete Problems. SIAM J. Comput. 10(3), 456–464 (1981)
19. Wagner, D.A.: A Generalized Birthday Problem. In: Yung, M. (ed.) Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2442, pp. 288–303. Springer (2002)

20. Wang, J.R.: Space-Efficient Randomized Algorithms for K-SUM. In: Schulz, A.S., Wagner, D. (eds.) Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8737, pp. 810–829. Springer (2014)

A 4-XOR Algorithms

In this section, we give more details on 4-XOR algorithms.

A.1 Schroeppel-Shamir Algorithm Variant

The Schroeppel-Shamir algorithm variant is described below. Denote $\tau = n/4$.

```
1. – Initialize a table $\mathcal{T}_1$, storing elements in $\mathbb{F}_2^\tau$.
   – For all $x^{(1)} \in \mathbb{F}_2^\tau$, store $x^{(1)}$ at index $f_1(x^{(1)})$ in $\mathcal{T}_1$.
2. – Initialize a table $\mathcal{T}_3$, storing elements in $\mathbb{F}_2^\tau$.
   – For all $x^{(3)} \in \mathbb{F}_2^\tau$, store $x^{(3)}$ at index $f_3(x^{(3)})$ in $\mathcal{T}_3$.
3. For all $u \in \mathbb{F}_2^\tau$:
   (a) – Initialize a table $\mathcal{T}_{1,2}$, storing pairs in $\mathbb{F}_2^\tau \times \mathbb{F}_2^\tau$.
       – For all $x^{(2)} \in \mathbb{F}_2^\tau$, search $\mathcal{T}_1$ for $f_2(x^{(2)})_{[\tau]} \oplus u \oplus t_{[\tau]}$. For each
         match $x^{(1)}$:
           • Let $v_{1,2} = f_1(x^{(1)})_{[\tau+1,n]} \oplus f_2(x^{(2)})_{[\tau+1,n]} \oplus t_{[\tau+1,n]}$. Store
             $(x^{(1)}, x^{(2)})$ at index $v_{1,2}$ in $\mathcal{T}_{1,2}$.
   (b) For all $x^{(4)} \in \mathbb{F}_2^\tau$, search $\mathcal{T}_3$ for $f_4(x^{(4)})_{[\tau]} \oplus u$. For each match $x^{(3)}$:
       – Let $v_{3,4} = f_3(x^{(3)})_{[\tau+1,n]} \oplus f_4(x^{(4)})_{[\tau+1,n]}$. Search $v_{3,4}$ in $\mathcal{T}_{1,2}$.
       – If a match $(x^{(1)}, x^{(2)})$ exists, output $(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)})$ as a
         solution to 4-XOR.
```

Analysis.

Correctness. If we fix $(x^{(1)}, x^{(2)})$, then for

$$u = f_1(x^{(1)})_{[\tau]} \oplus f_2(x^{(2)})_{[\tau]} \oplus t_{[\tau]},$$

$f_2(x^{(2)})_{[\tau]} \oplus u \oplus t_{[\tau]} = f_1(x^{(1)})_{[\tau]}$ is searched in $\mathcal{T}_1$ and $x^{(1)}$ is retrieved. Therefore, $(x^{(1)}, x^{(2)})$ is stored at index $v_{1,2}$ in $\mathcal{T}_{1,2}$.

If $(x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)})$ is a solution to 4-XOR, then

$$u = f_3(x^{(3)})_{[\tau]} \oplus f_4(x^{(4)})_{[\tau]}$$

holds as well. Similarly, $v_{3,4} = f_3(x^{(3)})_{[\tau+1,n]} \oplus f_4(x^{(4)})_{[\tau+1,n]}$ is searched in $\mathcal{T}_{1,2}$ and the solution is output. This implies that the algorithm does not miss a solution if it exists.

Complexity. Each of the first two step takes $O(2^{\tau})$ time. For each $u \in \mathbb{F}_2^{\tau}$, in Step 3.(a) the expected number of matches in $\mathcal{T}_1$ is $2^{\tau} \cdot 2^{\tau} \cdot 2^{-\tau} = 2^{\tau}$ (as we match on τ bits), which gives the size of $\mathcal{T}_{1,2}$. Thus, the total complexity of this step (over all iterations) is $O(2^{\tau} \cdot 2^{\tau}) = O(2^{2\tau}) = O(2^{n/2})$. A similar calculation applies to Step 3.(b). The total time complexity is therefore $O(2^{n/2})$, while the memory complexity is $O(2^{\tau}) = O(2^{n/4})$, which is the memory complexity of each table built.

