Report for the Course on Cyber-Physical Systems and IoT Security

Title of the reference paper: Lock It and Still Lose It – On the (In)Security of Automotive Remote Keyless Entry Systems – Garcia et al. [1]

Name of who is involved in the project: *Jan Clemens, Max Opperman, Valentin Maier*

1 Objectives

The objective of this work was to implement the attack on Hitag2 described in the work of Garcia et al. [1] in a simulated way that works by generating artificial traces and then trying to determine the key used in the Hitag2 process. This means the real-life aspects, so the trace recording of the key fob and also the crafting of packets with the determined key to open the car were excluded. The project solely covers the implementation of the attack. The results are accessible in the following git repo: *https://github.com/jackiecan/hitag2*

2 System Setup

Since we were not aiming to reproduce the attack in the same time as the original paper (~1 min) we decided to implement the attack in python. This makes the implementation better readable for the average research community. To still speed up the search a little bit we ran the script with pypy (pypy3.11-v7.3.20-win64) [2] which provides a python-runtime based on Just-In-Time compilation. Because we mainly use plain python code, we can achieve a huge speedup like this. To use pypy, the program is simply started with one of the pypy-python exe-files. The configuration of the attack is completely done in the main function of the attack file `attack.py`.
The attack can be run by simply executing the following command in the project folder, assuming the pypy folder is moved there:

```
$> pypy3.11-v7.3.20-win64\python.exe attack.py
```

To get Hitag2 traces we used the python implementation by Verstegen et al. `hitag2.py` [3] that was produced through another research effort to break Hitag2 [4].
Our attack-runs/experiments were conducted on a standard consumer-grade laptop equipped with an 8 core AMD Ryzen 7 5800U processor, 16 GB of RAM, and running Windows 11 Pro.
While we used AI tools like GitHub Copilot and Gemini to help write the code, their role was limited to generating specific, predefined snippets. They did not design the attack or handle the core logic, as the topic was far too complex. All conceptual thinking, as well as the granular bit-level debugging of variables, was done manually.

3 Experiments/Results

Since there is no public implementation of this attack, we tried to implement the attack by the descriptions in the paper ourself. This is why we merged the experiments and results chapter because the implementation is our main result. The following explanation goes alongside with the code and explains how we understood and implemented the attack. First, we go over the Hitag2 cipher itself, then the Attack and in the end, we will cover two example runs of the attack. Sadly, the attack didn't work fully, so we will discuss possible problems in the end.
We implemented the attack in a simulated approach, where we assumed to have many traces and knowledge about the full 28-bit counter value. In a real-life approach, we would only have access

to the 10 least significant bits of the counter which is relevant to generate the IV used in the cipher. But according to Garcia et al. this is legitimate because the upper bits are normally very small and can be tried out through running the attack multiple times. Our approach could be connected to a real-life trace recording of the key-fob data easily.

3.1 Hitag2

We started by examining the Hitag2 implementation we used. We removed some unnecessary code and created a version (`hitag_print.py`) that shows how the variables lie in the LFSR during the Hitag2 *Initialization* and *Key-Generation* steps. The file used for the attack (`hitag.py`) is commented and explains the cipher in combination with this explanation here. It is very important to know how variables are oriented in the LFSR and which bit is where exactly, because a simple one-bit-off error can completely prevent the attack from working without necessary being visible.

```
Using nonce: 00010010001101000101011001111000 | 0x12345678 | 305419896
inserting key:
  key:                        000100100011010001010110011110001001101010111100
  part of key to be inserted: 0001001000110100
  part of key reversed:       0010110001001000
   state: 000000000000000000000000000000000000000000000000
   state: 000000000000000000000000000000000000000000000000
   state: 000000000000000000000000000000000000000000000001
        ...
   state: 000000000000000000000000000000000001011000100100
   state: 000000000000000000000000000000000010110001001000
inserting uid:
  uid to be inserted: 00010010001101000101011001111000
  uid reversed:       00011110011010100010110001001000
   state: 000000000000000000000000000000000101100010010000
   state: 000000000000000000000000000000001011000100100000
   state: 000000000000000000000000000000010110001001000000
   state: 000000000000000000000000000000101100010010000001
        ...
   state: 001011000100100000011110011010100010110001001000
state: 001011000100100000011110011010100010110001001000
       |    key_rev    ||      uid_rev                  |
```

