Hitag 2 Hell – Brutally Optimizing Guess-and-Determine Attacks

Aram Verstegen
Automotive Security Research Department
FactorIT B.V. – Arnhem, The Netherlands.

Roel Verdult
Automotive Security Research Department
FactorIT B.V. – Arnhem, The Netherlands.

Wouter Bokslag
Automotive Security Research Department
FactorIT B.V. – Arnhem, The Netherlands.

Abstract

Cryptographic guess-and-determine (GD) attacks are occasionally mentioned in the literature, but most articles describe conceptual attack optimization while implementation details are seldom discussed. Therefore, we present in this paper not only a conceptual attack optimization, but also a fully detailed design strategy to optimize a general bit-sliced exhaustive search implementation. To demonstrate the applicability of our contribution we present a highly optimized practical brute-force attack on the Hitag2 stream cipher using a guess-and-determine approach. Our implementation explores the full 48-bit search space on a consumer desktop PC with one GPU in approximately 1 minute. The work is specifically effective to recover secret keys from the widely deployed Hitag2 Remote Keyless Entry (RKE) system. Compared to the most practical Hitag2 RKE attack published in the literature, our implementation is more than 500 times faster. Furthermore, our approach has a 100% success rate with only two captured RF frames and is extremely practical compared to previously published unrealistic sat-solver, cube cryptanalysis and correlation attacks which require hundreds of traces or truly random nonces. We fully release our source code as reference material for related research in the future.

1 Introduction

Despite well-known historical recommendations in the literature [22, 1, 14], several widely deployed bit-wise stream cipher based crypto systems [10, 15, 35, 25, 11, 8, 33] do not use their complete internal state to compute a keystream bit. This weakness exposes a cipher to a guess-and-determine (GD) attack. During such an attack the adversary initially *guesses* only a partial internal cipher state and evaluates the produced output bit against the observed keystream bit to *determine* if the guess was correct. As each cipher step halves (on average) the set of all possible internal state candidates, by propagating this information over several cipher steps an incremental approach can be found to reduce the overall computational attack complexity.

Contribution Our contribution in this paper is threefold. Firstly, we explore some conceptual optimization strategies to improve on the straight-forward guess-and-determine approach. Secondly, we discuss attack implementation limitations and their impact on performance. Finally, we verify our optimization efficiency by benchmarking the execution performance on consumer-grade hardware.

We present our initial solution as generally as possible, after which we explore several strategies to optimize the algorithm's implementation for modern consumer desktop machines. In each iteration our solution remains favorable to parallel computation and scales linearly over execution units. We demonstrate how these general algorithmic optimizations are applicable to accelerate cryptanalytic attacks.

Usability To demonstrate the usability and efficiency of our optimizations we apply them on a practical use-case. For this we selected from the literature a recently proposed cryptographic attack that performs a computationally hard exhaustive search. The cryptographic attack we choose to implement is described in the paper by Benadjila et. al. [2], published at *Usenix WOOT 2017*, which mounts a key recovery attack against the Hitag2 Remote Keyless Entry (RKE) system. While their article gives an excellent overview of the problem and cryptanalytic context, the proposed exhaustive search design is based on a trivial brute-force approach and does not seem to contain any implementation optimization at all. We estimate our optimized guess-and-determine attack to be over 500 times faster. After publication we will set up a public repository where we will release our complete

source code under the General Public License (GPL) to enable reuse of our methods by fellow researchers.

Relevance We performed the attack in practice on actual vehicles and tested various models (as mentioned in earlier work [12]) produced in 2017 from different makers that use the Hitag2 cipher and RKE protocol. We verified that our attack implementation explores all possible candidates and recovers a working 48-bit internal state in approximately 1 minute that can be used to forge an RKE frame that can open the doors of the vehicle.

Optimizations As *conceptual optimizations* we propose to apply memoization of non-linear subfunctions and a precomputation that avoids guessed valuations we know to be impossible. To significantly improve the execution performance we introduce *implementation optimizations* to unroll recursion as nested loops, precompute the non-linear function and internal state guesses through lookup tables and apply bit-slicing to parallelize this computationally hard problem. In the *performance benchmark* we compare our performance results against the reference paper [2] and show that our method and implementation is significantly faster than previous work.

Overview In this paper we address the related work in Section 2. Then, we explain the attack context and limitations in Section 3. Next, we describe our implementations and optimizations in Section 4. This is followed by a performance evaluation in Section 5. Finally, we present our conclusion in Section 6.

2 Related work

There are several examples in the literature [18, 19, 13, 36, 4, 26, 9] that successfully mount guess-and-determine attacks on various cryptographic algorithms. However, these articles lack specific implementation details and ignore the feasibility of mounting such attacks in a modern parallelized execution environment such as multi-core CPUs and general purpose GPU computation platforms like OpenCL.

During the last decade several attacks on Hitag2 were published in the literature. An overview of these attacks is presented in Table 1.

Earlier attacks on Hitag2 mainly focused on attacking the immobilizer authentication protocol. Those attacks universally require control or at least knowledge of the nonce, and preferably a truly random nonce. With knowledge of all nonce bits the complete secret key can be solved linearly by inverting the initialization of the cipher's internal state.

Attack	Description	Frames	Time	Practical
[30]	cube	500	N/A	no[1]
[32]	cryptanalytic	136	N/A	no[2]
[12]	correlation	4-8	1-10 min	no[3]
[29]	sat-solver	N/A	N/A	no[4]
[7]	sat-solver	4	2 days	**yes**
[34]	brute-force	2	4 years	**yes**
[2]	brute-force	2	18 hours	**yes**
[20]	brute-force	2	11 hours	**yes**
this paper	guess-and-determine	2	75 seconds	**yes**

[1] Sun et al. require control over the nonce
[2] Verdult et al. exploits a cipher property that requires random nonces
[3] Garcia et al. requires significant difference in nonce values
[4] Soos et al. require 50 bits of contiguous keystream.

Table 1: Comparison of published Hitag2 RKE attacks.

In the RKE context there is only knowledge of a partial nonce, thus the cipher's internal state must be recovered before resolving the actual key. Benadjila et al. [2] demonstrate that up to now this could only be done through a 'naive' brute-force solution to recover the cipher's internal state before attacking the key. Our attack uses the same restrictions but performs significantly better.

3 Hitag2 cipher and RKE protocol

Since we build on earlier work attacking Hitag2, we assume the reader is at least somewhat familiar with the Hitag2 cipher construct, as well as the unidirectional Hitag2 RKE protocol that has already been explained well in previous publications [2, 12]. In this section we will briefly summarize the two main relevant concepts.

A visualization of the state, LFSR feedback function and filter function f that make up the Hitag2 cipher is shown in Figure 1. The Hitag2 cipher uses a non-linear filter function, which was a popular design choice during the 80's because of its hardware friendly implementation properties [16, 17, 21, 27, 22, 1]. The initial state of this cipher is determined by a shared 48-bit secret key, a 32-bit transmitter UID and a 32-bit nonce. After initialization the lower 16 bits of the cipher state correspond to the highest 16 key bits, while the other 32 state bits represent the encrypted 32-bit nonce.

3.1 Guess-and-determine on Hitag2

We assume familiarity with the concept of a guess-and-determine attack, as the method has been covered extensively in the literature [31, 28].

A guess-and-determine attack on Hitag2 targets the filter function f, which takes 20 bits input from the secret 48-bit internal state in order to produce a single keystream bit. The filter function is given by Definition 3.1.

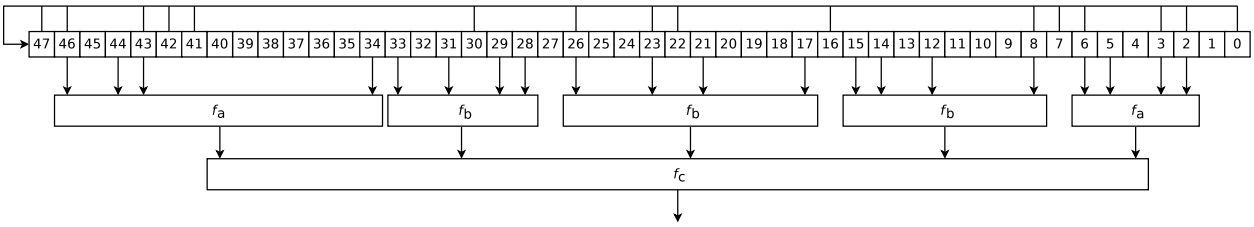

Figure 1: The Hitag2 state, LFSR and two-step filter function (f).

Definition 3.1.

$$\begin{aligned}
f_a(a,b,c,d) =& \neg(((a \vee b) \wedge c) \oplus (a \vee d) \oplus b) \\
f_b(a,b,c,d) =& \neg(((d \vee c) \wedge (a \oplus b)) \oplus (d \vee a \vee b)) \\
f_c(a,b,c,d,e) =& \neg((((((c \oplus e) \vee d) \wedge a) \oplus b) \wedge (c \oplus b)) \\
& \oplus (((d \oplus e) \vee a) \wedge ((d \oplus b) \vee c))) \\
f(x) =& f_c(f_a(x_2,x_3,x_5,x_6), f_b(x_8,x_{12},x_{14},x_{15}), \\
& f_b(x_{17},x_{21},x_{23},x_{26}), f_b(x_{28},x_{29},x_{31},x_{33}), \\
& f_a(x_{34},x_{43},x_{44},x_{46}))
\end{aligned}$$

The attack is based on the inversion of the f function so that based on an observed keystream bit, we obtain a set of partial candidate states that would have resulted in the observed output bit. For each of these candidate states, we invoke the round function while guessing the LFSR feedback if not all taps are known. For each keystream bit we incrementally invoke the round function to recover more bits from the internal cipher state; we refer to such an iteration with the term layer. At each invocation of the round function, several of the input bits for f may overlap with some of the previously guessed bits. Therefore there should be on average fewer guesses to make on the state for each subsequent keystream bit tested.

This leads to an attack in multiple layers, where in each layer a specific subset of the filter input bits is determined until at the last layer the entire state is determined, which can then be tested against the remainder of the observed keystream. Until the state is fully determined, at each layer the space of potential states increases exponentially, proportional to the number of guessed bits. Since the output of f has a uniform distribution for random inputs, we expect half of the candidate states tested against a keystream bit can be discarded on average as the generated keystream bit will not match the observed keystream bit.

3.2 Hitag2 RKE

There are some fundamental differences between Hitag2 in the immobilizer context and Hitag2 in the Remote Keyless Entry (RKE) context. In this section, we will briefly explain the RKE protocol, and how information recovered by a guess-and-determine attack can be used against Hitag2 in the RKE context.

3.2.1 Frame format

When pressing a button on a Hitag2 RKE keyfob, a data frame (visualized in Figure 2) that contains an authenticated command to operate the car's door or trunk locks is communicated through a UHF radio transmitter.

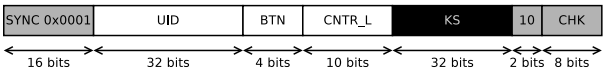

Figure 2: The Hitag2 RKE frame.

In this frame the field KS serves as a 32-bit MAC on the frame and is computed as a direct sample of keystream from the Hitag2 cipher after initialization. The 32-bit nonce used to initialize the cipher is unique to each transmission, but behaves sequentially. It is composed of a 28-bit counter and a four-bit button code. Next to the button code (BTN), the 10 least significant bits of this 28-bit counter are contained in the frame (CNTR_L). Thus, of the 32 nonce bits, all but the most significant 18 counter bits are known to an adversary. Because the button bits follow the counter in the nonce, the attacker can determine the 14 least significant bits of the nonce.

3.2.2 RKE Key recovery

Through an exhaustive search we can determine all (approximately 2^{16}) 48-bit cipher states that generate a given 32-bit Hitag2 keystream sample.

We can make use of the fact that bits from nonces in this protocol overlap to narrow down this list to the one real cipher state that generated it [1]. With a second sample of keystream observed from the same transmitter, and the observable nonce information used to generate both samples, we can roll back the encryption of the known lowest nonce bits during initialization of any such potential state to reach an earlier cipher state. From this pre-initialized state we can try to encrypt the observed nonce bits associated with the second sample to complete a second initialized state. If that state generates the second sample of keystream, we identified the correct initial states related to these two samples. As the least significant bits of the counter are contained in the known nonce

bits, we can just increment these counter bits and choose our own button code bits to predict the next initialized cipher state.

Once the real internal state is known, this idea can be generalized further by inverting the complete initialization while guessing all unknown nonce bits. This yields an *equivalent key* which will remain valid until all of the counter bits we can observe overflow into the internal state bits we have guessed so far. Such an equivalent key at any value of CNTR_L allows the adversary to forge the next $2^9 = 512$ values of KS on average. After this point, there are multiple candidates for an equivalent key, as cascading bit flips in the unobserved bits in the counter may have occurred from incrementing the counter to the point of overflowing the lowest 10 bits [2]. From any newly observed triplets of KS, BTN and CNTR_L, together with the information we used to compute a previous equivalent key, a new equivalent key can more efficiently be computed.

4 Implementations

The masks for Hitag2 filter input bits that need to be guessed in each layer can be generated using the function shown in Listing 1, which also determines when to stop guessing LFSR feedback bits. As shown in Table 2, this leads to an attack on Hitag2 in 9 layers, where we only need to guess one extra bit in the next layer's state (the LFSR feedback) on the first and second layer.

Each layer, more bits of the internal state are fixed by previous guesses. This process is visualized[3] in Figure 3. Although not strictly necessary, we decide to guess the LFSR feedback bit in layer 0 and 1, which gives us the advantage that we can compute LFSR feedback from layer 2 and onwards. As these bits are shifted to the right after applying the round function, they determine bit 47 in layer 1 and layer 2 respectively.

```
function generate_masks(filt_mask=0x5806b4a2d16c,
                        lfsr_mask=0xce0044c101cd):
  masks = []
  fill = last_lfsr_guess = 0
  while fill != 0xffffffffffff:
    masks.append(filt_mask) # save taps mask
    fill |= filt_mask # track known bits
    # Check if LFSR feedback should be guessed
    if (fill & lfsr_mask) != lfsr_mask:
      last_lfsr_guess += 1
    fill >>= 1 # update LFSR state
    fill |= (1<<47) # guess or get LFSR feedback
    # Take out known bits from next mask
    filt_mask &= ~fill
  return masks, last_lfsr_guess
```

Listing 1: Guessed bits mask generation.

Layer	Mask	Bits	Guess LFSR feedback
0	0x5806b4a2d16c	20	yes
1	0x5004a4a29148	14	yes
2	0x000400820100	4	no
3	0x000400020100	3	no
4	0x000400000000	1	no
5	0x000400000000	1	no
6	0x000400000000	1	no
7	0x000400000000	1	no
8	0x000400000000	1	no

Table 2: Which bits to guess at each layer.