A.2 Wagner's Algorithm for 4-XOR and Extensions

Wagner's k-tree algorithm [19] improves the time complexity when the domains of the functions are large.[8] Specifically, for $k = 4$, when $f_1, f_2, f_3, f_4 : \mathbb{F}_2^{n/3} \to \mathbb{F}_2^{n}$, the number of expected solutions is $2^{4 \cdot n/3} \cdot 2^{-n} = 2^{n/3}$ and the k-tree algorithm finds one of them in time and memory complexities of $O(2^{n/3})$.

The algorithm is very similar to the Schroeppel-Shamir variant described above, but it uses $\tau = n/3$ instead of $\tau = n/4$. This increases the complexity of each iteration for $u \in \mathbb{F}_2^{\tau}$ to $O(2^{n/3})$. However, the main observation is that with good probability, we only need to iterate over one value of u. In particular, fixing u, we expect a single 4-XOR solution to satisfy the additional $(n/3)$-bit condition $u = f_1(x^{(1)})_{[n/3]} \oplus f_2(x^{(2)})_{[n/3]} \oplus t_{[n/3]}$.

In the general case where $f_1, f_2, f_3, f_4 : \mathbb{F}_2^{\kappa} \to \mathbb{F}_2^{n}$ for $n/4 \leq \kappa \leq n/3$, a hybrid algorithm was devised in [12]. Its time complexity is $O(2^{n-2\kappa})$, while its memory complexity is $O(2^{\kappa})$. The algorithm remains similar and uses $\tau = \kappa$. Thus, the complexity for each $u \in \mathbb{F}_2^{\kappa}$ is $O(2^{\kappa})$, and we need to iterate over $O(2^{n-3\kappa})$ values of u to obtain a solution with high probability.

B Optimizations of Attack G2-1 for the XOR Combiner

Since Attack G2-1 of Section 4.3 is applicable to a general XOR combiner of 4 stream ciphers, it would be useful to expand on how to optimize it on the general XOR combiner in case the length of the keystream available to the attacker is not limited as in the case of GEA-2. Indeed, while the linear algebra complexity is negligible for the attack on GEA-2 due to the short keystream, it does become a bottleneck for longer keystreams.

First, consider the first step in which the masks are deduced by solving a linear equation system. Note that the matrix Z obtained before XORing $z^{(\ell')}$ to each row is a Toeplitz matrix, as it is composed of shifted keysteams. There are known algorithms for solving such Toeplitz systems that are much faster than Gaussian elimination (see [17]).

Second, consider the application of the masks to the outputs of all states of each register for the purpose of building the tables (Step 2 and Step 3). Fix a mask

[8] The basic version for $k = 4$ was already described by Camion and Patarin [10].

m of length λ and a starting state $\hat{A}$. Clock $\hat{A}$ $2\lambda - 1$ times to produce λ shifted keystreams of length λ. These shifted keystreams (once again) form a Toeplitz matrix, and we need to compute its product with the vector m. There are efficient algorithms known for this purpose (based on the fast Fourier transform) which are much faster than a naive matrix-vector product computation.