Figure 1 Pre-Initialization of LFSR example

The implementation by Verstegen et al. is mostly compatible with the descriptions of Garcia et al. but they consider the LFSR mirrored. This is not really a problem but has to be considered when visualizing the LFSR and especially when analyzing the lookup tables for the functions f_a, f_b, f_c. We see that they are different because of the "reversed" bit layout. The actual internal processes and also the mathematical definitions of the attack are still compatible, we just have to consider correctly where which bit lays. When we speak of "right" and "left" in our implementation and explanations we refer to the visualization of the LFSR as in **Figure 2** or **Figure 4** (Compared to Garcia et al. this is exactly the other way around).

From the output of `hitag_print.py` (`example.txt`) we see how the cipher works. Hitag2 consists of two phases: *Initialization* and *Keystream generation*. Before the initialization, the upper 16 bits of the key (most significant bits: $k_0..k_{15}$) alongside with the UID are shifted into the LFSR. Because the most significant bits are shifted in the LFSR first from the right-side, the key part and UID seem to lay reversed in the LFSR, but of course this is just dependent on the way you look at it. This happens in the first part of the `hitag2_init(…)` function. This procedure and its outcome are visualized by the output of the print version of the code seen in **Figure 1**.

Now and the real Initialization starts where new bits are introduced on the left side of the LFSR though estimating the `f20`-function and xoring its result (b) with the corresponding IV-bit and the other key bits ($k_{16}..k_{47}$), see **Figure 2**. This then goes on for 32 rounds in which the LFSR is shifted to the right and the newly generated bits are inserted on the left. How this impacts the bits in the LFSR is also demonstrated in **Figure 3** and again explained in the second part of the function `hitag2_init(…)`. The filter function `f20` used in the cipher is implemented by a truth table lookup and combines multiple smaller functions f_a, f_b, f_c. On the example f_a we see that the input bits in the function determine a bit-index of the value `0x3c65` where the bitvalue on that index corresponds to the output of the function.

```
def f20_0(state):
    return ((0x3c65 >> i4(state, 2, 3, 5, 6)) & 1)
```

The value (`0x3c65`) is different than in the paper (`0xA63C`) because the orientation of the bits while forming the index is flipped. Still, they implement the same functionality.

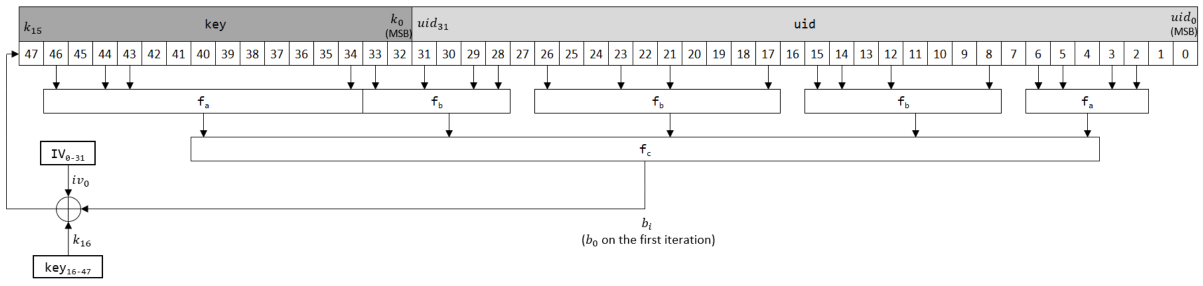

Figure 2 Hitag2 Initialization Process (based on [5])

```
   nonce:         00010010001101000101011001111000
   key:           000100100011010001010110011110001001101010111100
   key part used:                 01010110011110001001101010111100
state: 001011000100100000011110011010100010110001001000 | f20: 1, nonce bit: 0 | key bit: 0
state: 100101100010010000001111001101010001011000100100 | f20: 1, nonce bit: 0 | key bit: 1
state: 010010110001001000000111100110101000101100010010 | f20: 0, nonce bit: 0 | key bit: 0
state: 001001011000100100000011110011010100010110001001 | f20: 1, nonce bit: 1 | key bit: 1
state: 100100101100010010000001111001101010001011000100 | f20: 1, nonce bit: 0 | key bit: 0
state: 110010010110001001000000111100110101000101100010 | f20: 1, nonce bit: 0 | key bit: 1
state: 011001001011000100100000011110011010100010110001 | f20: 1, nonce bit: 1 | key bit: 1
...
state: 011111000110100011000001101100100101100010010000 | f20: 1, nonce bit: 0 | key bit: 0
final init state: 101111100011010001100000110110010010110001001000
                                                ||     key_rev    |
```