The algorithm finds approximately 2^{39} fully determined states which generate the first 9 bits of the keystream sample. These 48-bit states must then be tested against the remaining 23 observed keystream bits, which will narrow down the list of candidates to approximately 2^{16} states that produced the observed 32-bit sample [4]. For each such state that has generated 32 bits of keystream output the LFSR function can then be applied in reverse 32 times to get its initial state. Note that the algorithm can easily be parallelized by letting execution units commence their work at layer 1, while a controlling unit dispatches the 2^{19} successful outputs from layer 0 among them.

4.1 Naive recursive GD

We first define two functions that are used by the next implementations. A function *test* to check candidate states after they are fully determined is shown in Listing 3.

Optimization 4.1 (Precomputed f)**.** To get the results of f computations more quickly we can precompute its output for all 2^{20} possible inputs.

Furthermore, we define the function *expand* to deposit bits from a counter over bits in a mask value, shown in Listing 2.

```
function expand(mask, value, size=48):
  fill = 0
  for i in range(size):
    if mask & 1:
      fill |= (value & 1)<<i
      value >>= 1
    mask >>= 1
  return fill
```

Listing 2: The *expand* function.

Optimization 4.2 (Precomputed *expand*)**.** To get the result of *expand* more quickly we can precompute all possible mask valuations for the mask of each layer [5].

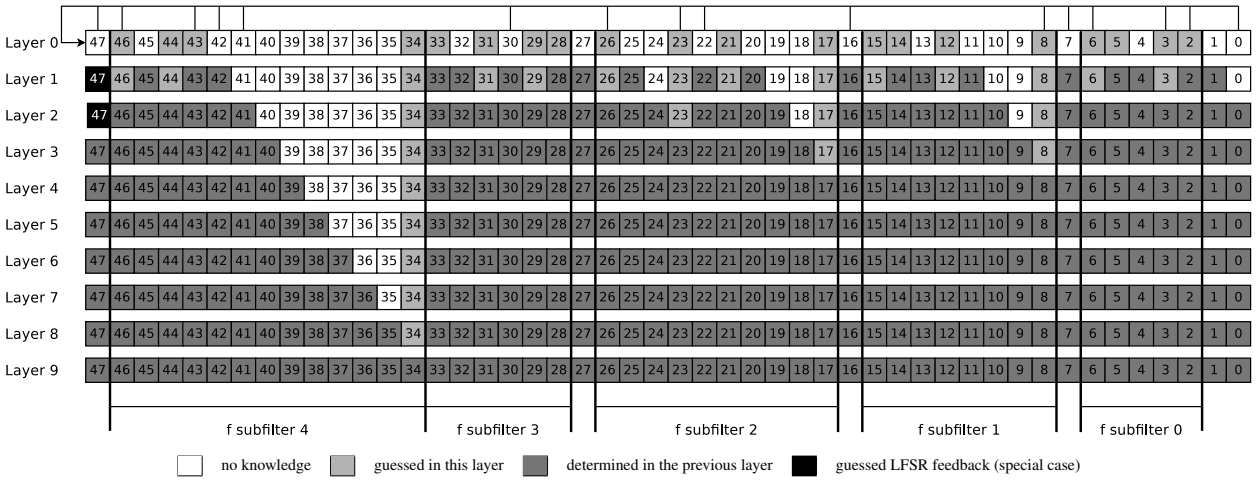

Figure 3: Visualization of bits guessed at each layer which eventually determine the Hitag2 internal cipher state.

```
function test(state):
  for bit in range(len(masks), 32):
    if f(state) != keystream[bit]:
      return
    state = lfsr(state)
  print state
```

Listing 3: The *test* function.

A straightforward implementation of an algorithm to use the masks from Table 2 is a recursive one, shown in Listing 4. It consists of the following steps:

1. While the state is still not fully determined:
   (a) Determine all the the layer's filter input bits by iterating a counter
   (b) Expand the counter value to the layer's mask using the *expand* function
   (c) Combine the expanded value with the previous layer's state
   (d) Compute and check the result of f against the observed keystream bit for this layer
   (e) If the guess was correct, iterate to the next layer with an updated (guessed or computed) LFSR feedback bit
2. When the state is fully determined, test it against the rest of the observed keystream using the *test* function

Some optimizations can immediately be applied to this naive implementation.

Optimization 4.3 (Recursion unrolling)**.** To remove recursion from the implementation we can unroll it by placing the recursive work in nested loops.

```
function fill_layer(state, layer):
  if layer < len(masks):
    for i in range(1<<bits[layer]):
      fill = expand(masks[layer], i)
      new_state = state | fill
      if f(new_state) != keystream[layer]:
        continue
      if layer < last_lfsr_guess:
        fill_layer(new_state>>1,
                   layer+1)
        fill_layer(new_state>>1 | (1<<47),
                   layer+1)
      else:
        fill_layer(lfsr(new_state),
                   layer+1)
  else:
    test(state)
fill_layer(0, 0)
```

Listing 4: Naive recursive solution.

4.2 GD avoiding impossible guesses

In the previous implementation half of the partially guessed states we generate and test are immediately discarded as they do not yield the observed keystream bit for their respective layer. Instead of simply precomputing *expand* and *f*, we can compute a lookup table per layer that perfectly complements the state at each layer to produce the next observed keystream bit [6].

Optimization 4.4 (Precomputed guesses based on the observed keystream)**.** Use the observed keystream bit to avoid generating and evaluating cipher states which don't produce this keystream bit. This effectively combines the work of *expand* and *f*, but only yields results that pass the keystream test. By taking the set filter input bits as an index into a hash table, we can quickly navigate to a

lookup table that contains only the guesses we need to create the next valid state, meaning we don't have to call the *expand* or f functions at all in the recursive function. These lookup tables can be computed as shown in Listing 5.

```
function generate_fills(filt_mask=0x5806b4a2d16c):
  layer_fills = [{} for _ in range(len(masks))]
  for i in range(1<<20):
    taps_fill = expand(filt_mask, i)
    out = f(taps_fill)
    for layer in range(len(masks)):
      if out != keystream[layer]:
        continue
      new = taps_fill & masks[layer]
      old = taps_fill & ~masks[layer]
      try:
        layer_fills[layer][old].append(new)
      except: # Table miss
        layer_fills[layer][old] = [new]
  return layer_fills
```

Listing 5: Generating ideal guesses.

The hash table based approach shown in Listing 6 iterates on average half the guesses in a layer compared to the naive approach. Unfortunately, these tables are too large ($2^{19} \cdot 6$ bytes $\approx$ 3MB / layer, 27MB in total) to fit in the cache of common CPUs, leading to memory accesses, not computation becoming a bottleneck.

```
function fill_layer(state, layer,
                    filt_mask=0x5806b4a2d16c):
  if layer < len(masks):
    try:
      table = layer_fills[layer][state & filt_mask]
      for fill in table:
        new_state = state | fill
        if layer < last_lfsr_guess:
          fill_layer(new_state>>1,
                     layer+1, filt_mask)
          fill_layer(new_state>>1 | (1<<47),
                     layer+1, filt_mask)
        else:
          fill_layer(lfsr(new_state),
                     layer+1, filt_mask)
    except: # Table miss
      pass
  else:
    test(state)
fill_layer(0, 0)
```

Listing 6: Solution which avoids impossible guesses.

After reaching the maximal depth for the guess-and-determine stage where the first 9 keystream bits are tested without relying on f, we can use the *test* function to find whether any of the candidate states will also generate the remaining 23 bits of keystream.

4.3 Unrolled GD with memoized subfilters

To find a compromise between the redundant work done in the naive approach (subsection 4.1) and the memory limitations encountered in the approach that avoids impossible guesses (subsection 4.2), we look at the filter function f in more depth. Because this function applies its transformation in two steps, we can precompute the *subfilters* f_a and f_b as soon as the input bits for a subfilter are fully determined.

The state information known at layer 1 determines the inputs for subfilters 0 and 3 that are used in the next layer, as visualized in Figure 3. This means that we can precompute (and then *memoize*) the outputs of these subfilters after (conceptually) shifting this bit-valuation to the right by one bit. Using such memoizations we can eliminate the inefficiency of recomputing subfilters f_a and f_b with identical inputs from the internal cipher state. Similarly we can try to memoize these subfilter computations for all upcoming layers.

Optimization 4.5 (Memoization of f subfunction output)**.** Cache the result of any subfilter computation at the first possibility and avoid future recomputation for the same subfilter inputs.

The layer at which each subfilter is entirely determined can be computed using the function in Listing 7 [7], and its results for memoizing the first 9 keystream bit tests are shown in Table 3. This table shows that at a slight additional computational cost in the upper layers, we can save significantly in the lower ones.

Layer	Subfilters computed		Invocations
	Naive	Memoized	
0	5	5	2^{20}
1	5	11	2^{34}
2	5	10	2^{37}
3	5	10	2^{39}
4	5	2	2^{39}
5	5	2	2^{39}
6	5	2	2^{39}
7	5	2	2^{39}
8	5	1	2^{39}

Table 3: Subfilters computed naively or memoized.

For an architecture with v-bit registers, $27 - \log v$ is a general solution for the last bit test to memoize in the bit-sliced (Optimization 4.6) *test* function. This number can be used as the depth parameter for the function in Listing 7 to determine which subfilter inputs for each keystream bit test can be determined at which layer. This

number is however only to be used as a maximum, because in practice cache sizes come in to play again: the cost of evicting our earliest precomputed work from the cache outweighs the gain of deeper optimization. Some experimentation is required to get ideal results.

Our implementation uses 9 nested loops to unroll the recursion, where each layer works on the same globally shared cipher state and permutes its bits. At each layer, subfilter outputs for upcoming keystream bit tests are precomputed and stored in local variables.

4.4 Bit-sliced OpenCL implementation

Optimization 4.6 (Bit-slicing)**.** To parallelize the implementation on modern vector machines we can store the bits for multiple 48-bit cipher states in 48 *vectors* of machine-native register width (v). This allows computing the f function in parallel on v slices at a time [3, 24]. It requires an unrolled implementation (Optimization 4.3), and cannot be applied effectively in the naive GD variant. This is due to the elimination of candidates (on average 50% per layer) quickly leading to a block of bit-sliced states in which every state gets processed, but only few are still valid candidates.

The unrolled, memoized variant of our algorithm (subsection 4.3) uses a globally shared cipher state which builds up the known state information as it is guessed at each layer. This differs from previous variants, where recursive function calls take place that pass the cipher states as a parameter to the next level. Having all information in the same memory area allows for more effective memoization, since each layer of the algorithm can rely on information from earlier layers within its own local scope. This removes the need for shared memory lookups and ensures the workload takes place in a limited, predetermined local memory space.

This is an ideal situation to introduce bit-slicing of the algorithm, where for each state the same operation is performed in parallel. Although roughly half the candidates are eliminated at each layer, all redundant f steps (the bulk of the workload in each iteration) have been moved to earlier layers. This significantly limits the growth of the total workload.

Bit-slicing the memoized implementation allows us to scale up the execution parallelism to the platform's supported native register width in bits. On Intel x86-64 machines with the AVX2 extension, up to 256-bit registers are supported, while with AVX3 they can be 512 bits. Since we can keep the memory requirements contained to only a local memory space, this is also an ideal situation to employ the massive parallelism afforded by the OpenCL GPU computation platform. On OpenCL, registers are 32 bits wide.

Our bit-sliced implementation can be summarized as follows:

1. Starting at layer 1, bit-slice the output of a single successful layer 0 result over all v 48-bit slices to produce v identical bit-sliced cipher states in thread-local memory
2. Set $\log v$ bits to guess in layer 1 to all v valuations using constant v-bit values
3. In 8 nested loops:
   (a) Permute the thread-local cipher state with bit-vectors of zeroes or ones for each bit to guess in the layer (skipping the $\log v$ bits pre-set by constants at layer 1)
   (b) Compute the memoized f function
   (c) Keep a bit-wise index of successful results for each bit tested in a v-bit vector, which is AND-ed with the previous layer's results vector if any – if there are no results, don't iterate deeper
   (d) Precompute all possible LFSR feedback
   (e) Perform memoization for future tests
4. In the innermost loop, where the states are fully determined, perform more (memoized) keystream bit tests as long as there is keystream to test and any of the v cipher states pass, while keeping the LFSR feedback and the results vector updated
5. For the results that get this far, unslice all states and look up matching states by their index from the results vector to pass to the output queue

This implementation works similarly on CPU and GPU platforms[8].

To further optimize the non-linear filter function f in a bit-sliced context, we apply a platform-specific optimization known to us from earlier work [23, 5, 6]. On Nvidia-based GPUs the LUT3 operation can be used to express an arbitrary 3-bit SIMD computation by means of a lookup table encoded in the instruction (`lop3.b32`). We have tried to make an ideal translation from the f function to a construction of such 3-bit LUTs by a best effort hand optimized approach, which gives a significant (more than threefold) improvement in overall performance.

5 Performance evaluation

We have implemented the various approaches detailed in this paper, and have evaluated how they perform on

Optimization type	Platform	Cores	Running time
None (brute force) (32-way bit-sliced) [20]	Tesla C2050	448 CUDA	660 minutes
None (brute force) (32-way bit-sliced) [2]	GTX 780Ti	2880 CUDA	1080 minutes
None (brute force) (32-way bit-sliced) [2]	16 x Tesla K80	79872 CUDA	45 minutes
Naive (optimizations 4.1, 4.2, 4.3)	i7 3700K	4 Intel	127 minutes
Avoiding impossible guesses (optimizations 4.1, 4.3, 4.4)	i7 3700K	4 Intel	96 minutes
256-way bit-sliced, memoized to depth 12 (optimizations 4.3, 4.5, 4.6)	i7 3700K	4 Intel	59 minutes
32-way bit-sliced, memoized to depth 17 (optimizations 4.3, 4.5, 4.6)	GTX 1080Ti	3584 CUDA	**1.2 minutes**

Table 4: Performance evaluation of average running time compared to related work.

our setup. All variants were benchmarked on a desktop system with a quad-core Intel i7 3700K processor with hyperthreading running at 3.5GHz. The GPU is a PNY NVidia GTX 1080Ti, equipped with 3584 CUDA cores running at 1480MHz. A set of four random 32-bit keystreams was generated and used for benchmarking. The execution times for exhaustion of the 48-bit search space were then averaged per implementation. The variance between the four runs is negligible ($\sim 1\%$) for all implementations. The results are shown in Table 4 where we compare them with known results from related work. Our best implementations for both the CPU and GPU architectures are included in the appendix.

6 Conclusion

We demonstrate that a straightforward guess-and-determine approach can in general be optimized to improve the computational throughput, using Hitag2 as a case study. We have iteratively improved upon a simple concept, demonstrating a significant speedup due to careful optimization. Some of our optimizations focus on implementational improvements, while others require analysis of the cipher to identify and subsequently eliminate frequent recomputation. This optimized implementation was ported to the OpenCL platform to enable massively parallel execution, which allows performing the attack on consumer desktop hardware in 1.2 minutes.