C SageMath Source Code for Linear Algebra Computations

C.1 Auxiliary functions

```
#————Returns the initializing matrix for a register with Galois
    polynomial p (taken from [2])————
def getInitMatrix(p ,seedLength ,shift):
     P.<x> = PolynomialRing (GF(2))
     l = p.degree()
     #Construct transformation matrix A for LFSR in Galois mode
     A = companion_matrix(p , 'left')
     M = zero_matrix(GF(2), seedLength, l)
     for c in range (seedLength):
         x = zero_vector(GF(2), l)
         k = zero_vector(GF(2), seedLength)
         k[c] = 1
         for j in range (seedLength):
             x[0] = x[0] + k[(j+shift) % seedLength]
             x = A*x
         M[c] = x
     return M


#————Returns the projection of the space 'space' onto the bits '
    listOfBits'————
def project(space, listOfBits):
     l = len(listOfBits)
     d = space.degree()
     #Construct projection matrix Pr
     Pr = zero_matrix(GF(2), l, d)
     for i in range (l):
         x = zero_vector(GF(2), d)
         x[listOfBits[i]] = 1
         Pr[i] = x
     return (Pr.transpose()).restrict_domain(space).image()


#————Returns the lift of the space 'space' to a space of degree '
    degree'————
#————The original space in the columns of 'listOfbits', the rest
    fill with zeroes
def lift(space, degree, listOfBits):
     l = len(listOfBits)
     #Construct lift matrix Lift
     Lift = zero_matrix(GF(2), l, degree)
     for i in range (l):
         x = zero_vector(GF(2), degree)
         x[listOfBits[i]] = 1
         Lift[i] = x
     #Returns the image of the space under the lift matrix
     return Lift.restrict_domain(space).image()


#————Returns the lift of the vector 'vector' to a space of degree '
    degree'————
```

```
#————The original vector in the columns of 'listOfbits', the rest
     fill with zeroes
def liftVector(vector, degree, listOfBits):
      l = len(listOfBits)
      #Construct lift matrix Lift
      Lift = zero_matrix(GF(2), l, degree)
      for i in range (l):
          x = zero_vector(GF(2), degree)
          x[listOfBits[i]] = 1
          Lift[i] = x
      #Returns the image of the vector under the lift matrix
      return vector*Lift


#————Returns the list of the corresponding unit vectors————
def listOfUnitVectors(space, listOfBits):
     L = []
     for i in range (len(listOfBits)):
         x = zero_vector(GF(2), space.degree())
         x[listOfBits[i]]=1
         L.append(x)
     return L


#————Merge two matrices with the same number of rows, to one matrix
     ————
#————The columns of Mat1 will be placed in the columns from
     listOfBits1
#————The columns of Mat2 will be placed in the columns from
     listOfBits2
def mergeMat (Mat1, listOfBits1, Mat2, listOfBits2):
     l1 = len(listOfBits1)
     l2 = len(listOfBits2)
     n = Mat1.nrows()
     M = zero_matrix(GF(2), n, (l1+l2))
     for i in range (n):
         x = liftVector(Mat1[i], (l1+l2), listOfBits1) + liftVector(Mat2[
             i], (l1+l2), listOfBits2)
         M[i] = x
     return M
```