Figure 3 Initialization Process example

This finally leaves us in the state where the key stream starts being extracted, see **Figure 4**. It is important to note that a portion of the key resides directly in the LFSR which will be later used to start the attack. The keystream is generated by simply running `f20` on the LFSR and shifting the LFSR to the right while creating a new input bit that is inserted on the left by xoring the bits on the tapped positions. The first generated keystream bit becomes the MSB in the keystream. This procedure is also shown in **Figure 5** and implemented in the function `hitag2(…)`.

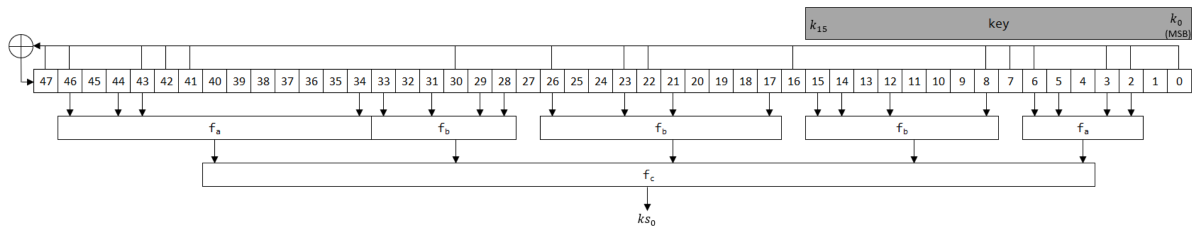

Figure 4 Hitag2 Keystream generation (based on [5])

```
                                       |    key_rev    |
state: 1011111000110100011000001101100100101100010010000 -> 0 | 0
state: 010111110001101000110000011011001001011000100100 -> 1 | 1
state: 001011111000110100011000001101100100101100010010 -> 0 | 2
state: 000101111100011010001100000110110010010110001001 -> 0 | 4
...
state: 0000101111100000101011000110000010111110001101000 -> 1 | 1085563609
keystream: 01000000101101000110001011011001 | 0x40b462d9 | 1085563609
```
Figure 5 Keystream generation example

3.2 Attack

So far, we only covered the Hitag2 implementation which it is very important to understand in detail first, because the attack works very similar alongside the actual process of Hitag2. The general idea is to try to guess the key-bits by guessing the internal state of the LFSR and scoring this configuration in combination with what we know to be the correct keystream. We implemented the attack in the file `attack.py` which is also commented extensively. Never the less we explain the bigger picture of the attack and out implementation here.

We start by generating some traces. In a real-life scenario this information would be taken from the collected traces of the key fob. The paper recommended 4-8 traces, we generated 10 where the counter is advanced by 10 for each trace to achieve more powerful correlation information and to have more traces to score against (more robust result). Before we start the attack, we run a precomputation that is explained later.

Now that the traces generated, they are handed over to the first attack part function `garcia_attack_step1_2_3(…)`. We basically start by guessing (means iterating over all possibilities) the upper 16-bit part of the key that is fully present in the LFSR right before the keystream generation starts (see Figure 4). We then see that some of the input bits to `f20` are covered by our currently guessed window (8 green inputs in Figure 6) and some are not (12 red inputs in Figure 6).