Earlier cryptographic attacks on Hitag2 that are faster than brute-force are inapplicable to the RKE context. We have conceptualized and implemented a new practical cryptographic attack against Hitag2 that can be used in the immobilizer context as well as in the more restricting RKE context. With an over 500 times faster implementation than earlier work [2], we perform significantly better than the most practical published attack on Hitag2 RKE. We will release our source code as reference material for related research in the future. Moreover, we invite our fellow researchers to take advantage of the described guess-and-determine optimization process to improve otherwise prohibitively time-consuming implementations towards the point of practicality.

References

[1] Ross J Anderson. Tree functions and cipher systems. *Cryptologia*, 15(3):194–202, 1991.

[2] Ryad Benadjila, Mathieu Renard, José Lopes-Esteves, and Chaouki Kasmi. One car, two frames: Attacks on Hitag-2 remote keyless entry systems revisited. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[3] Eli Biham. A fast new DES implementation in software. In *4th International Workshop on Fast Software Encryption (FSE 1997)*, volume 1267 of *Lecture Notes in Computer Science*, pages 260–272. Springer-Verlag, 1997.

[4] Andrey Bogdanov. Attacks on the KeeLoq block cipher and authentication systems. In *3rd Conference on RFID Security (RFIDSec 2007)*, volume 2007, 2007.

[5] Vadim Bulavintsev, Alexander Semenov, and Oleg Zaikin. Implementation of a brute force attack on the a5/1 keystream generator in a GPU-based volunteer computing project. In *Proceedings of the Third International Conference on BOINC-based High Performance Computing: Fundamental Research and Development (BOINC:FAST 2017)*, volume 3, pages 94–101, 2017.

[6] Vadim Bulavintsev, Alexander Semenov, Oleg Zaikin, and Stepan Kochemazov. A bitslice implementation of andersons attack on A5/1. *Open Engineering*, 8(1):7–16, 2018.

[7] Nicolas T. Courtois, Sean O'Neil, and Jean-Jacques Quisquater. Practical algebraic attacks on the Hitag2 stream cipher. In *12th Information Security Conference (ISC 2009)*, volume 5735 of *Lecture Notes in Computer Science*, pages 167–176. Springer-Verlag, 2009.

[8] Benedikt Driessen, Ralf Hund, Carsten Willems, Carsten Paar, and Thorsten Holz. Don't trust satellite phones: A security analysis of two satphone

standards. In *33rd IEEE Symposium on Security and Privacy (S&P 2012)*, pages 128–142. IEEE Computer Society, 2012.

[9] Xiutao Feng, Jun Liu, Zhaocun Zhou, Chuankun Wu, and Dengguo Feng. A byte-based guess and determine attack on SOSEMANUK. In *16th International Conference on the Theory and Application of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT 2010)*, volume 6477 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 2010.

[10] Flavio D. Garcia, Gerhard de Koning Gans, Ruben Muijrers, Peter van Rossum, Roel Verdult, Ronny Wichers Schreur, and Bart Jacobs. Dismantling MIFARE Classic. In *13th European Symposium on Research in Computer Security (ESORICS 2008)*, volume 5283 of *Lecture Notes in Computer Science*, pages 97–114. Springer-Verlag, 2008.

[11] Flavio D. Garcia, Gerhard de Koning Gans, Roel Verdult, and Milosch Meriac. Dismantling iClass and iClass Elite. In *17th European Symposium on Research in Computer Security (ESORICS 2012)*, volume 7459 of *Lecture Notes in Computer Science*, pages 697–715. Springer-Verlag, 2012.

[12] Flavio D. Garcia, David Oswald, Timo Kasper, and Pierre Pavlidès. Lock it and still lose it - on the (in)security of automotive remote keyless entry systems. In *25th USENIX Security Symposium (USENIX Security 2016)*, pages 929–944. USENIX Association, 2016.

[13] Praveen S Gauravaram and William L Millan. Improved attack on the cellular authentication and voice encryption algorithm (CAVE). In *Cryptographic Algorithms and their Uses (CAU 2004)*, pages 1–13. Queensland University of Technology, 2004.

[14] Jovan Dj Golić. On the security of nonlinear filter generators. In *3rd International Workshop on Fast Software Encryption (FSE 1996)*, volume 1039 of *Lecture Notes in Computer Science*, pages 173–188. Springer-Verlag, 1996.

[15] Jovan Dj. Golić. Cryptanalysis of alleged A5 stream cipher. In *16th International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT 1997)*, volume 1233 of *Lecture Notes in Computer Science*, pages 239–255. Springer-Verlag, 1997.

[16] S.W. Golomb. *Shift Register Sequences*. Holden-Day Series in Information Systems. Holden-Day, 1967.

[17] El Groth. Generation of binary sequences with controllable complexity. *IEEE Transactions on Information Theory*, 17(3):288–296, 1971.

[18] Philip Hawkes and Gregory G Rose. Exploiting multiples of the connection polynomial in word-oriented stream ciphers. In *6th International Conference on the Theory and Application of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT 2000)*, volume 1976 of *Lecture Notes in Computer Science*, pages 303–316. Springer-Verlag, 2000.

[19] Philip Hawkes and Gregory G Rose. Guess-and-determine attacks on SNOW. In *9th International Workshop on Selected Areas in Cryptography (SAC 2002)*, volume 2595 of *Lecture Notes in Computer Science*, pages 37–46. Springer-Verlag, 2003.

[20] Vincent Immler. Breaking hitag 2 revisited. In *2nd International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE 2012)*, volume 7644 of *Lecture Notes in Computer Science*, pages 126–143. Springer-Verlag, 2012.

[21] Edwin Key. An analysis of the structure and complexity of nonlinear binary sequence generators. *IEEE Transactions on Information Theory*, 22(6):732–736, 1976.

[22] GJ Kuhn. Algorithms for self-synchronizing ciphers. In *1st Southern African Conference on Communications and Signal Processing (COMSIG 1988)*, pages 159–164. IEEE, 1988.

[23] Charles Eric LaForest. *High-speed soft-processor architecture for FPGA overlays*. PhD thesis, University of Toronto (Canada), 2015.

[24] Mitsuru Matsui and Junko Nakajima. On the power of bitslice implementation on intel core2 processor. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 121–134. Springer, 2007.

[25] Karsten Nohl, Erik Tews, and Ralf-Philipp Weinmann. Cryptanalysis of the DECT standard cipher. In *17th International Workshop on Fast Software Encryption (FSE 2010)*, volume 6147 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2010.

[26] Enes Pasalic. On guess and determine cryptanalysis of LFSR-based stream ciphers. *IEEE Transactions on Information Theory*, 55(7):3398–3406, 2009.

[27] Vera S Pless. Encryption schemes for computer confidentiality. *IEEE Transactions on Computers*, 100(11):1133–1136, 1977.

[28] Bruce Schneier. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[29] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In *12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer-Verlag, 2009.

[30] Siwei Sun, Lei Hu, Yonghong Xie, and Xiangyong Zeng. Cube cryptanalysis of Hitag2 stream cipher. In *10th International Conference on Cryptology and Network Security (CANS 2011)*, volume 7092 of *Lecture Notes in Computer Science*, pages 15–25. Springer-Verlag, 2011.

[31] Roel Verdult. *The (in)security of proprietary cryptography*. PhD thesis, Radboud University, The Netherlands and KU Leuven, Belgium, April 2015.

[32] Roel Verdult, Flavio D. Garcia, and Josep Balasch. Gone in 360 seconds: Hijacking with Hitag2. In *21st USENIX Security Symposium (USENIX Security 2012)*, pages 237–252. USENIX Association, 2012.

[33] Roel Verdult, Flavio D. Garcia, and Barış Ege. Dismantling megamos crypto: Wirelessly lockpicking a vehicle immobilizer. In *22nd USENIX Security Symposium (USENIX Security 2013)*, pages 703–718. USENIX Association, 2015.

[34] Petr Štembera and Martin Novotný. Breaking Hitag2 with reconfigurable hardware. In *14th Euromicro Conference on Digital System Design (DSD 2011)*, pages 558–563. IEEE Computer Society, 2011.

[35] David Wagner, Leone Simpson, Ed Dawson, John Kelsey, William Millan, and Bruce Schneier. Cryptanalysis of ORYX. In *5th International Workshop on Selected Areas in Cryptography (SAC 1998)*, volume 1556 of *Lecture Notes in Computer Science*, pages 631–631. Springer-Verlag, 1999.

[36] Bin Zhang and Dengguo Feng. New guess-and-determine attack on the self-shrinking generator. In *12th International Conference on the Theory and Application of Cryptology and Information Security, Advances in Cryptology (ASIACRYPT 2006)*, volume 4284 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 2006.

Notes

[1] If the nonces are not related, such as in the immobilizer protocol, we can still use the fact that initial states always share their lowest 16 bits: the correct initial states can still be determined by narrowing down the first results through another exhaustive search attack on a subsequent sample of keystream.

[2] In any case, there are no more than 17 bits that could have flipped (we can be sure that bit 10 flips) and lower bits are on average much more likely to have flipped first provided the counter was not resynchronized by the car.

[3] Note that we never actually make a guess on the state bits 0 or 1 in the first layer: we can actually skip storing these bits in our implementations as a memory optimization but avoid doing so in our examples for simplicity.

[4] Intuitively, for longer keystream samples, this list would be much smaller: a 48-bit sample of keystream would only yield a few valid states.

[5] The *expand* function can be expressed using the x86 instruction `PDEP`, but we have found lookup tables to be faster.

[6] If we don't find a table at the current layer, it means that one or more of our previous guesses was wrong.

[7] The function in Listing 7 can also compute solutions for the situation where we delay our first guess on the LFSR feedback – which doesn't impact the memoization done at the first 9 layers.

[8] There are slight differences in how work is parallelized at layer 0 as a consequence of the platform.

A Memoization algorithm

```
function generate_memos(depth,
                        sub_masks=[0x00000000006c,
                                   0x00000000d100,
                                   0x000004a20000,
                                   0x0002b0000000,
                                   0x580400000000],
                        lfsr_mask=0xce0044c101cd,
                        lfsr_guess_delay=0):
  # keep track of known subfilters and bits
  filt_found = [ [0 for _ in range(len(sub_masks))] for _ in range(len(keystream)) ]
  lfsr_guesses = [0]*lfsr_guess_delay + [1 if i and i <= last_lfsr_gues else 0 for i in range(32)]
  lfsr_found = [0 for _ in(range(len(keystream))) ]
  fill = 0
  for layer in range(len(masks)):
    # track filter input
    fill |= (masks[layer]<<layer)
    if lfsr_guesses[layer]:
        # track LFSR guess
        fill |= (1<<(47+layer+lfsr_guess_delay))
    if (((fill<<1)>>layer)&lfsr_mask) == lfsr_mask:
        # track LFSR feedback
        fill |= (1<<(47+layer))
    lfsr_found[layer] = 1
    for next_bit in range(layer, depth):
      # greedily precompute LFSR feedback
      if (lfsr_mask is not None or layer == len(masks)-1) \
      and not lfsr_found[next_bit] \
      and (((fill << 1) >> next_bit) & lfsr_mask) == lfsr_mask:
        print "LFSR feedback for layer", next_bit-1, "known at layer", layer
        fill |= (1<<(47+next_bit))
        lfsr_found[next_bit] = 1
    # Immediately use these LFSR feedback assumptions
    for next_bit in range(layer, depth):
      # find computable subfilters
      for subfilter in range(len(sub_masks)):
        if filt_found[next_bit][subfilter]:
          continue
        target = sub_masks[subfilter]
        if ((fill>>next_bit) & target) == target:
          print "Subfilter", subfilter, "for layer", next_bit, "known at layer", layer
          filt_found[next_bit][subfilter] = 1
```

Listing 7: Determine steps of the filter to be memoized.

