# EEPROM: It Will All End in Tears

Philippe Teuwen[1] and Christian Herrmann[2]
pteuwen@quarkslab.com
ch@icesql.net

[1] Quarkslab
[2] IceSQL AB

**Abstract.** RFID tags are supposed to be robust to situations such as a quick removal from the powering field when the user swipes a tag over a reader. In this paper, we describe the various physical effects that can happen when an EEPROM *write* or *erase* operation is interrupted, and we explain how to control these side effects to learn about the inner mechanisms of security features and to challenge them. We show how to defeat four types of security features on different tags: erasing OTP bits, recovering a locking password, unlocking a read-only UID and resetting a secure counter. We attack them successfully thanks to the different tools we developed and we share these tools to the community to facilitate future research.

## Acronyms

| | |
|---|---|
| **EEPROM** | Electrically-Erasable Programmable Read-Only Memory |
| **HF** | High Frequency |
| **LF** | Low Frequency |
| **NFC** | Near Field Communication |
| **OTP** | One-Time Programmable |
| **RFID** | Radio Frequency Identification |
| **UID** | Unique Identifier |

## 1 Introduction

Tearing-off is a term used in the RFID and NFC domains to express the fact that when users present their tag to a reader, they may walk away too quickly while the reader is still busy talking to the card. This can become problematic if the card is busy performing a write operation in its EEPROM because data on the tag should not be left in an inconsistent state. Modern tags are well designed against this issue and always ensure a robust state by e.g. committing a write at once, only if the energy budget allows for it.

This is true at least for physical walking away tear-offs, when the energy level of the powering field drops at a relatively slow pace, but interesting things can happen if we abruptly drop the power source during an EEPROM write in a programmatically and timely controlled way. In 2008, Teepe mentioned the possibility to trigger tearing events in order to corrupt data voluntarily [16] and in 2020, Grisolía and Ukmar demonstrated how tear-off attacks could reset the OTP (*One-Time Programmable*) bits of a Russian tag [2], which we will shortly recap as a first successful example.

**Contributions.** Our contributions lay in the observations we could formulate based on the various experiments we conducted on different tags and in the way they can be used to reveal the inner mechanisms of some security features and to elaborate attack strategies to defeat them by a fine control of tear-off side effects. These observations pave the way for more experiments on other embedded EEPROMS, in RFID tags or in other forms. We demonstrate their relevance by presenting three new attacks on the security features of three different RFID tags. We also added new tools to the Proxmark3 RRG code repository [5] to bring a generic support for tearing experiments for all types of tags supported by the Proxmark3.

**Paper organization.** In Section 2, we describe how EEPROM technology works and stores logical bits, then in Section 3 we describe the various physical effects that can happen at high level when an EEPROM *write* or *erase* is interrupted and when an EEPROM *read* is performed after such interruption. In Section 4, we explain how these effects can have security consequences and what type of security features could be targeted. Section 5 shows how to defeat four types of security features on four different low-frequency and high-frequency tags, from the easiest to the most complex and most recent one: erasing OTP bits (§ 5.1), recovering a locking password (§ 5.2), unlocking a read-only UID (§ 5.3) and resetting a secure counter (§ 5.4). In Section 6, we present the different tools we made available in the Proxmark3 RRG code repository to facilitate future research in this exciting area. Finally, we detail our conclusions in Section 7.

## 2 EEPROM Internals

To understand the rather strange behaviors of EEPROMs described in this article, it is essential to have a grasp on the underlying physics.

Since the technological and statistical reality varies across chips and manufacturers, we will only give a rough model of an EEPROM cell, good enough to explain various empirical observations stated in the article.

When stored in an EEPROM, bits are not just 2-state entities, either 0 or 1. Bits are actually represented by the presence or absence of a bunch of electrons stored in the floating gate of a transistor.

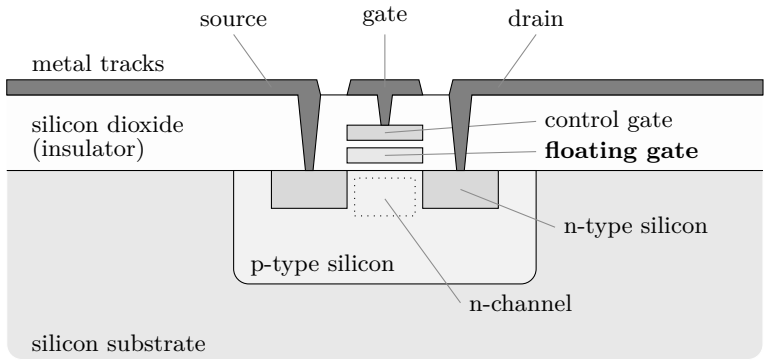


**Fig. 1.** EEPROM transistor.

The floating gate, shown in bold on Figure 1, is isolated by a very thin oxide layer. The value of the stored bit is the result of the indirect measure of the charge stored in this floating gate. Electrons can be dragged into (by channel hot injection) or removed (by Fowler-Nordheim tunnelling) depending on the relative high positive or negative voltages applied across source, drain and a second gate called control gate [3]. When no voltage is applied, electrons are kept captive in the floating gate and the memory state is retained, hence the non-volatile nature of EEPROMs. Different technologies exist so we will not describe further this mechanism as we do not know which one is deployed in the devices we studied.