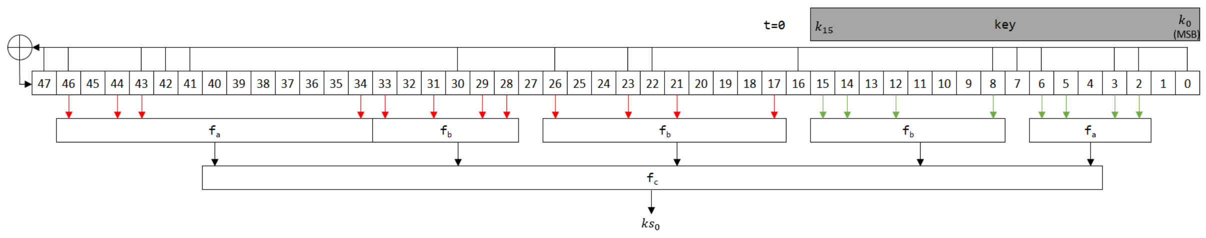

Figure 6 Attack guessing shift t=0

The core of the attack is that the `f20` function is biased and leaks information about the key. The current guess of the key/LSFR is scored based on the number of combinations for the "unknown" inputs that create the correct keystream bit (we know this from the trace). This means we would need to execute `f20` 2^{12} times and since the configurations of the guess window might be used again in the next steps (see blow, when more keystream bits are scored) we should precompute parts of this scoring process.

Garcia et al. don't cover this precomputing in their paper but overall, the attack is described very briefly and gives no information on the actual implementation, nor the optimization strategies they used, so probably they implemented similar things.

First we determine how many input bits (green inputs in **Figure 6**) are in our current window according to the *window size* (in Step 5 extended, now fixed to 16) and the *shift t* to the right in the function `precalculate_window_table(…)`. The shifting is done to be able to score multiple keystream bits. This evaluation is necessary since the amount of input bits that are determined by our current guess window don't get smaller at every shift. For example, for window-size 16 with shifts of 2 and 3 still the same amount of input bits is inside the window:

```
Window Size: 16                                          (out/window_table.txt)
  t=0: Known Indices: [2, 3, 5, 6, 8, 12, 14, 15]
  t=1: Known Indices: [2, 3, 5, 6, 8, 12, 14]
  t=2: Known Indices: [2, 3, 5, 6, 8, 12]
  t=3: Known Indices: [2, 3, 5, 6, 8, 12]
...
```

With the knowledge about the number of known input bits we then can use this to look up the probability that `f20` outputs a 1 for this bit configuration in the LFSR. For this we consult the table precomputed by `precalculate_probability_table(…)`. Here we are computing the scoring they provide in the paper for every possible configuration of known input-bits for all window sizes. For example, if we have 8 known input bits, we consult the table with the current *window size* and *shift t* at the index determined by the configuration of the known input bits. In the table we have already computed the amount of configurations of unknown bits that lead to `f20` outputting a 1, e.g. if the known input bits are 00000011, then there is a 0.375 chance that f20 outputs a 1 (→bias).

```
Known bits: 8                                               (out/prob_table.txt)
...
  Known Val 00000011: P(1)=0.375000
...
```

This probability is then converted to a "score" in the attack by multiplying it by 2 (in the scoring formula Garcia et al. divide by 2^{19-n} and not 2^{20-n} to center the score around 1 (We simply multiply the probability by 2 in the attack. Like this, guesses that show correlation accumulate a high score while negative ones result in lower scores.

Note: The written description of step 3 in the paper describes that in the first iteration 8 input bits are known, and that by shifting the window it is possible to score against the keystream bits $ks_0 \dots ks_{15}$. This means for every shift we have a new configuration in the LFSR and a keystream bit that is produced by executing `f20` on this LFSR state. The recursive formula they provide for this scoring

$$\text{score}(x_0, ks_0) = \text{bit\_score}(x_0, ks_0)$$
$$\text{score}(x_0 \dots x_{n-1}, ks_0 \dots ks_{n-1}) = \text{bit\_score}(x_0 \dots x_{n-1}, ks_{n-1}) * \text{score}(x_0 \dots x_{n-2}, ks_0 \dots ks_{n-2})$$

doesn't fully cover this concept though, because if you break it down for this first iteration ($n = 8$) we reach this:

$$\begin{aligned}\text{score}(x_0 \dots x_7, ks_0 \dots ks_7) &= \text{bit\_score}(x_0 \dots x_7, ks_7) * \text{score}(x_0 \dots x_6, ks_0 \dots ks_6) \\ &= \text{bit\_score}(x_0 \dots x_7, ks_7) * \text{bit\_score}(x_0 \dots x_6, ks_6) * \\ &\quad \text{score}(x_0 \dots x_5, ks_0 \dots ks_5)\end{aligned}$$

and with that after some more break downs finally:

$$\begin{aligned}\text{score}(x_0 \dots x_7, ks_0 \dots ks_7) = {} & \text{bit\_score}(x_0 \dots x_7, ks_7) * \text{bit\_score}(x_0 \dots x_6, ks_6) * \\ & \text{bit\_score}(x_0 \dots x_5, ks_5) * \text{bit\_score}(x_0 \dots x_4, ks_4) * \\ & \text{bit\_score}(x_0 \dots x_3, ks_3) * \text{bit\_score}(x_0 \dots x_2, ks_2) * \\ & \text{bit\_score}(x_0 \dots x_1, ks_1) * \text{bit\_score}(x_0, ks_0)\end{aligned}$$

This formula would suggest that with every shift one of the input bits falls out of the window and with that we score against the next keystream bit, which seems to also only cover $ks_0 \ldots ks_7$. On top of that the order of the keystream bits seems reversed since in their notation ks_0 should be the first/MSB of the keystream so also the first keystream bit against which we score. The formula states that the first one we score against is ks_7 when we have the "full window" though.

We stuck to the interpretation that is closer to the textual explanation, so that the LFSR configuration is scored against the next keystream bit at every shift. These bit scores are then simply multiplied to determine the full score for the guess.
This is done for all possible guesses of the current window size and only the 400.000 best candidates are kept. After the first round we only have $2^{16} = 65536$ candidates, but in later iterations of step 5, candidates will be cut off (can be seen in the console when the attack is executed). This is another core aspect of the attack because we use this approach to eliminate candidates with the correlation knowledge we gain, to not need to brute force all the possibilities. To inspect the candidates, our python script exports all the candidates after the evaluation but before the "reduction" to a file in the folder `/out`.
In step 5 we go over all candidates from the last step and extend the window size by one, this means we include one more bit in our key guess, so we extend every guess by bit 0 and also add the same with the extension of bit 1. From this point on, we have to respect the construction of the new bits which are inserted during the initialization phase. This can be seen in **Figure 7** – the bits in the LFSR are not equal the key anymore but rather dependent on the XOR operation in the initialization phase. Since we know the IV bits and also computed the output of `f20` on the state before (see **Figure 2** output b_0), we can simply compute the new a_{48} (and in the next iteration the next one). That's why we keep track of the recreated state per trace in our implementation (seen in the candidate files we export). Since the IV is different in each trace, also the LFSR bit is different for each trace (necessary to consider in the scoring).
In the second part of **Figure 7** we see that this generated bit is covered by our extended guessing window (size 17) on the left and we can simply do the scoring and shifting again like before. One again only the top 400.00 candidates are kept and we continue by making enlarging our window further to go on with the attack.