B x86-64 implementation

```
#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <inttypes.h>
#include <pthread.h>

const uint8_t bits[9] = {20, 14, 4, 3, 1, 1, 1, 1, 1};
#define lfsr_inv(state) (((state)<<1) | (__builtin_parityll((state) & ((0xce0044c101cd>>1)|(1ull<<(47))))))
#define i4(x,a,b,c,d) ((uint32_t)((((x)>>(a))&1)<<3)|(((x)>>(b))&1)<<2|(((x)>>(c))&1)<<1|(((x)>>(d))&1))
#define f(state) ((0xdd3929b >> ( (((0x3c65 >> i4(state, 2, 3, 5, 6) ) & 1) <<4) \
                                | ((( 0xee5 >> i4(state, 8,12,14,15) ) & 1) <<3) \
                                | ((( 0xee5 >> i4(state,17,21,23,26) ) & 1) <<2) \
                                | ((( 0xee5 >> i4(state,28,29,31,33) ) & 1) <<1) \
                                | (((0x3c65 >> i4(state,34,43,44,46) ) & 1) ))) & 1)
```

```
#define MAX_BITSLICES 256
#define VECTOR_SIZE (MAX_BITSLICES/8)

typedef unsigned int __attribute__((aligned(VECTOR_SIZE))) __attribute__((vector_size(VECTOR_SIZE))) bitslice_value_t;
typedef union {
    bitslice_value_t value;
    uint64_t bytes64[MAX_BITSLICES/64];
    uint8_t bytes[MAX_BITSLICES/8];
} bitslice_t;

// we never actually set or use the lowest 2 bits the initial state, so we can save 2 bitslices everywhere
__thread bitslice_t state[-2+32+48];

bitslice_t keystream[32];
bitslice_t bs_zeroes, bs_ones;

#define f_a_bs(a,b,c,d)       (~(((a|b)&c)^(a|d)^b)) // 6 ops
#define f_b_bs(a,b,c,d)       (~(((d|c)&(a^b))^(d|a|b))) // 7 ops
#define f_c_bs(a,b,c,d,e)     (~(((((c^e)|d)&a)^b)&(c^b))^(((d^e)|a)&((d^b)|c)))) // 13 ops
#define lfsr_bs(i) (state[-2+i+ 0].value ^ state[-2+i+ 2].value ^ state[-2+i+ 3].value ^ state[-2+i+ 6].value ^ \
                    state[-2+i+ 7].value ^ state[-2+i+ 8].value ^ state[-2+i+16].value ^ state[-2+i+22].value ^ \
                    state[-2+i+23].value ^ state[-2+i+26].value ^ state[-2+i+30].value ^ state[-2+i+41].value ^ \
                    state[-2+i+42].value ^ state[-2+i+43].value ^ state[-2+i+46].value ^ state[-2+i+47].value);
#define get_bit(n, word) ((word >> (n)) & 1)
#define get_vector_bit(slice, value) get_bit(slice&0x3f, value.bytes64[slice>>6])

const uint64_t expand(uint64_t mask, uint64_t value){
    uint64_t fill = 0;
    for(uint64_t bit_index = 0; bit_index < 48; bit_index++){
        if(mask & 1){
            fill |= (value&1)<<bit_index;
            value >>= 1;
        }
        mask >>= 1;
    }
    return fill;
}

void bitslice(const uint64_t value, bitslice_t * restrict bitsliced_value, const size_t bit_len, bool reverse){
    size_t bit_idx;
    for(bit_idx = 0; bit_idx < bit_len; bit_idx++){
        bool bit;
        if(reverse){
          bit = get_bit(bit_len-1-bit_idx, value);
        } else {
          bit = get_bit(bit_idx, value);
        }
        if(bit){
            bitsliced_value[bit_idx].value = bs_ones.value;
        } else {
            bitsliced_value[bit_idx].value = bs_zeroes.value;
        }
    }
}

const uint64_t unbitslice(const bitslice_t * restrict b, const uint8_t s, const uint8_t n){
    uint64_t result = 0;
    for (uint8_t i = 0; i < n; ++i) {
        result <<= 1;
        result |= get_vector_bit(s, b[n-1-i]);
    }
    return result;
}

uint64_t candidates[(1<<20)];
bitslice_t initial_bitslices[48];
size_t filter_pos[20] = {4, 7, 9, 13, 16, 18, 22, 24, 27, 30, 32, 35, 45, 47  };
size_t thread_count = 8;
size_t layer_0_found;
void find_state(size_t thread);

void main(int argc, char* argv[]){
  // set constants
  memset(bs_ones.bytes, 0xff, VECTOR_SIZE);
  memset(bs_zeroes.bytes, 0x00, VECTOR_SIZE);

  uint64_t target = 0x48656c6c;

  // bitslice inverse target bits
  bitslice(~target, keystream, 32, true);

  // bitslice all possible 256 values in the lowest 8 bits
  memset(initial_bitslices[0].bytes, 0xaa, VECTOR_SIZE);
  memset(initial_bitslices[1].bytes, 0xcc, VECTOR_SIZE);
  memset(initial_bitslices[2].bytes, 0xf0, VECTOR_SIZE);
  size_t interval = 1;
  for(size_t bit = 3; bit < 8; bit++){
    for(size_t byte = 0; byte < VECTOR_SIZE;){
      for(size_t length = 0; length < interval; length++){
        initial_bitslices[bit].bytes[byte++] = 0x00;
      }
      for(size_t length = 0; length < interval; length++){
        initial_bitslices[bit].bytes[byte++] = 0xff;
      }
    }
    interval<<=1;
  }

  // compute layer 0 output
  for(size_t i0 = 0; i0 < 1<<20; i0++){
    uint64_t state0 = expand(0x5806b4a2d16c, i0);
    if(f(state0) == target>>31){
      candidates[layer_0_found++] = state0;
    }
  }

  // start threads and wait on them
```

```
  pthread_t thread_handles[thread_count];
  for(size_t thread = 0; thread < thread_count; thread++){
    pthread_create(&thread_handles[thread], NULL, find_state, (void*) thread);
  }
  for(size_t thread = 0; thread < thread_count; thread++){
    pthread_join(thread_handles[thread], NULL);
  }
}

void find_state(size_t thread){
  for(size_t index = thread; index < layer_0_found; index+=thread_count){
    uint64_t state0 = candidates[index];
    bitslice(state0>>2, &state[0], 46, false);
    for(size_t bit = 0; bit < 8; bit++){
      state[-2+filter_pos[bit]] = initial_bitslices[bit];
    }
    for(uint16_t i1 = 0; i1 < (1<<(bits[1]+1)>>8); i1++){
      state[-2+27].value = ((bool) (i1 & 0x1)) ? bs_ones.value : bs_zeroes.value;
      state[-2+30].value = ((bool) (i1 & 0x2)) ? bs_ones.value : bs_zeroes.value;
      state[-2+32].value = ((bool) (i1 & 0x4)) ? bs_ones.value : bs_zeroes.value;
      state[-2+35].value = ((bool) (i1 & 0x8)) ? bs_ones.value : bs_zeroes.value;
      state[-2+45].value = ((bool) (i1 & 0x10)) ? bs_ones.value : bs_zeroes.value;
      state[-2+47].value = ((bool) (i1 & 0x20)) ? bs_ones.value : bs_zeroes.value;
      state[-2+48].value = ((bool) (i1 & 0x40)) ? bs_ones.value : bs_zeroes.value; // guess lfsr output 0
      // 0xfc07fef3f9fe
      const bitslice_value_t filter1_0 = f_a_bs(state[-2+3].value,state[-2+4].value,state[-2+6].value,state[-2+7].value);
      const bitslice_value_t filter1_1 = f_b_bs(state[-2+9].value,state[-2+13].value,state[-2+15].value,state[-2+16].value);
      const bitslice_value_t filter1_2 = f_b_bs(state[-2+18].value,state[-2+22].value,state[-2+24].value,state[-2+27].value);
      const bitslice_value_t filter1_3 = f_b_bs(state[-2+29].value,state[-2+30].value,state[-2+32].value,state[-2+34].value);
      const bitslice_value_t filter1_4 = f_a_bs(state[-2+35].value,state[-2+44].value,state[-2+45].value,state[-2+47].value);
      const bitslice_value_t filter1 = f_c_bs(filter1_0, filter1_1, filter1_2, filter1_3, filter1_4);
      bitslice_t results1;
      results1.value = filter1 ^ keystream[1].value;
      if(results1.bytes64[0] == 0
         && results1.bytes64[1] == 0
         && results1.bytes64[2] == 0
         && results1.bytes64[3] == 0
      ){
        continue;
      }
      const bitslice_value_t filter2_0 = f_a_bs(state[-2+4].value,state[-2+5].value,state[-2+7].value,state[-2+8].value);
      const bitslice_value_t filter2_3 = f_b_bs(state[-2+30].value,state[-2+31].value,state[-2+33].value,state[-2+35].value);
      const bitslice_value_t filter3_0 = f_a_bs(state[-2+5].value,state[-2+6].value,state[-2+8].value,state[-2+9].value);
      const bitslice_value_t filter5_2 = f_b_bs(state[-2+22].value,state[-2+26].value,state[-2+28].value,state[-2+31].value);
      const bitslice_value_t filter6_2 = f_b_bs(state[-2+23].value,state[-2+27].value,state[-2+29].value,state[-2+32].value);
      const bitslice_value_t filter7_2 = f_b_bs(state[-2+24].value,state[-2+28].value,state[-2+30].value,state[-2+33].value);
      const bitslice_value_t filter9_1 = f_b_bs(state[-2+17].value,state[-2+21].value,state[-2+23].value,state[-2+24].value);
      const bitslice_value_t filter9_2 = f_b_bs(state[-2+26].value,state[-2+30].value,state[-2+32].value,state[-2+35].value);
      const bitslice_value_t filter10_0 = f_a_bs(state[-2+12].value,state[-2+13].value,state[-2+15].value,state[-2+16].value);
      const bitslice_value_t filter11_0 = f_a_bs(state[-2+13].value,state[-2+14].value,state[-2+16].value,state[-2+17].value);
      const bitslice_value_t filter12_0 = f_a_bs(state[-2+14].value,state[-2+15].value,state[-2+17].value,state[-2+18].value);
      for(uint16_t i2 = 0; i2 < (1<<(bits[2]+1)); i2++){
        state[-2+10].value = ((bool) (i2 & 0x1)) ? bs_ones.value : bs_zeroes.value;
        state[-2+19].value = ((bool) (i2 & 0x2)) ? bs_ones.value : bs_zeroes.value;
        state[-2+25].value = ((bool) (i2 & 0x4)) ? bs_ones.value : bs_zeroes.value;
        state[-2+36].value = ((bool) (i2 & 0x8)) ? bs_ones.value : bs_zeroes.value;
        state[-2+49].value = ((bool) (i2 & 0x10)) ? bs_ones.value : bs_zeroes.value; // guess lfsr output 1
        // 0xfe07fffbfdff
        const bitslice_value_t filter2_1 = f_b_bs(state[-2+10].value,state[-2+14].value,state[-2+16].value,state[-2+17].value);
        const bitslice_value_t filter2_2 = f_b_bs(state[-2+19].value,state[-2+23].value,state[-2+25].value,state[-2+28].value);
        const bitslice_value_t filter2_4 = f_a_bs(state[-2+36].value,state[-2+45].value,state[-2+46].value,state[-2+48].value);
        const bitslice_value_t filter2 = f_c_bs(filter2_0, filter2_1, filter2_2, filter2_3, filter2_4);
        bitslice_t results2;
        results2.value = results1.value & (filter2 ^ keystream[2].value);
        if(results2.bytes64[0] == 0
           && results2.bytes64[1] == 0
           && results2.bytes64[2] == 0
           && results2.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+50].value = lfsr_bs(2);
        const bitslice_value_t filter3_3 = f_b_bs(state[-2+31].value,state[-2+32].value,state[-2+34].value,state[-2+36].value);
        const bitslice_value_t filter4_0 = f_a_bs(state[-2+6].value,state[-2+7].value,state[-2+9].value,state[-2+10].value);
        const bitslice_value_t filter4_1 = f_b_bs(state[-2+12].value,state[-2+16].value,state[-2+18].value,state[-2+19].value);
        const bitslice_value_t filter4_2 = f_b_bs(state[-2+21].value,state[-2+25].value,state[-2+27].value,state[-2+30].value);
        const bitslice_value_t filter7_0 = f_a_bs(state[-2+9].value,state[-2+10].value,state[-2+12].value,state[-2+13].value);
        const bitslice_value_t filter7_1 = f_b_bs(state[-2+15].value,state[-2+19].value,state[-2+21].value,state[-2+22].value);
        const bitslice_value_t filter8_2 = f_b_bs(state[-2+25].value,state[-2+29].value,state[-2+31].value,state[-2+34].value);
        const bitslice_value_t filter10_1 = f_b_bs(state[-2+18].value,state[-2+22].value,state[-2+24].value,state[-2+25].value);
        const bitslice_value_t filter10_2 = f_b_bs(state[-2+27].value,state[-2+31].value,state[-2+33].value,state[-2+36].value);
        const bitslice_value_t filter11_1 = f_b_bs(state[-2+19].value,state[-2+23].value,state[-2+25].value,state[-2+26].value);
        for(uint8_t i3 = 0; i3 < (1<<bits[3]); i3++){
          state[-2+11].value = ((bool) (i3 & 0x1)) ? bs_ones.value : bs_zeroes.value;
          state[-2+20].value = ((bool) (i3 & 0x2)) ? bs_ones.value : bs_zeroes.value;
          state[-2+37].value = ((bool) (i3 & 0x4)) ? bs_ones.value : bs_zeroes.value;
          // 0xff07fffffff
          const bitslice_value_t filter3_1 = f_b_bs(state[-2+11].value,state[-2+15].value,state[-2+17].value,state[-2+18].value);
          const bitslice_value_t filter3_2 = f_b_bs(state[-2+20].value,state[-2+24].value,state[-2+26].value,state[-2+29].value);
          const bitslice_value_t filter3_4 = f_a_bs(state[-2+37].value,state[-2+46].value,state[-2+47].value,state[-2+49].value);
          const bitslice_value_t filter3 = f_c_bs(filter3_0, filter3_1, filter3_2, filter3_3, filter3_4);
          bitslice_t results3;
          results3.value = results2.value & (filter3 ^ keystream[3].value);
          if(results3.bytes64[0] == 0
             && results3.bytes64[1] == 0
             && results3.bytes64[2] == 0
             && results3.bytes64[3] == 0
          ){
            continue;
          }
          state[-2+51].value = lfsr_bs(3);
          state[-2+52].value = lfsr_bs(4);
          state[-2+53].value = lfsr_bs(5);
          state[-2+54].value = lfsr_bs(6);
          state[-2+55].value = lfsr_bs(7);
          const bitslice_value_t filter4_3 = f_b_bs(state[-2+32].value,state[-2+33].value,state[-2+35].value,state[-2+37].value);
          const bitslice_value_t filter5_0 = f_a_bs(state[-2+7].value,state[-2+8].value,state[-2+10].value,state[-2+11].value);
```

```