C.2 Attack G1

```
#————The data (Galois polynomials, seed length, shifting) for GEA
     -1————
P.<x> = PolynomialRing(GF(2))
seedLength_1 = 64
pA = x^31+x^30+x^28+x^27+x^23+x^22+x^21+x^19+x^18+x^15+x^11+x^10+x^8+x
     ^7+x^6+x^4+x^3+x^2+1
shiftA_1 = 0
pB = x^32+x^31+x^29+x^25+x^19+x^18+x^17+x^16+x^9+x^8+x^7+x^3+x^2+x+1
shiftB_1 = 16
pC = x^33+x^30+x^27+x^23+x^21+x^20+x^19+x^18+x^17+x^15+x^14+x^11+x^10+x
     ^9+x^4+x^2+1
shiftC_1 = 32

#————The initialization matrices for the registers of GEA
     -1————
MA1 = getInitMatrix(pA, seedLength_1, shiftA_1)
MB1 = getInitMatrix(pB, seedLength_1, shiftB_1)
MC1 = getInitMatrix(pC, seedLength_1, shiftC_1)
M1 = block_matrix([MA1, MB1, MC1], nrows=1)
MAC1 = block_matrix([MA1, MC1], nrows=1)

#————Construct the spaces V1, V2, V3————
```

```
#dim 24
V2 = M1.restrict_domain(MAC1.kernel()).image();
#dim 32
V3 = M1.restrict_domain(MB1.kernel()).image();
U1, pi1, lift1 = M1.image().quotient_abstract(V2+V3)
#dim 8
V1 = lift1(U1)


#————Construct the spaces V3_AC, V3_JAJC, VA, VC, V4————
AC = [i for i in range(31)] + [i for i in range(63,96)]
V3_AC = project(V3, AC)
JA = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 21, 22,
    23, 24, 25, 26, 27, 28, 29]
JC = [63, 64, 65, 66, 67, 68, 69, 70, 71, 73, 74, 75, 76, 77, 82, 83,
    84, 85, 86, 93, 94, 95]
JAJC = JA+JC
#dim 32
V3_JAJC = project(V3, JAJC)
V = VectorSpace(GF(2), 48)
U_JA = V.subspace(listOfUnitVectors(V, range(26)))
U_JC = V.subspace(listOfUnitVectors(V, range(26,48)))
#dim 10
VA = V3_JAJC.intersection(U_JA)
#dim 6
VC = V3_JAJC.intersection(U_JC)
U2, pi2, lift2 = V3_JAJC.quotient_abstract(VA+VC)
#dim 16
V4 = lift2(U2)


#————Uniquely extend the subspaces of V3_JAJC to subspaces of V3_AC
    ————
JClow = [JC[i]-32 for i in range(len(JC))]
JAJClow = JA+JClow
JS = list(set(i for i in range(64))-set(JAJClow))
JS.sort()

#—Compute matrices
V3_ACker = V3_AC.basis_matrix().right_kernel()
V3_JAJCkerIn64 = lift(V3_JAJC.basis_matrix().right_kernel(), 64, JAJClow
    )
U4, pi4, lift4 = V3_ACker.quotient_abstract(V3_JAJCkerIn64)
the16EqSpace = lift4(U4)
L = the16EqSpace.basis_matrix()[range(16), JS]
LInv = L^(-1)
D = the16EqSpace.basis_matrix()[range(16), JAJClow]

#—Construct VA_prime
CA = D * (VA.basis_matrix().transpose())
XA = (LInv * CA).transpose()
VA_primeBasisMat = mergeMat (VA.basis_matrix(), JAJClow, XA, JS)
#dim 10
VA_prime = span(VA_primeBasisMat)

#—Construct VC_prime
CC = D * (VC.basis_matrix().transpose())
XC = (LInv * CC).transpose()
VC_primeBasisMat = mergeMat (VC.basis_matrix(), JAJClow, XC, JS)
#dim 6
VC_prime = span(VC_primeBasisMat)

#—Construct V4_prime
C4 = D * (V4.basis_matrix().transpose())
X4 = (LInv * C4).transpose()
V4_primeBasisMat = mergeMat (V4.basis_matrix(), JAJClow, X4, JS)
#dim 16
V4_prime = span(V4_primeBasisMat)
```

C.3 Attack G2-2

```
#———The data (Galois polynomials, seed length, shifting) for GEA
    -2———
P.<x> = PolynomialRing(GF(2))
seedLength_2 = 97
pA = x^31+x^30+x^28+x^27+x^23+x^22+x^21+x^19+x^18+x^15+x^11+x^10+x^8+x
    ^7+x^6+x^4+x^3+x^2+1
shiftA_2 = 16
pB = x^32+x^31+x^29+x^25+x^19+x^18+x^17+x^16+x^9+x^8+x^7+x^3+x^2+x+1
shiftB_2 = 33
pC = x^33+x^30+x^27+x^23+x^21+x^20+x^19+x^18+x^17+x^15+x^14+x^11+x^10+x
    ^9+x^4+x^2+1
shiftC_2 = 51
pD = x^29+x^28+x^25+x^24+x^23+x^22+x^21+x^20+x^19+x^17+x^15+x^13+x^12+x
    ^9+x^8+x^6+x^3+x+1
shiftD_2 = 0


#———The initialization matrices for the registers of GEA-2———
MA2 = getInitMatrix(pA, seedLength_2, shiftA_2)
MB2 = getInitMatrix(pB, seedLength_2, shiftB_2)
MC2 = getInitMatrix(pC, seedLength_2, shiftC_2)
MD2 = getInitMatrix(pD, seedLength_2, shiftD_2)
M2 = block_matrix([MA2, MB2, MC2, MD2], nrows=1)

#———Construct the spaces V_BC, V_1, V_2, V_3———
#dim 56
V_BC = M2.right_kernel().basis_matrix()[range(9), range(31,96)].
    right_kernel()
U = VectorSpace(GF(2), 65)
U_B = U.subspace(listOfUnitVectors(U, range(32)))
U_C = U.subspace(listOfUnitVectors(U, range(32,65)))
#dim 23
V_2 = V_BC.intersection(U_B)
#dim 24
V_3 = V_BC.intersection(U_C)
U3, pi3, lift3 = V_BC.quotient_abstract(V_2+V_3)
#dim 9
V_1 = lift3(U3)
```