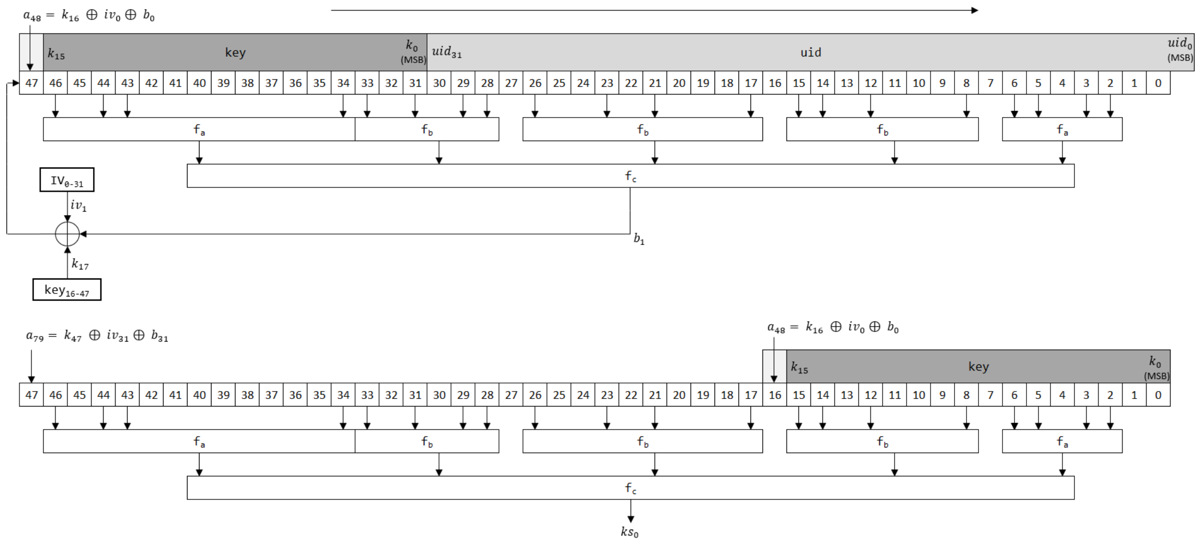

Figure 7 Bit construction Initialization Phase when extending window

In this context, the authors state something a bit confusing once more. They write that with this procedure "*key bits* k_{16+i} *can be computed for* $i \in [0,31]$" [1], which would mean that we get the full key through this approach. But they also state, that the window will only be extended from 17 to 32. This makes sense, since in the traces of the key fob only 32 keystream bits are sent which means we can only score against a maximum of 32 keystream bits. This only allows us to recover the top 32 bits of the key! Still the full key has 48 bits. This doesn't add up fully. Therefore, we added `garcia_attack_step6(…)` which takes the top key candidates after the attack and simply brute forces the last 16 bits for each of them. Since this only means 2^{16} iterations of Hitag2 per candidate, it is doable, as long as the "actual key candidate" is among the top candidates. Anyways the paper leaves this part out.

4 Discussion

4.1 Attack runs

Sadly, the attack didn't seem to produce the correct result like it should have, according to the paper. We tried many configurations of keys, UIDs, number of traces, smaller/wider distances between the counters, but it mostly never worked fully (except once).
Because the keys are extended bit-by-bit we know exactly which intermediate keys have to stay in the candidates when we repeat Step 5 for all the window sizes. If the intermediate key for a round gets excluded and not passed to the next round, the attack can't work, since only the passed-on candidates are extended by one bit (0 and 1), but if the bit before is not like it "should be" (this guess is excluded) it is impossible to determine the real key in later attack steps. This information is also printed in the console during the attack.
With most of our tests the necessary intermediate keys simply got a too low score and didn't place among the top 400.000 candidates, which excluded them during the attack. Surprisingly, the first test we did, succeeded, but then none other followed. So, we have to conclude that this this was probably more luck, but shows for example that the 6th attack step works accordingly.
We hand in the project with the following experiment runs:

Experiment ID	*Key*	*UID*	*Traces*	*Successful?*
Experiment 0 */out0*	0x123456789abc	0x01234567	10 traces counters: 10 steps apart	YES, interm. key at position 769 after last round
Experiment 1 console_out1.txt	0xabcdef123456	0x89abcdef	10 traces counters: 10 steps apart	No, at window-size 25: interm. key at position 798618, so excluded
Experiment 2 console_out2.txt	0x123456789abc	0x01234567	20 traces counters: 10 steps apart	YES, interm. key at position 1093 after last round
Experiment 3 console_out3.txt	0x123456789abc	0x01234567	8 traces counters: 1 steps apart	No, at window-size 25: interm. key at position 455077, so excluded
Experiment 4 console_out4.txt	0x276359283601	0x01234567	10 traces counters: 10 steps apart	No, at window-size 20: interm. key at position 408091, so excluded
Experiment 5 console_out5.txt	0xabcdef123456	0x89abcdef	10 traces counters: 5627726 steps apart	No, at window-size 26: interm. key at position 480018, so excluded
Experiment 6 console_out6.txt	0xabcdef123456	0x89abcdef	10 traces counters: starting at 0xAAAAAAA, 10 steps apart (random bigger value)	No, at window-size 20: interm. key at position 504221, so excluded

The runs can be recreated by changing the ID in `experiment_traces(experiment_id)`. The console outputs are in the folder `/console`. And for experiment 0 we included all files in `/out1`.