const bitslice_value_t filter5_1 = f_b_bs(state[-2+13].value,state[-2+17].value,state[-2+19].value,state[-2+20].value);
const bitslice_value_t filter6_0 = f_a_bs(state[-2+8].value,state[-2+9].value,state[-2+11].value,state[-2+12].value);
const bitslice_value_t filter6_1 = f_b_bs(state[-2+14].value,state[-2+18].value,state[-2+20].value,state[-2+21].value);
const bitslice_value_t filter8_0 = f_a_bs(state[-2+10].value,state[-2+11].value,state[-2+13].value,state[-2+14].value);
const bitslice_value_t filter8_1 = f_b_bs(state[-2+16].value,state[-2+20].value,state[-2+22].value,state[-2+23].value);
const bitslice_value_t filter9_0 = f_a_bs(state[-2+11].value,state[-2+12].value,state[-2+14].value,state[-2+15].value);
const bitslice_value_t filter9_4 = f_a_bs(state[-2+43].value,state[-2+52].value,state[-2+53].value,state[-2+55].value);
const bitslice_value_t filter11_2 = f_b_bs(state[-2+28].value,state[-2+32].value,state[-2+34].value,state[-2+37].value);
const bitslice_value_t filter12_1 = f_b_bs(state[-2+20].value,state[-2+24].value,state[-2+26].value,state[-2+27].value);
for(uint8_t i4 = 0; i4 < (1<<bits[4]); i4++){
  state[-2+38].value = ((bool) (i4 & 0x1)) ? bs_ones.value : bs_zeroes.value;
  // 0xff87ffffffff
  const bitslice_value_t filter4_4 = f_a_bs(state[-2+38].value,state[-2+47].value,state[-2+48].value,state[-2+50].value);
  const bitslice_value_t filter4 = f_c_bs(filter4_0, filter4_1, filter4_2, filter4_3, filter4_4);
  bitslice_t results4;
  results4.value = results3.value & (filter4 ^ keystream[4].value);
  if(results4.bytes64[0] == 0
     && results4.bytes64[1] == 0
     && results4.bytes64[2] == 0
     && results4.bytes64[3] == 0
  ){
    continue;
  }
  const bitslice_value_t filter5_3 = f_b_bs(state[-2+33].value,state[-2+34].value,state[-2+36].value,state[-2+38].value);
  const bitslice_value_t filter12_2 = f_b_bs(state[-2+29].value,state[-2+33].value,state[-2+35].value,state[-2+38].value);
  for(uint8_t i5 = 0; i5 < (1<<bits[5]); i5++){
    state[-2+39].value = ((bool) (i5 & 0x1)) ? bs_ones.value : bs_zeroes.value;
    // 0xffc7ffffffff
    const bitslice_value_t filter5_4 = f_a_bs(state[-2+39].value,state[-2+48].value,state[-2+49].value,state[-2+51].value);
    const bitslice_value_t filter5 = f_c_bs(filter5_0, filter5_1, filter5_2, filter5_3, filter5_4);
    bitslice_t results5;
    results5.value = results4.value & (filter5 ^ keystream[5].value);
    if(results5.bytes64[0] == 0
       && results5.bytes64[1] == 0
       && results5.bytes64[2] == 0
       && results5.bytes64[3] == 0
    ){
      continue;
    }
    const bitslice_value_t filter6_3 = f_b_bs(state[-2+34].value,state[-2+35].value,state[-2+37].value,state[-2+39].value);
    for(uint8_t i6 = 0; i6 < (1<<bits[6]); i6++){
      state[-2+40].value = ((bool) (i6 & 0x1)) ? bs_ones.value : bs_zeroes.value;
      // 0xffe7ffffffff
      const bitslice_value_t filter6_4 = f_a_bs(state[-2+40].value,state[-2+49].value,state[-2+50].value,state[-2+52].value);
      const bitslice_value_t filter6 = f_c_bs(filter6_0, filter6_1, filter6_2, filter6_3, filter6_4);
      bitslice_t results6;
      results6.value = results5.value & (filter6 ^ keystream[6].value);
      if(results6.bytes64[0] == 0
         && results6.bytes64[1] == 0
         && results6.bytes64[2] == 0
         && results6.bytes64[3] == 0
      ){
        continue;
      }
      const bitslice_value_t filter7_3 = f_b_bs(state[-2+35].value,state[-2+36].value,state[-2+38].value,state[-2+40].value);
      for(uint8_t i7 = 0; i7 < (1<<bits[7]); i7++){
        state[-2+41].value = ((bool) (i7 & 0x1)) ? bs_ones.value : bs_zeroes.value;
        // 0xfff7ffffffff
        const bitslice_value_t filter7_4 = f_a_bs(state[-2+41].value,state[-2+50].value,state[-2+51].value,state[-2+53].value);
        const bitslice_value_t filter7 = f_c_bs(filter7_0, filter7_1, filter7_2, filter7_3, filter7_4);
        bitslice_t results7;
        results7.value = results6.value & (filter7 ^ keystream[7].value);
        if(results7.bytes64[0] == 0
           && results7.bytes64[1] == 0
           && results7.bytes64[2] == 0
           && results7.bytes64[3] == 0
        ){
          continue;
        }
        const bitslice_value_t filter8_3 = f_b_bs(state[-2+36].value,state[-2+37].value,state[-2+39].value,state[-2+41].value);
        const bitslice_value_t filter10_3 = f_b_bs(state[-2+38].value,state[-2+39].value,state[-2+41].value,state[-2+43].value);
        const bitslice_value_t filter12_3 = f_b_bs(state[-2+40].value,state[-2+41].value,state[-2+43].value,state[-2+45].value);
        for(uint8_t i8 = 0; i8 < (1<<bits[8]); i8++){
          state[-2+42].value = ((bool) (i8 & 0x1)) ? bs_ones.value : bs_zeroes.value;
          // 0xffffffffffff
          const bitslice_value_t filter8_4 = f_a_bs(state[-2+42].value,state[-2+51].value,state[-2+52].value,state[-2+54].value);
          const bitslice_value_t filter9_3 = f_b_bs(state[-2+37].value,state[-2+38].value,state[-2+40].value,state[-2+42].value);
          const bitslice_value_t filter11_3 = f_b_bs(state[-2+39].value,state[-2+40].value,state[-2+42].value,state[-2+44].value);
          const bitslice_value_t filter8 = f_c_bs(filter8_0, filter8_1, filter8_2, filter8_3, filter8_4);
          bitslice_t results8;
          results8.value = results7.value & (filter8 ^ keystream[8].value);
          if(results8.bytes64[0] == 0
             && results8.bytes64[1] == 0
             && results8.bytes64[2] == 0
             && results8.bytes64[3] == 0
          ){
            continue;
          }
          const bitslice_value_t filter9 = f_c_bs(filter9_0, filter9_1, filter9_2, filter9_3, filter9_4);
          results8.value &= (filter9 ^ keystream[9].value);
          if(results8.bytes64[0] == 0
             && results8.bytes64[1] == 0
             && results8.bytes64[2] == 0
             && results8.bytes64[3] == 0
          ){
            continue;
          }
          state[-2+56].value = lfsr_bs(8);
          const bitslice_value_t filter10_4 = f_a_bs(state[-2+44].value,state[-2+53].value,state[-2+54].value,state[-2+56].value);
          const bitslice_value_t filter10 = f_c_bs(filter10_0, filter10_1, filter10_2, filter10_3, filter10_4);
          results8.value &= (filter10 ^ keystream[10].value);
          if(results8.bytes64[0] == 0
             && results8.bytes64[1] == 0
             && results8.bytes64[2] == 0
             && results8.bytes64[3] == 0
          ){
            continue;
```

```
}
state[-2+57].value = lfsr_bs(9);
const bitslice_value_t filter11_4 = f_a_bs(state[-2+45].value,state[-2+54].value,state[-2+55].value,state[-2+57].value);
const bitslice_value_t filter11 = f_c_bs(filter11_0, filter11_1, filter11_2, filter11_3, filter11_4);
results8.value &= (filter11 ^ keystream[11].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+58].value = lfsr_bs(10);
const bitslice_value_t filter12_4 = f_a_bs(state[-2+46].value,state[-2+55].value,state[-2+56].value,state[-2+58].value);
const bitslice_value_t filter12 = f_c_bs(filter12_0, filter12_1, filter12_2, filter12_3, filter12_4);
results8.value &= (filter12 ^ keystream[12].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+59].value = lfsr_bs(11);
const bitslice_value_t filter13_0 = f_a_bs(state[-2+15].value,state[-2+16].value,state[-2+18].value,state[-2+19].value);
const bitslice_value_t filter13_1 = f_b_bs(state[-2+21].value,state[-2+25].value,state[-2+27].value,state[-2+28].value);
const bitslice_value_t filter13_2 = f_b_bs(state[-2+30].value,state[-2+34].value,state[-2+36].value,state[-2+39].value);
const bitslice_value_t filter13_3 = f_b_bs(state[-2+41].value,state[-2+42].value,state[-2+44].value,state[-2+46].value);
const bitslice_value_t filter13_4 = f_a_bs(state[-2+47].value,state[-2+56].value,state[-2+57].value,state[-2+59].value);
const bitslice_value_t filter13 = f_c_bs(filter13_0, filter13_1, filter13_2, filter13_3, filter13_4);
results8.value &= (filter13 ^ keystream[13].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+60].value = lfsr_bs(12);
const bitslice_value_t filter14_0 = f_a_bs(state[-2+16].value,state[-2+17].value,state[-2+19].value,state[-2+20].value);
const bitslice_value_t filter14_1 = f_b_bs(state[-2+22].value,state[-2+26].value,state[-2+28].value,state[-2+29].value);
const bitslice_value_t filter14_2 = f_b_bs(state[-2+31].value,state[-2+35].value,state[-2+37].value,state[-2+40].value);
const bitslice_value_t filter14_3 = f_b_bs(state[-2+42].value,state[-2+43].value,state[-2+45].value,state[-2+47].value);
const bitslice_value_t filter14_4 = f_a_bs(state[-2+48].value,state[-2+57].value,state[-2+58].value,state[-2+60].value);
const bitslice_value_t filter14 = f_c_bs(filter14_0, filter14_1, filter14_2, filter14_3, filter14_4);
results8.value &= (filter14 ^ keystream[14].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+61].value = lfsr_bs(13);
const bitslice_value_t filter15_0 = f_a_bs(state[-2+17].value,state[-2+18].value,state[-2+20].value,state[-2+21].value);
const bitslice_value_t filter15_1 = f_b_bs(state[-2+23].value,state[-2+27].value,state[-2+29].value,state[-2+30].value);
const bitslice_value_t filter15_2 = f_b_bs(state[-2+32].value,state[-2+36].value,state[-2+38].value,state[-2+41].value);
const bitslice_value_t filter15_3 = f_b_bs(state[-2+43].value,state[-2+44].value,state[-2+46].value,state[-2+48].value);
const bitslice_value_t filter15_4 = f_a_bs(state[-2+49].value,state[-2+58].value,state[-2+59].value,state[-2+61].value);
const bitslice_value_t filter15 = f_c_bs(filter15_0, filter15_1, filter15_2, filter15_3, filter15_4);
results8.value &= (filter15 ^ keystream[15].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+62].value = lfsr_bs(14);
const bitslice_value_t filter16_0 = f_a_bs(state[-2+18].value,state[-2+19].value,state[-2+21].value,state[-2+22].value);
const bitslice_value_t filter16_1 = f_b_bs(state[-2+24].value,state[-2+28].value,state[-2+30].value,state[-2+31].value);
const bitslice_value_t filter16_2 = f_b_bs(state[-2+33].value,state[-2+37].value,state[-2+39].value,state[-2+42].value);
const bitslice_value_t filter16_3 = f_b_bs(state[-2+44].value,state[-2+45].value,state[-2+47].value,state[-2+49].value);
const bitslice_value_t filter16_4 = f_a_bs(state[-2+50].value,state[-2+59].value,state[-2+60].value,state[-2+62].value);
const bitslice_value_t filter16 = f_c_bs(filter16_0, filter16_1, filter16_2, filter16_3, filter16_4);
results8.value &= (filter16 ^ keystream[16].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+63].value = lfsr_bs(15);
const bitslice_value_t filter17_0 = f_a_bs(state[-2+19].value,state[-2+20].value,state[-2+22].value,state[-2+23].value);
const bitslice_value_t filter17_1 = f_b_bs(state[-2+25].value,state[-2+29].value,state[-2+31].value,state[-2+32].value);
const bitslice_value_t filter17_2 = f_b_bs(state[-2+34].value,state[-2+38].value,state[-2+40].value,state[-2+43].value);
const bitslice_value_t filter17_3 = f_b_bs(state[-2+45].value,state[-2+46].value,state[-2+48].value,state[-2+50].value);
const bitslice_value_t filter17_4 = f_a_bs(state[-2+51].value,state[-2+60].value,state[-2+61].value,state[-2+63].value);
const bitslice_value_t filter17 = f_c_bs(filter17_0, filter17_1, filter17_2, filter17_3, filter17_4);
results8.value &= (filter17 ^ keystream[17].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+64].value = lfsr_bs(16);
const bitslice_value_t filter18_0 = f_a_bs(state[-2+20].value,state[-2+21].value,state[-2+23].value,state[-2+24].value);
const bitslice_value_t filter18_1 = f_b_bs(state[-2+26].value,state[-2+30].value,state[-2+32].value,state[-2+33].value);
const bitslice_value_t filter18_2 = f_b_bs(state[-2+35].value,state[-2+39].value,state[-2+41].value,state[-2+44].value);
const bitslice_value_t filter18_3 = f_b_bs(state[-2+46].value,state[-2+47].value,state[-2+49].value,state[-2+51].value);
const bitslice_value_t filter18_4 = f_a_bs(state[-2+52].value,state[-2+61].value,state[-2+62].value,state[-2+64].value);
const bitslice_value_t filter18 = f_c_bs(filter18_0, filter18_1, filter18_2, filter18_3, filter18_4);
results8.value &= (filter18 ^ keystream[18].value);
if(results8.bytes64[0] == 0
```

```
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+65].value = lfsr_bs(17);
const bitslice_value_t filter19_0 = f_a_bs(state[-2+21].value,state[-2+22].value,state[-2+24].value,state[-2+25].value);
const bitslice_value_t filter19_1 = f_b_bs(state[-2+27].value,state[-2+31].value,state[-2+33].value,state[-2+34].value);