In experiment 0 the real key is not the highest scored one, but still, among the top 1000, so we can find it pretty quickly in attack step 6. With this, we could break the cipher in around 15 minutes and recover the real key! In experiment 1 we can see that the intermediate key of the real key gets excluded during the attack, leading to an unsuccessful attack. We then tried to play around with the working experiment 0 and adjusted some parameters. But also, this didn't lead to remarkable results. Experiment 2 worked (it was also nearly the same as experiment 0) but surprisingly with more traces the final result actually scored a bit lower and still is not among the very first candidates, which doesn't go along the explanations of "more correlation information – better scoring". With a lower amount of 8 traces (lower bound of requirements by Garcia et al.) and "less informative correlation information" (counters only one apart) in experiment 3, the attack turned unsuccessful, where it still should still be possible according to Garcia et al. Also experiment 4 was not successful, which shows, the attack success is also not coupled to specific UIDs. We tested this to check the idea if maybe the attack is only successful under certain requirements, e.g. if f20 would be only biased with certain LFSR-values where the $UIDs$ plays a big role in the beginning. We thought it might be possible these requirements were unknown to us and maybe even to Garcia et al. when they carried out the attack, so it's not included in the paper, but experiment 4 proves otherwise, at least for the UID example. Still there could be other "silent" requirements. Also, when we take experiment 1 again and try to make the attack work by providing "better correlation information" by placing the counter values further apart (experiment 5) or starting with a higher value (counter has more entropy – experiment 6), the attack doesn't work.

4.2 Problems

Possible explanations for the attack to not work fully could be that a one-bit-error or a wrong layout of the variables due to a misconception doesn't score the correlation information correctly. This way the attack might seem to work, but the guess that should stay in the first 400.000 candidates simply achieves a score too low after a few rounds and is excluded (like in most experiments), making it impossible to guess the right key in the end. Especially the computation of the new LFSR bits in the initialization phase $a_{48}..a_{49}$, the precomputation and the scoring itself could host this little error, destroying the functionality of the attack. Still with exhaustive debugging we couldn't identify it.

In addition, the paper has contradictions in the explanation and the specified formula – according to our understanding – like already pointed out, and Garcia et al. didn't cover the computation of the last 16 key bits in the paper (brute force). This raises the question, if our understanding of the attack is correct and all necessary parts needed to implement the attack were actually covered by the paper.

4.3 Workarounds

To make the attack work anyways we collected some ideas, but realized they also don't make the attack work as promised in the paper:

Since the key seems to stay in the candidates a few rounds, we could start with the brute forcing earlier, but this still implies that we are more trying around than a following a deterministic approach, since we couldn't determine a stable cutoff where this would make sense.
According to Garcia et al. more traces and countervalues that lie further from each other might result in better results because of better correlation information. This also couldn't be verified with our implementation, which doesn't make it a suitable enhancement (in our approach). Also, a higher cutoff (than 400.000) might seem to help the attack succeed, but also for this, we couldn't really find a cutoff that worked for our implementations without ruining the efficiency of the attack.

4.3 Further Optimations

If the attack would have worked, the next steps would be to rewrite the code in C with multi-threading to make the attack a lot faster. Still with our approach with pypy we already get a short runtime that satisfies our requirements.

5 Conclusion

We conclude by stating that the attack didn't work fully, except on one lucky example. We demonstrated how the attack fails in different scenarios, but couldn't identify the error that makes it fail clearly.

References

[1] F. D. Garcia, D. Oswald, T. Kasper, and P. Pavlidès, "Lock it and still lose it - on the (in)security of automotive remote keyless entry systems," in *Proceedings of the 25th USENIX Conference on Security Symposium*, in SEC'16. USA: USENIX Association, Aug. 2016, pp. 929–944.

[2] The PyPy Project, *pypy*. (2026). Accessed: Feb. 18, 2026. [Online]. Available: https://www.pypy.org/download_advanced.html

[3] A. Verstegen, *factoritbv/hitag2hell*. (Feb. 03, 2026). C. Accessed: Feb. 12, 2026. [Online]. Available: https://github.com/factoritbv/hitag2hell

[4] A. Verstegen, R. Verdult, and W. Bokslag, "Hitag 2 hell - brutally optimizing guess-and-determine attacks," in *Proceedings of the 12th USENIX Conference on Offensive Technologies*, in WOOT'18. USA: USENIX Association, Aug. 2018, p. 14.

[5] V. Gayoso Martínez, L. Hernández Encinas, A. Martín Muñoz, and J. Zhang, "Breaking a Hitag2 Protocol with Low Cost Technology:," in *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2017, pp. 579–584. doi: 10.5220/0006271905790584.