const bitslice_value_t filter19_2 = f_b_bs(state[-2+36].value,state[-2+40].value,state[-2+42].value,state[-2+45].value);
const bitslice_value_t filter19_3 = f_b_bs(state[-2+47].value,state[-2+48].value,state[-2+50].value,state[-2+52].value);
const bitslice_value_t filter19_4 = f_a_bs(state[-2+53].value,state[-2+62].value,state[-2+63].value,state[-2+65].value);
const bitslice_value_t filter19 = f_c_bs(filter19_0, filter19_1, filter19_2, filter19_3, filter19_4);
results8.value &= (filter19 ^ keystream[19].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+66].value = lfsr_bs(18);
const bitslice_value_t filter20_0 = f_a_bs(state[-2+22].value,state[-2+23].value,state[-2+25].value,state[-2+26].value);
const bitslice_value_t filter20_1 = f_b_bs(state[-2+28].value,state[-2+32].value,state[-2+34].value,state[-2+35].value);
const bitslice_value_t filter20_2 = f_b_bs(state[-2+37].value,state[-2+41].value,state[-2+43].value,state[-2+46].value);
const bitslice_value_t filter20_3 = f_b_bs(state[-2+48].value,state[-2+49].value,state[-2+51].value,state[-2+53].value);
const bitslice_value_t filter20_4 = f_a_bs(state[-2+54].value,state[-2+63].value,state[-2+64].value,state[-2+66].value);
const bitslice_value_t filter20 = f_c_bs(filter20_0, filter20_1, filter20_2, filter20_3, filter20_4);
results8.value &= (filter20 ^ keystream[20].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+67].value = lfsr_bs(19);
const bitslice_value_t filter21_0 = f_a_bs(state[-2+23].value,state[-2+24].value,state[-2+26].value,state[-2+27].value);
const bitslice_value_t filter21_1 = f_b_bs(state[-2+29].value,state[-2+33].value,state[-2+35].value,state[-2+36].value);
const bitslice_value_t filter21_2 = f_b_bs(state[-2+38].value,state[-2+42].value,state[-2+44].value,state[-2+47].value);
const bitslice_value_t filter21_3 = f_b_bs(state[-2+49].value,state[-2+50].value,state[-2+52].value,state[-2+54].value);
const bitslice_value_t filter21_4 = f_a_bs(state[-2+55].value,state[-2+64].value,state[-2+65].value,state[-2+67].value);
const bitslice_value_t filter21 = f_c_bs(filter21_0, filter21_1, filter21_2, filter21_3, filter21_4);
results8.value &= (filter21 ^ keystream[21].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+68].value = lfsr_bs(20);
const bitslice_value_t filter22_0 = f_a_bs(state[-2+24].value,state[-2+25].value,state[-2+27].value,state[-2+28].value);
const bitslice_value_t filter22_1 = f_b_bs(state[-2+30].value,state[-2+34].value,state[-2+36].value,state[-2+37].value);
const bitslice_value_t filter22_2 = f_b_bs(state[-2+39].value,state[-2+43].value,state[-2+45].value,state[-2+48].value);
const bitslice_value_t filter22_3 = f_b_bs(state[-2+50].value,state[-2+51].value,state[-2+53].value,state[-2+55].value);
const bitslice_value_t filter22_4 = f_a_bs(state[-2+56].value,state[-2+65].value,state[-2+66].value,state[-2+68].value);
const bitslice_value_t filter22 = f_c_bs(filter22_0, filter22_1, filter22_2, filter22_3, filter22_4);
results8.value &= (filter22 ^ keystream[22].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+69].value = lfsr_bs(21);
const bitslice_value_t filter23_0 = f_a_bs(state[-2+25].value,state[-2+26].value,state[-2+28].value,state[-2+29].value);
const bitslice_value_t filter23_1 = f_b_bs(state[-2+31].value,state[-2+35].value,state[-2+37].value,state[-2+38].value);
const bitslice_value_t filter23_2 = f_b_bs(state[-2+40].value,state[-2+44].value,state[-2+46].value,state[-2+49].value);
const bitslice_value_t filter23_3 = f_b_bs(state[-2+51].value,state[-2+52].value,state[-2+54].value,state[-2+56].value);
const bitslice_value_t filter23_4 = f_a_bs(state[-2+57].value,state[-2+66].value,state[-2+67].value,state[-2+69].value);
const bitslice_value_t filter23 = f_c_bs(filter23_0, filter23_1, filter23_2, filter23_3, filter23_4);
results8.value &= (filter23 ^ keystream[23].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+70].value = lfsr_bs(22);
const bitslice_value_t filter24_0 = f_a_bs(state[-2+26].value,state[-2+27].value,state[-2+29].value,state[-2+30].value);
const bitslice_value_t filter24_1 = f_b_bs(state[-2+32].value,state[-2+36].value,state[-2+38].value,state[-2+39].value);
const bitslice_value_t filter24_2 = f_b_bs(state[-2+41].value,state[-2+45].value,state[-2+47].value,state[-2+50].value);
const bitslice_value_t filter24_3 = f_b_bs(state[-2+52].value,state[-2+53].value,state[-2+55].value,state[-2+57].value);
const bitslice_value_t filter24_4 = f_a_bs(state[-2+58].value,state[-2+67].value,state[-2+68].value,state[-2+70].value);
const bitslice_value_t filter24 = f_c_bs(filter24_0, filter24_1, filter24_2, filter24_3, filter24_4);
results8.value &= (filter24 ^ keystream[24].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
   && results8.bytes64[3] == 0
){
  continue;
}
state[-2+71].value = lfsr_bs(23);
const bitslice_value_t filter25_0 = f_a_bs(state[-2+27].value,state[-2+28].value,state[-2+30].value,state[-2+31].value);
const bitslice_value_t filter25_1 = f_b_bs(state[-2+33].value,state[-2+37].value,state[-2+39].value,state[-2+40].value);
const bitslice_value_t filter25_2 = f_b_bs(state[-2+42].value,state[-2+46].value,state[-2+48].value,state[-2+51].value);
const bitslice_value_t filter25_3 = f_b_bs(state[-2+53].value,state[-2+54].value,state[-2+56].value,state[-2+58].value);
const bitslice_value_t filter25_4 = f_a_bs(state[-2+59].value,state[-2+68].value,state[-2+69].value,state[-2+71].value);
const bitslice_value_t filter25 = f_c_bs(filter25_0, filter25_1, filter25_2, filter25_3, filter25_4);
results8.value &= (filter25 ^ keystream[25].value);
if(results8.bytes64[0] == 0
   && results8.bytes64[1] == 0
   && results8.bytes64[2] == 0
```

```
          && results8.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+72].value = lfsr_bs(24);
        const bitslice_value_t filter26_0 = f_a_bs(state[-2+28].value,state[-2+29].value,state[-2+31].value,state[-2+32].value);
        const bitslice_value_t filter26_1 = f_b_bs(state[-2+34].value,state[-2+38].value,state[-2+40].value,state[-2+41].value);
        const bitslice_value_t filter26_2 = f_b_bs(state[-2+43].value,state[-2+47].value,state[-2+49].value,state[-2+52].value);
        const bitslice_value_t filter26_3 = f_b_bs(state[-2+54].value,state[-2+55].value,state[-2+57].value,state[-2+59].value);
        const bitslice_value_t filter26_4 = f_a_bs(state[-2+60].value,state[-2+69].value,state[-2+70].value,state[-2+72].value);
        const bitslice_value_t filter26 = f_c_bs(filter26_0, filter26_1, filter26_2, filter26_3, filter26_4);
        results8.value &= (filter26 ^ keystream[26].value);
        if(results8.bytes64[0] == 0
           && results8.bytes64[1] == 0
           && results8.bytes64[2] == 0
           && results8.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+73].value = lfsr_bs(25);
        const bitslice_value_t filter27_0 = f_a_bs(state[-2+29].value,state[-2+30].value,state[-2+32].value,state[-2+33].value);
        const bitslice_value_t filter27_1 = f_b_bs(state[-2+35].value,state[-2+39].value,state[-2+41].value,state[-2+42].value);
        const bitslice_value_t filter27_2 = f_b_bs(state[-2+44].value,state[-2+48].value,state[-2+50].value,state[-2+53].value);
        const bitslice_value_t filter27_3 = f_b_bs(state[-2+55].value,state[-2+56].value,state[-2+58].value,state[-2+60].value);
        const bitslice_value_t filter27_4 = f_a_bs(state[-2+61].value,state[-2+70].value,state[-2+71].value,state[-2+73].value);
        const bitslice_value_t filter27 = f_c_bs(filter27_0, filter27_1, filter27_2, filter27_3, filter27_4);
        results8.value &= (filter27 ^ keystream[27].value);
        if(results8.bytes64[0] == 0
           && results8.bytes64[1] == 0
           && results8.bytes64[2] == 0
           && results8.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+74].value = lfsr_bs(26);
        const bitslice_value_t filter28_0 = f_a_bs(state[-2+30].value,state[-2+31].value,state[-2+33].value,state[-2+34].value);
        const bitslice_value_t filter28_1 = f_b_bs(state[-2+36].value,state[-2+40].value,state[-2+42].value,state[-2+43].value);
        const bitslice_value_t filter28_2 = f_b_bs(state[-2+45].value,state[-2+49].value,state[-2+51].value,state[-2+54].value);
        const bitslice_value_t filter28_3 = f_b_bs(state[-2+56].value,state[-2+57].value,state[-2+59].value,state[-2+61].value);
        const bitslice_value_t filter28_4 = f_a_bs(state[-2+62].value,state[-2+71].value,state[-2+72].value,state[-2+74].value);
        const bitslice_value_t filter28 = f_c_bs(filter28_0, filter28_1, filter28_2, filter28_3, filter28_4);
        results8.value &= (filter28 ^ keystream[28].value);
        if(results8.bytes64[0] == 0
           && results8.bytes64[1] == 0
           && results8.bytes64[2] == 0
           && results8.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+75].value = lfsr_bs(27);
        const bitslice_value_t filter29_0 = f_a_bs(state[-2+31].value,state[-2+32].value,state[-2+34].value,state[-2+35].value);
        const bitslice_value_t filter29_1 = f_b_bs(state[-2+37].value,state[-2+41].value,state[-2+43].value,state[-2+44].value);
        const bitslice_value_t filter29_2 = f_b_bs(state[-2+46].value,state[-2+50].value,state[-2+52].value,state[-2+55].value);
        const bitslice_value_t filter29_3 = f_b_bs(state[-2+57].value,state[-2+58].value,state[-2+60].value,state[-2+62].value);
        const bitslice_value_t filter29_4 = f_a_bs(state[-2+63].value,state[-2+72].value,state[-2+73].value,state[-2+75].value);
        const bitslice_value_t filter29 = f_c_bs(filter29_0, filter29_1, filter29_2, filter29_3, filter29_4);
        results8.value &= (filter29 ^ keystream[29].value);
        if(results8.bytes64[0] == 0
           && results8.bytes64[1] == 0
           && results8.bytes64[2] == 0
           && results8.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+76].value = lfsr_bs(28);
        const bitslice_value_t filter30_0 = f_a_bs(state[-2+32].value,state[-2+33].value,state[-2+35].value,state[-2+36].value);
        const bitslice_value_t filter30_1 = f_b_bs(state[-2+38].value,state[-2+42].value,state[-2+44].value,state[-2+45].value);
        const bitslice_value_t filter30_2 = f_b_bs(state[-2+47].value,state[-2+51].value,state[-2+53].value,state[-2+56].value);
        const bitslice_value_t filter30_3 = f_b_bs(state[-2+58].value,state[-2+59].value,state[-2+61].value,state[-2+63].value);
        const bitslice_value_t filter30_4 = f_a_bs(state[-2+64].value,state[-2+73].value,state[-2+74].value,state[-2+76].value);
        const bitslice_value_t filter30 = f_c_bs(filter30_0, filter30_1, filter30_2, filter30_3, filter30_4);
        results8.value &= (filter30 ^ keystream[30].value);
        if(results8.bytes64[0] == 0
           && results8.bytes64[1] == 0
           && results8.bytes64[2] == 0
           && results8.bytes64[3] == 0
        ){
          continue;
        }
        state[-2+77].value = lfsr_bs(29);
        const bitslice_value_t filter31_0 = f_a_bs(state[-2+33].value,state[-2+34].value,state[-2+36].value,state[-2+37].value);
        const bitslice_value_t filter31_1 = f_b_bs(state[-2+39].value,state[-2+43].value,state[-2+45].value,state[-2+46].value);
        const bitslice_value_t filter31_2 = f_b_bs(state[-2+48].value,state[-2+52].value,state[-2+54].value,state[-2+57].value);
        const bitslice_value_t filter31_3 = f_b_bs(state[-2+59].value,state[-2+60].value,state[-2+62].value,state[-2+64].value);
        const bitslice_value_t filter31_4 = f_a_bs(state[-2+65].value,state[-2+74].value,state[-2+75].value,state[-2+77].value);
        const bitslice_value_t filter31 = f_c_bs(filter31_0, filter31_1, filter31_2, filter31_3, filter31_4);
        results8.value &= (filter31 ^ keystream[31].value);
        if(results8.bytes64[0] == 0
           && results8.bytes64[1] == 0
           && results8.bytes64[2] == 0
           && results8.bytes64[3] == 0
        ){
          continue;
        }

        for(size_t r = 0; r < MAX_BITSLICES; r++){
          if(!get_vector_bit(r,results8)) continue;
          // take the state from layer 2 so we can recover the lowest 2 bits by inverting the LFSR
          uint64_t state31 = unbitslice(&state[-2+2], r, 48);
          state31 = lfsr_inv(state31);
          state31 = lfsr_inv(state31);
          printf("Found slice %03zu: %012llx\n", r, state31&((1ull<<48)-1));
        }
      } // 8
    } // 7
  } // 6
} // 5
```

```
          } // 4
        } // 3
      } // 2
    } // 1
  } // 0
}
```

C OpenCL implementation

```
#define MAX_BITSLICES 32
#define KEYSTREAM_LENGTH 32
typedef uint bitslice_t __attribute__((aligned(MAX_BITSLICES/8)));

inline uint lut3(uint a, uint b, uint c, uint imm){
    uint r;
    asm("lop3.b32 %0, %1, %2, %3, %4;"
    : "=r" (r)
    : "r" (a), "r" (b), "r" (c), "i" (imm));
    return r;
}
#define f_a_bs_lut_1          (((0xf0|0xcc)&0xaa)^0xcc)
#define f_a_bs_lut_2          (~((0xf0|0xcc)^0xaa))
#define f_a_bs(a,b,c,d)       ((lut3(a,d,lut3(a,b,c,f_a_bs_lut_1),f_a_bs_lut_2))) // 2 luts

#define f_b_bs_lut_1          (((0xf0|0xcc)&0xaa))
#define f_b_bs_lut_2          (~((0xf0|0xcc|0xaa)))
#define f_b_bs(a,b,c,d)       ((lut3(d,c,a^b,f_b_bs_lut_1)^lut3(d,a,b, f_b_bs_lut_2))) // 2 luts, 2 xors

#define f_c_bs_lut_1          (((0xf0^0xcc)|0xaa))
#define f_c_bs_lut_2          (~((0xf0^0xcc)&(0xaa^0xcc)))

// 4 luts, 2 ands, 1 xor
#define f_c_bs(a,b,c,d,e)     (((lut3((lut3(c,e,d, f_c_bs_lut_1) & a), b, c, f_c_bs_lut_2)) ^ (lut3(d,e,a, f_c_bs_lut_1) & lut3(d,b,c,f_c_bs_lut_1))))

// non-lut version of F: 20 lookups + 6*2 + 7*3 + 13 + = 66 ops
// lut version:          20 lookups + 2*2 + 4*3 + 7 + = 43 ops

#define lfsr_lut (0xf0^0xaa^0xcc)
// 7 luts, 1 xor
#define lfsr_bs(i) ( lut3(lut3(lut3(state[-2+i+ 0], state[-2+i+ 2], state[-2+i+ 3], lfsr_lut), \
                               lut3(state[-2+i+ 6], state[-2+i+ 7], state[-2+i+ 8], lfsr_lut), \
                               lut3(state[-2+i+16], state[-2+i+22], state[-2+i+23], lfsr_lut), \
                               lfsr_lut), \
                          lut3(state[-2+i+26], state[-2+i+30], state[-2+i+41], lfsr_lut), \
                          lut3(state[-2+i+42], state[-2+i+43], state[-2+i+46], lfsr_lut), lfsr_lut) ^ state[-2+i+47])

// 46 iterations * 4 ops
inline void bitslice(bitslice_t * restrict b, ulong x, const uchar n) {
    for (uchar i = 0; i < n; ++i) {
        b[i] = -(x & 1);
        x >>= 1;
    }
}

// don't care about the complexity of this function
inline ulong unbitslice(const bitslice_t * restrict b, const uchar s, const uchar n)
{
    const bitslice_t mask = ((bitslice_t) 1) << s;
    ulong result = 0;
    for (char i = n-1; i >= 0; --i) {
        result <<= 1;
        result |= (bool) (b[i] & mask);
    }
    return result;
}

// format this array with 32 bitsliced vectors of ones and zeroes representing the inverted keystream
__constant bitslice_t keystream[KEYSTREAM_LENGTH] = %s;

__kernel
__attribute__((vec_type_hint(bitslice_t)))
void find_state(const uint candidate_index_base,
                __global const ushort * restrict candidates,
                __global ulong * restrict matches,
                __global uint * restrict matches_found)
{
  // we never actually set or use the lowest 2 bits the initial state, so we can save 2 bitslices everywhere
  bitslice_t state[-2+48+KEYSTREAM_LENGTH];
  // set bits 0+2, 0+3, 0+5, 0+6, 0+8, 0+12, 0+14, 0+15, 0+17, 0+21, 0+23, 0+26, 0+28, 0+29, 0+31, 0+33, 0+34, 0+43, 0+44, 0+46
  // get the 48-bit cipher states as 3 16-bit words from the host memory queue (to save 25% throughput)
  const uint index = 3 * (candidate_index_base + get_global_id(0)); // dimension 0 should at least keep the execution units saturated - 8k is fine
  const ulong candidate = ((ulong) candidates[index] << 32) | ((ulong) candidates[index+1] << 16) | candidates[index+2];
  // set all 48 state bits except the lowest 2
  bitslice(&state[-2+2], candidate, 46);
  // set bits 3, 6, 8, 12, 15
  state[-2+1+3] = 0xaaaaaaaa;
  state[-2+1+6] = 0xcccccccc;
  state[-2+1+8] = 0xf0f0f0f0;
  state[-2+1+12] = 0xff00ff00;
  state[-2+1+15] = 0xffff0000;
  ushort i1 = get_global_id(1); // dimension 1 should be 1024
  state[-2+18] = -((bool) (i1 & 0x1));
  state[-2+22] = -((bool) (i1 & 0x2));
  state[-2+24] = -((bool) (i1 & 0x4));
  state[-2+27] = -((bool) (i1 & 0x8));
  state[-2+30] = -((bool) (i1 & 0x10));
  state[-2+32] = -((bool) (i1 & 0x20));
  state[-2+35] = -((bool) (i1 & 0x40));
  state[-2+45] = -((bool) (i1 & 0x80));
  state[-2+47] = -((bool) (i1 & 0x100));
  state[-2+48] = -((bool) (i1 & 0x200)); // guess lfsr output 0
  // 0xfc07fef3f9fe
  const bitslice_t filter1_0 = f_a_bs(state[-2+3],state[-2+4],state[-2+6],state[-2+7]);
  const bitslice_t filter1_1 = f_b_bs(state[-2+9],state[-2+13],state[-2+15],state[-2+16]);
```

```
const bitslice_t filter1_2 = f_b_bs(state[-2+18],state[-2+22],state[-2+24],state[-2+27]);
const bitslice_t filter1_3 = f_b_bs(state[-2+29],state[-2+30],state[-2+32],state[-2+34]);
const bitslice_t filter1_4 = f_a_bs(state[-2+35],state[-2+44],state[-2+45],state[-2+47]);
const bitslice_t filter1 = f_c_bs(filter1_0, filter1_1, filter1_2, filter1_3, filter1_4);
const bitslice_t results1 = filter1 ^ keystream[1];
if(!results1) return;
const bitslice_t filter2_0 = f_a_bs(state[-2+4],state[-2+5],state[-2+7],state[-2+8]);
const bitslice_t filter2_3 = f_b_bs(state[-2+30],state[-2+31],state[-2+33],state[-2+35]);
const bitslice_t filter3_0 = f_a_bs(state[-2+5],state[-2+6],state[-2+8],state[-2+9]);
const bitslice_t filter5_2 = f_b_bs(state[-2+22],state[-2+26],state[-2+28],state[-2+31]);
const bitslice_t filter6_2 = f_b_bs(state[-2+23],state[-2+27],state[-2+29],state[-2+32]);
const bitslice_t filter7_2 = f_b_bs(state[-2+24],state[-2+28],state[-2+30],state[-2+33]);
const bitslice_t filter9_1 = f_b_bs(state[-2+17],state[-2+21],state[-2+23],state[-2+24]);
const bitslice_t filter9_2 = f_b_bs(state[-2+26],state[-2+30],state[-2+32],state[-2+35]);
const bitslice_t filter10_0 = f_a_bs(state[-2+12],state[-2+13],state[-2+15],state[-2+16]);
const bitslice_t filter11_0 = f_a_bs(state[-2+13],state[-2+14],state[-2+16],state[-2+17]);
const bitslice_t filter12_0 = f_a_bs(state[-2+14],state[-2+15],state[-2+17],state[-2+18]);
const bitslice_t filter14_1 = f_b_bs(state[-2+22],state[-2+26],state[-2+28],state[-2+29]);
const bitslice_t filter15_1 = f_b_bs(state[-2+23],state[-2+27],state[-2+29],state[-2+30]);
const bitslice_t filter15_3 = f_b_bs(state[-2+43],state[-2+44],state[-2+46],state[-2+48]);
const bitslice_t filter16_1 = f_b_bs(state[-2+24],state[-2+28],state[-2+30],state[-2+31]);
for(uchar i2 = 0; i2 < (1<<5);){
  state[-2+10] = -((bool) (i2 & 0x1));
  state[-2+19] = -((bool) (i2 & 0x2));
  state[-2+25] = -((bool) (i2 & 0x4));
  state[-2+36] = -((bool) (i2 & 0x8));
  state[-2+49] = -((bool) (i2 & 0x10)); // guess lfsr output 1
  i2++;
  // 0xfe07fffbfdff
  const bitslice_t filter2_1 = f_b_bs(state[-2+10],state[-2+14],state[-2+16],state[-2+17]);
  const bitslice_t filter2_2 = f_b_bs(state[-2+19],state[-2+23],state[-2+25],state[-2+28]);
  const bitslice_t filter2_4 = f_a_bs(state[-2+36],state[-2+45],state[-2+46],state[-2+48]);
  const bitslice_t filter2 = f_c_bs(filter2_0, filter2_1, filter2_2, filter2_3, filter2_4);
  const bitslice_t results2 = results1 & (filter2 ^ keystream[2]);
  if(!results2) continue;
  state[-2+50] = lfsr_bs(2);
  const bitslice_t filter3_3 = f_b_bs(state[-2+31],state[-2+32],state[-2+34],state[-2+36]);
  const bitslice_t filter4_0 = f_a_bs(state[-2+6],state[-2+7],state[-2+9],state[-2+10]);
  const bitslice_t filter4_1 = f_b_bs(state[-2+12],state[-2+16],state[-2+18],state[-2+19]);
  const bitslice_t filter4_2 = f_b_bs(state[-2+21],state[-2+25],state[-2+27],state[-2+30]);
  const bitslice_t filter7_0 = f_a_bs(state[-2+9],state[-2+10],state[-2+12],state[-2+13]);
  const bitslice_t filter7_1 = f_b_bs(state[-2+15],state[-2+19],state[-2+21],state[-2+22]);
  const bitslice_t filter8_2 = f_b_bs(state[-2+25],state[-2+29],state[-2+31],state[-2+34]);
  const bitslice_t filter10_1 = f_b_bs(state[-2+18],state[-2+22],state[-2+24],state[-2+25]);
  const bitslice_t filter10_2 = f_b_bs(state[-2+27],state[-2+31],state[-2+33],state[-2+36]);
  const bitslice_t filter11_1 = f_b_bs(state[-2+19],state[-2+23],state[-2+25],state[-2+26]);
  const bitslice_t filter13_0 = f_a_bs(state[-2+15],state[-2+16],state[-2+18],state[-2+19]);
  const bitslice_t filter13_1 = f_b_bs(state[-2+21],state[-2+25],state[-2+27],state[-2+28]);
  const bitslice_t filter16_0 = f_a_bs(state[-2+18],state[-2+19],state[-2+21],state[-2+22]);
  const bitslice_t filter16_3 = f_b_bs(state[-2+44],state[-2+45],state[-2+47],state[-2+49]);
  const bitslice_t filter17_1 = f_b_bs(state[-2+25],state[-2+29],state[-2+31],state[-2+32]);
  const bitslice_t filter17_3 = f_b_bs(state[-2+45],state[-2+46],state[-2+48],state[-2+50]);
  for(uchar i3 = 0; i3 < (1<<3);){
    state[-2+11] = -((bool) (i3 & 0x1));
    state[-2+20] = -((bool) (i3 & 0x2));
    state[-2+37] = -((bool) (i3 & 0x4));
    i3++;
    // 0xff07ffffffff
    const bitslice_t filter3_1 = f_b_bs(state[-2+11],state[-2+15],state[-2+17],state[-2+18]);
    const bitslice_t filter3_2 = f_b_bs(state[-2+20],state[-2+24],state[-2+26],state[-2+29]);
    const bitslice_t filter3_4 = f_a_bs(state[-2+37],state[-2+46],state[-2+47],state[-2+49]);
    const bitslice_t filter3 = f_c_bs(filter3_0, filter3_1, filter3_2, filter3_3, filter3_4);
    const bitslice_t results3 = results2 & (filter3 ^ keystream[3]);
    if(!results3) continue;
    state[-2+51] = lfsr_bs(3);
    state[-2+52] = lfsr_bs(4);
    state[-2+53] = lfsr_bs(5);
    state[-2+54] = lfsr_bs(6);
    state[-2+55] = lfsr_bs(7);
    const bitslice_t filter4_3 = f_b_bs(state[-2+32],state[-2+33],state[-2+35],state[-2+37]);
    const bitslice_t filter5_0 = f_a_bs(state[-2+7],state[-2+8],state[-2+10],state[-2+11]);
    const bitslice_t filter5_1 = f_b_bs(state[-2+13],state[-2+17],state[-2+19],state[-2+20]);
    const bitslice_t filter6_0 = f_a_bs(state[-2+8],state[-2+9],state[-2+11],state[-2+12]);
    const bitslice_t filter6_1 = f_b_bs(state[-2+14],state[-2+18],state[-2+20],state[-2+21]);
    const bitslice_t filter8_0 = f_a_bs(state[-2+10],state[-2+11],state[-2+13],state[-2+14]);
    const bitslice_t filter8_1 = f_b_bs(state[-2+16],state[-2+20],state[-2+22],state[-2+23]);
    const bitslice_t filter9_0 = f_a_bs(state[-2+11],state[-2+12],state[-2+14],state[-2+15]);
    const bitslice_t filter9_4 = f_a_bs(state[-2+43],state[-2+52],state[-2+53],state[-2+55]);
    const bitslice_t filter11_2 = f_b_bs(state[-2+28],state[-2+32],state[-2+34],state[-2+37]);
    const bitslice_t filter12_1 = f_b_bs(state[-2+20],state[-2+24],state[-2+26],state[-2+27]);
    const bitslice_t filter14_0 = f_a_bs(state[-2+16],state[-2+17],state[-2+19],state[-2+20]);
    const bitslice_t filter15_0 = f_a_bs(state[-2+17],state[-2+18],state[-2+20],state[-2+21]);
    const bitslice_t filter17_0 = f_a_bs(state[-2+19],state[-2+20],state[-2+22],state[-2+23]);
    for(uchar i4 = 0; i4 < (1<<1);){
      state[-2+38] = -i4;
      i4++;
      // 0xff87ffffffff
      const bitslice_t filter4_4 = f_a_bs(state[-2+38],state[-2+47],state[-2+48],state[-2+50]);
      const bitslice_t filter4 = f_c_bs(filter4_0, filter4_1, filter4_2, filter4_3, filter4_4);
      const bitslice_t results4 = results3 & (filter4 ^ keystream[4]);
      if(!results4) continue;
      state[-2+56] = lfsr_bs(8);
      const bitslice_t filter5_3 = f_b_bs(state[-2+33],state[-2+34],state[-2+36],state[-2+38]);
      const bitslice_t filter10_4 = f_a_bs(state[-2+44],state[-2+53],state[-2+54],state[-2+56]);
      const bitslice_t filter12_2 = f_b_bs(state[-2+29],state[-2+33],state[-2+35],state[-2+38]);
      for(uchar i5 = 0; i5 < (1<<1);){
        state[-2+39] = -i5;
        i5++;
        // 0xffc7ffffffff
        const bitslice_t filter5_4 = f_a_bs(state[-2+39],state[-2+48],state[-2+49],state[-2+51]);
        const bitslice_t filter5 = f_c_bs(filter5_0, filter5_1, filter5_2, filter5_3, filter5_4);
        const bitslice_t results5 = results4 & (filter5 ^ keystream[5]);
        if(!results5) continue;
        state[-2+57] = lfsr_bs(9);
        const bitslice_t filter6_3 = f_b_bs(state[-2+34],state[-2+35],state[-2+37],state[-2+39]);
        const bitslice_t filter11_4 = f_a_bs(state[-2+45],state[-2+54],state[-2+55],state[-2+57]);
        const bitslice_t filter13_2 = f_b_bs(state[-2+30],state[-2+34],state[-2+36],state[-2+39]);
```

```
for(uchar i6 = 0; i6 < (1<<1);){
  state[-2+40] = -i6;
  i6++;
  // 0xffe7ffffffff
  const bitslice_t filter6_4 = f_a_bs(state[-2+40],state[-2+49],state[-2+50],state[-2+52]);
  const bitslice_t filter6 = f_c_bs(filter6_0, filter6_1, filter6_2, filter6_3, filter6_4);
  const bitslice_t results6 = results5 & (filter6 ^ keystream[6]);
  if(!results6) continue;
  state[-2+58] = lfsr_bs(10);
  const bitslice_t filter7_3 = f_b_bs(state[-2+35],state[-2+36],state[-2+38],state[-2+40]);
  const bitslice_t filter12_4 = f_a_bs(state[-2+46],state[-2+55],state[-2+56],state[-2+58]);
  const bitslice_t filter14_2 = f_b_bs(state[-2+31],state[-2+35],state[-2+37],state[-2+40]);
  const bitslice_t filter17_2 = f_b_bs(state[-2+34],state[-2+38],state[-2+40],state[-2+43]);
  #pragma unroll
  for(uchar i7 = 0; i7 < (1<<1);){
    state[-2+41] = -i7;
    i7++;
    // 0xfff7ffffffff
    const bitslice_t filter7_4 = f_a_bs(state[-2+41],state[-2+50],state[-2+51],state[-2+53]);
    const bitslice_t filter7 = f_c_bs(filter7_0, filter7_1, filter7_2, filter7_3, filter7_4);
    const bitslice_t results7 = results6 & (filter7 ^ keystream[7]);
    if(!results7) continue;
    state[-2+59] = lfsr_bs(11);
    const bitslice_t filter8_3 = f_b_bs(state[-2+36],state[-2+37],state[-2+39],state[-2+41]);
    const bitslice_t filter10_3 = f_b_bs(state[-2+38],state[-2+39],state[-2+41],state[-2+43]);
    const bitslice_t filter10 = f_c_bs(filter10_0, filter10_1, filter10_2, filter10_3, filter10_4);
    const bitslice_t filter12_3 = f_b_bs(state[-2+40],state[-2+41],state[-2+43],state[-2+45]);
    const bitslice_t filter12 = f_c_bs(filter12_0, filter12_1, filter12_2, filter12_3, filter12_4);
    const bitslice_t filter13_4 = f_a_bs(state[-2+47],state[-2+56],state[-2+57],state[-2+59]);
    const bitslice_t filter15_2 = f_b_bs(state[-2+32],state[-2+36],state[-2+38],state[-2+41]);
    #pragma unroll
    for(uchar i8 = 0; i8 < (1<<1);){
      state[-2+42] = -i8;
      i8++;
      // 0xffffffffffff
      const bitslice_t filter8_4 = f_a_bs(state[-2+42],state[-2+51],state[-2+52],state[-2+54]);
      const bitslice_t filter8 = f_c_bs(filter8_0, filter8_1, filter8_2, filter8_3, filter8_4);
      bitslice_t results8 = results7 & (filter8 ^ keystream[8]);
      if(!results8) continue;
      const bitslice_t filter9_3 = f_b_bs(state[-2+37],state[-2+38],state[-2+40],state[-2+42]);
      const bitslice_t filter9 = f_c_bs(filter9_0, filter9_1, filter9_2, filter9_3, filter9_4);
      results8 &= (filter9 ^ keystream[9]);
      if(!results8) continue;
      results8 &= (filter10 ^ keystream[10]);
      if(!results8) continue;
      const bitslice_t filter11_3 = f_b_bs(state[-2+39],state[-2+40],state[-2+42],state[-2+44]);
      const bitslice_t filter11 = f_c_bs(filter11_0, filter11_1, filter11_2, filter11_3, filter11_4);
      results8 &= (filter11 ^ keystream[11]);
      if(!results8) continue;
      results8 &= (filter12 ^ keystream[12]);
      if(!results8) continue;
      const bitslice_t filter13_3 = f_b_bs(state[-2+41],state[-2+42],state[-2+44],state[-2+46]);
      const bitslice_t filter13 = f_c_bs(filter13_0, filter13_1, filter13_2, filter13_3, filter13_4);
      results8 &= (filter13 ^ keystream[13]);
      if(!results8) continue;
      state[-2+60] = lfsr_bs(12);
      const bitslice_t filter14_3 = f_b_bs(state[-2+42],state[-2+43],state[-2+45],state[-2+47]);
      const bitslice_t filter14_4 = f_a_bs(state[-2+48],state[-2+57],state[-2+58],state[-2+60]);
      const bitslice_t filter14 = f_c_bs(filter14_0, filter14_1, filter14_2, filter14_3, filter14_4);
      results8 &= (filter14 ^ keystream[14]);
      if(!results8) continue;
      state[-2+61] = lfsr_bs(13);
      const bitslice_t filter15_4 = f_a_bs(state[-2+49],state[-2+58],state[-2+59],state[-2+61]);
      const bitslice_t filter15 = f_c_bs(filter15_0, filter15_1, filter15_2, filter15_3, filter15_4);
      results8 &= (filter15 ^ keystream[15]);
      if(!results8) continue;
      state[-2+62] = lfsr_bs(14);
      const bitslice_t filter16_2 = f_b_bs(state[-2+33],state[-2+37],state[-2+39],state[-2+42]);
      const bitslice_t filter16_4 = f_a_bs(state[-2+50],state[-2+59],state[-2+60],state[-2+62]);
      const bitslice_t filter16 = f_c_bs(filter16_0, filter16_1, filter16_2, filter16_3, filter16_4);
      results8 &= (filter16 ^ keystream[16]);
      if(!results8) continue;
      state[-2+63] = lfsr_bs(15);
      const bitslice_t filter17_4 = f_a_bs(state[-2+51],state[-2+60],state[-2+61],state[-2+63]);
      const bitslice_t filter17 = f_c_bs(filter17_0, filter17_1, filter17_2, filter17_3, filter17_4);
      results8 &= (filter17 ^ keystream[17]);
      if(!results8) continue;
      state[-2+64] = lfsr_bs(16);
      const bitslice_t filter18_0 = f_a_bs(state[-2+20],state[-2+21],state[-2+23],state[-2+24]);
      const bitslice_t filter18_1 = f_b_bs(state[-2+26],state[-2+30],state[-2+32],state[-2+33]);
      const bitslice_t filter18_2 = f_b_bs(state[-2+35],state[-2+39],state[-2+41],state[-2+44]);
      const bitslice_t filter18_3 = f_b_bs(state[-2+46],state[-2+47],state[-2+49],state[-2+51]);
      const bitslice_t filter18_4 = f_a_bs(state[-2+52],state[-2+61],state[-2+62],state[-2+64]);
      const bitslice_t filter18 = f_c_bs(filter18_0, filter18_1, filter18_2, filter18_3, filter18_4);
      results8 &= (filter18 ^ keystream[18]);
      if(!results8) continue;
      state[-2+65] = lfsr_bs(17);
      const bitslice_t filter19_0 = f_a_bs(state[-2+21],state[-2+22],state[-2+24],state[-2+25]);
      const bitslice_t filter19_1 = f_b_bs(state[-2+27],state[-2+31],state[-2+33],state[-2+34]);
      const bitslice_t filter19_2 = f_b_bs(state[-2+36],state[-2+40],state[-2+42],state[-2+45]);
      const bitslice_t filter19_3 = f_b_bs(state[-2+47],state[-2+48],state[-2+50],state[-2+52]);
      const bitslice_t filter19_4 = f_a_bs(state[-2+53],state[-2+62],state[-2+63],state[-2+65]);
      const bitslice_t filter19 = f_c_bs(filter19_0, filter19_1, filter19_2, filter19_3, filter19_4);
      results8 &= (filter19 ^ keystream[19]);
      if(!results8) continue;
      state[-2+66] = lfsr_bs(18);
      const bitslice_t filter20_0 = f_a_bs(state[-2+22],state[-2+23],state[-2+25],state[-2+26]);
      const bitslice_t filter20_1 = f_b_bs(state[-2+28],state[-2+32],state[-2+34],state[-2+35]);
      const bitslice_t filter20_2 = f_b_bs(state[-2+37],state[-2+41],state[-2+43],state[-2+46]);
      const bitslice_t filter20_3 = f_b_bs(state[-2+48],state[-2+49],state[-2+51],state[-2+53]);
      const bitslice_t filter20_4 = f_a_bs(state[-2+54],state[-2+63],state[-2+64],state[-2+66]);
      const bitslice_t filter20 = f_c_bs(filter20_0, filter20_1, filter20_2, filter20_3, filter20_4);
      results8 &= (filter20 ^ keystream[20]);
      if(!results8) continue;
      state[-2+67] = lfsr_bs(19);
      const bitslice_t filter21_0 = f_a_bs(state[-2+23],state[-2+24],state[-2+26],state[-2+27]);
      const bitslice_t filter21_1 = f_b_bs(state[-2+29],state[-2+33],state[-2+35],state[-2+36]);
```

```
    const bitslice_t filter21_2 = f_b_bs(state[-2+38],state[-2+42],state[-2+44],state[-2+47]);
    const bitslice_t filter21_3 = f_b_bs(state[-2+49],state[-2+50],state[-2+52],state[-2+54]);
    const bitslice_t filter21_4 = f_a_bs(state[-2+55],state[-2+64],state[-2+65],state[-2+67]);
    const bitslice_t filter21 = f_c_bs(filter21_0, filter21_1, filter21_2, filter21_3, filter21_4);
    results8 &= (filter21 ^ keystream[21]);
    if(!results8) continue;
    state[-2+68] = lfsr_bs(20);
    const bitslice_t filter22_0 = f_a_bs(state[-2+24],state[-2+25],state[-2+27],state[-2+28]);
    const bitslice_t filter22_1 = f_b_bs(state[-2+30],state[-2+34],state[-2+36],state[-2+37]);
    const bitslice_t filter22_2 = f_b_bs(state[-2+39],state[-2+43],state[-2+45],state[-2+48]);
    const bitslice_t filter22_3 = f_b_bs(state[-2+50],state[-2+51],state[-2+53],state[-2+55]);
    const bitslice_t filter22_4 = f_a_bs(state[-2+56],state[-2+65],state[-2+66],state[-2+68]);
    const bitslice_t filter22 = f_c_bs(filter22_0, filter22_1, filter22_2, filter22_3, filter22_4);
    results8 &= (filter22 ^ keystream[22]);
    if(!results8) continue;
    state[-2+69] = lfsr_bs(21);
    const bitslice_t filter23_0 = f_a_bs(state[-2+25],state[-2+26],state[-2+28],state[-2+29]);
    const bitslice_t filter23_1 = f_b_bs(state[-2+31],state[-2+35],state[-2+37],state[-2+38]);
    const bitslice_t filter23_2 = f_b_bs(state[-2+40],state[-2+44],state[-2+46],state[-2+49]);
    const bitslice_t filter23_3 = f_b_bs(state[-2+51],state[-2+52],state[-2+54],state[-2+56]);
    const bitslice_t filter23_4 = f_a_bs(state[-2+57],state[-2+66],state[-2+67],state[-2+69]);
    const bitslice_t filter23 = f_c_bs(filter23_0, filter23_1, filter23_2, filter23_3, filter23_4);
    results8 &= (filter23 ^ keystream[23]);
    if(!results8) continue;
    state[-2+70] = lfsr_bs(22);
    const bitslice_t filter24_0 = f_a_bs(state[-2+26],state[-2+27],state[-2+29],state[-2+30]);
    const bitslice_t filter24_1 = f_b_bs(state[-2+32],state[-2+36],state[-2+38],state[-2+39]);
    const bitslice_t filter24_2 = f_b_bs(state[-2+41],state[-2+45],state[-2+47],state[-2+50]);
    const bitslice_t filter24_3 = f_b_bs(state[-2+52],state[-2+53],state[-2+55],state[-2+57]);
    const bitslice_t filter24_4 = f_a_bs(state[-2+58],state[-2+67],state[-2+68],state[-2+70]);
    const bitslice_t filter24 = f_c_bs(filter24_0, filter24_1, filter24_2, filter24_3, filter24_4);
    results8 &= (filter24 ^ keystream[24]);
    if(!results8) continue;
    state[-2+71] = lfsr_bs(23);
    const bitslice_t filter25_0 = f_a_bs(state[-2+27],state[-2+28],state[-2+30],state[-2+31]);
    const bitslice_t filter25_1 = f_b_bs(state[-2+33],state[-2+37],state[-2+39],state[-2+40]);
    const bitslice_t filter25_2 = f_b_bs(state[-2+42],state[-2+46],state[-2+48],state[-2+51]);
    const bitslice_t filter25_3 = f_b_bs(state[-2+53],state[-2+54],state[-2+56],state[-2+58]);
    const bitslice_t filter25_4 = f_a_bs(state[-2+59],state[-2+68],state[-2+69],state[-2+71]);
    const bitslice_t filter25 = f_c_bs(filter25_0, filter25_1, filter25_2, filter25_3, filter25_4);
    results8 &= (filter25 ^ keystream[25]);
    if(!results8) continue;
    state[-2+72] = lfsr_bs(24);
    const bitslice_t filter26_0 = f_a_bs(state[-2+28],state[-2+29],state[-2+31],state[-2+32]);
    const bitslice_t filter26_1 = f_b_bs(state[-2+34],state[-2+38],state[-2+40],state[-2+41]);
    const bitslice_t filter26_2 = f_b_bs(state[-2+43],state[-2+47],state[-2+49],state[-2+52]);
    const bitslice_t filter26_3 = f_b_bs(state[-2+54],state[-2+55],state[-2+57],state[-2+59]);
    const bitslice_t filter26_4 = f_a_bs(state[-2+60],state[-2+69],state[-2+70],state[-2+72]);
    const bitslice_t filter26 = f_c_bs(filter26_0, filter26_1, filter26_2, filter26_3, filter26_4);
    results8 &= (filter26 ^ keystream[26]);
    if(!results8) continue;
    state[-2+73] = lfsr_bs(25);
    const bitslice_t filter27_0 = f_a_bs(state[-2+29],state[-2+30],state[-2+32],state[-2+33]);
    const bitslice_t filter27_1 = f_b_bs(state[-2+35],state[-2+39],state[-2+41],state[-2+42]);
    const bitslice_t filter27_2 = f_b_bs(state[-2+44],state[-2+48],state[-2+50],state[-2+53]);
    const bitslice_t filter27_3 = f_b_bs(state[-2+55],state[-2+56],state[-2+58],state[-2+60]);
    const bitslice_t filter27_4 = f_a_bs(state[-2+61],state[-2+70],state[-2+71],state[-2+73]);
    const bitslice_t filter27 = f_c_bs(filter27_0, filter27_1, filter27_2, filter27_3, filter27_4);
    results8 &= (filter27 ^ keystream[27]);
    if(!results8) continue;
    state[-2+74] = lfsr_bs(26);
    const bitslice_t filter28_0 = f_a_bs(state[-2+30],state[-2+31],state[-2+33],state[-2+34]);
    const bitslice_t filter28_1 = f_b_bs(state[-2+36],state[-2+40],state[-2+42],state[-2+43]);
    const bitslice_t filter28_2 = f_b_bs(state[-2+45],state[-2+49],state[-2+51],state[-2+54]);
    const bitslice_t filter28_3 = f_b_bs(state[-2+56],state[-2+57],state[-2+59],state[-2+61]);
    const bitslice_t filter28_4 = f_a_bs(state[-2+62],state[-2+71],state[-2+72],state[-2+74]);
    const bitslice_t filter28 = f_c_bs(filter28_0, filter28_1, filter28_2, filter28_3, filter28_4);
    results8 &= (filter28 ^ keystream[28]);
    if(!results8) continue;
    state[-2+75] = lfsr_bs(27);
    const bitslice_t filter29_0 = f_a_bs(state[-2+31],state[-2+32],state[-2+34],state[-2+35]);
    const bitslice_t filter29_1 = f_b_bs(state[-2+37],state[-2+41],state[-2+43],state[-2+44]);
    const bitslice_t filter29_2 = f_b_bs(state[-2+46],state[-2+50],state[-2+52],state[-2+55]);
    const bitslice_t filter29_3 = f_b_bs(state[-2+57],state[-2+58],state[-2+60],state[-2+62]);
    const bitslice_t filter29_4 = f_a_bs(state[-2+63],state[-2+72],state[-2+73],state[-2+75]);
    const bitslice_t filter29 = f_c_bs(filter29_0, filter29_1, filter29_2, filter29_3, filter29_4);
    results8 &= (filter29 ^ keystream[29]);
    if(!results8) continue;
    state[-2+76] = lfsr_bs(28);
    const bitslice_t filter30_0 = f_a_bs(state[-2+32],state[-2+33],state[-2+35],state[-2+36]);
    const bitslice_t filter30_1 = f_b_bs(state[-2+38],state[-2+42],state[-2+44],state[-2+45]);
    const bitslice_t filter30_2 = f_b_bs(state[-2+47],state[-2+51],state[-2+53],state[-2+56]);
    const bitslice_t filter30_3 = f_b_bs(state[-2+58],state[-2+59],state[-2+61],state[-2+63]);
    const bitslice_t filter30_4 = f_a_bs(state[-2+64],state[-2+73],state[-2+74],state[-2+76]);
    const bitslice_t filter30 = f_c_bs(filter30_0, filter30_1, filter30_2, filter30_3, filter30_4);
    results8 &= (filter30 ^ keystream[30]);
    if(!results8) continue;
    state[-2+77] = lfsr_bs(29);
    const bitslice_t filter31_0 = f_a_bs(state[-2+33],state[-2+34],state[-2+36],state[-2+37]);
    const bitslice_t filter31_1 = f_b_bs(state[-2+39],state[-2+43],state[-2+45],state[-2+46]);
    const bitslice_t filter31_2 = f_b_bs(state[-2+48],state[-2+52],state[-2+54],state[-2+57]);
    const bitslice_t filter31_3 = f_b_bs(state[-2+59],state[-2+60],state[-2+62],state[-2+64]);
    const bitslice_t filter31_4 = f_a_bs(state[-2+65],state[-2+74],state[-2+75],state[-2+77]);
    const bitslice_t filter31 = f_c_bs(filter31_0, filter31_1, filter31_2, filter31_3, filter31_4);
    results8 &= (filter31 ^ keystream[31]);
    if(!results8) continue;
    uchar match_index = 0;
    // Save results
    while(results8 && (match_index < MAX_BITSLICES)){
        uchar shift = clz(results8)+1;
        match_index += shift;
        // take the state from layer 2 so we can recover the lowest 2 bits on the host by inverting the LFSR
        matches[atomic_inc(matches_found)] = unbitslice(&state[-2+2], MAX_BITSLICES-match_index, 48);
        results8 <<= shift;
    }
  } // 8
} // 7
```

```
        } // 6
      } // 5
    } // 4
  } // 3
} // 2
} // 1
```