To modify values stored in an EEPROM, two basic operations are available: *erase* and *write*. Typically, bits can be written individually towards one value but erasing them back to their initial value is only possible on a whole range of them at once, typically a full *word*, *page* or *bank* depending on the specific product, representing ranges from about 4 bytes to an entire memory.

**Observation 1 (Logic implementation)** *We notice two kinds of logic implementation:*

— *(A) Erasing operation resets all bits of a range to value* 0 *and writing operation sets some individual bits to value* 1*;*
— *(B) Erasing operation resets all bits of a range to value* 1 *and writing operation sets some individual bits to value* 0*.*

A writing operation encompasses a full *word* or *page* and the result will be the combination of the already set bits with the newly set bits. If the logic is to erase to 0 and set to 1, the result of a write is the logical *OR* between the old value and the value to be written. If the logic is inverse, a logical *AND* is used to combine these values.

Typically, a `WRITE` command requires internally an *erase* operation followed by a *write*, so arbitrary values can be written. Other more complex commands such as increasing a counter, changing a configuration, etc. will follow a pattern such as a *read* of the current value, a *computation* of the new value based on the old one and possibly some command parameters and finally an *erase* and a *write*. We will see that the pattern can be more complex when the commands implement anti-tearing countermeasures.

Still, dragging electrons in and out of the gate is not an atomic event and if the process is prematurely interrupted, quantity of electrons in the gate can be anything between *empty* and *full*.

## 3 Interrupting EEPROM Operations

When an EEPROM operation is interrupted, it affects the number of trapped electrons, which, in turn, affects the logical value when read back, based on other factors as well.

Abruptly interrupting an operation on an RFID tag can be done simply by quickly shutting down the reader field at the proper timing as RFID tags get powered by a magnetic field generated by the reader.

### 3.1 Quantity of Trapped Electrons

A simple illustration is to see the gate as a bucket that will be filled or emptied by *erase* and *write* internal operations. To keep things simple, we assume that if the bucket is less than half full, it is interpreted as 0, otherwise as 1.

A basic `WRITE` command is composed of an *erase* phase and a *write* phase. Depending on when we interrupt it, we can

— diminish the number of electrons in all the gates which were not already empty if we interrupt the *erase*;

— empty completely the gates if we interrupt between the *erase* and the *write*;
— increase the number of electrons in a subset of the gates, depending on the written value.

All buckets are not filled exactly at the same rate because transistors, built from dopants injected in the silicium, do not get the exact same amount and spatial distribution of dopants during the manufacturing.

The consequence is that some buckets will cross the 50% threshold sooner than others. Consider one byte – stored by 8 gates – where `0xFF` is written over `0xFF`. We interrupt the writing at different times, then we read back this byte. For increasing times, we can observe something like `0xFF`→`0xDB`→`0xC9`→`0x80`→`0x00` then `0x00` for a while, then e.g. `0x00`→`0x48`→`0x6e`→`0xFF`. Figure 2 illustrates the number of electrons being slowly removed at different rates across the 8 gates, explaining how such sequence of values could be observed when the erase phase is interrupted at different timings. Timings depend on the EEPROM technology and manufacturer.

Obviously, we cannot directly read the electric charge value of a gate and what we describe is based on indirect observations implying `READ` commands.

Repeating the same experiment on the same set of transistors will lead to about the same results: statistically, the same subset of transistors swap faster than the others. But there is also some randomness, so it is a statistical trend and there will be some variability to take into account across repetitions. Note that there is no direct relationship between the transistors that get erased first and the ones that get written first (or last) because *erase* and *write* rely on different physical effects: channel hot injection versus Fowler-Nordheim tunnelling.

**Observation 2 (Biased bits)** *For a given word location, some specific gates will be filled faster than other ones — and some will be erased faster than other ones — in quite a reproducible manner.*

Moreover, our experiments have shown that the following trick can be used in some situations to increase precision and reproducibility.

**Observation 3 (Progressive tear-off)** *It is possible to have a better control on the number of trapped electrons in the gates by initiating and interrupting an internal operation at an early stage multiple times, slowly adding or removing charges until reaching the desired value, rather than trying to reach the point of interest in one single interrupt.*

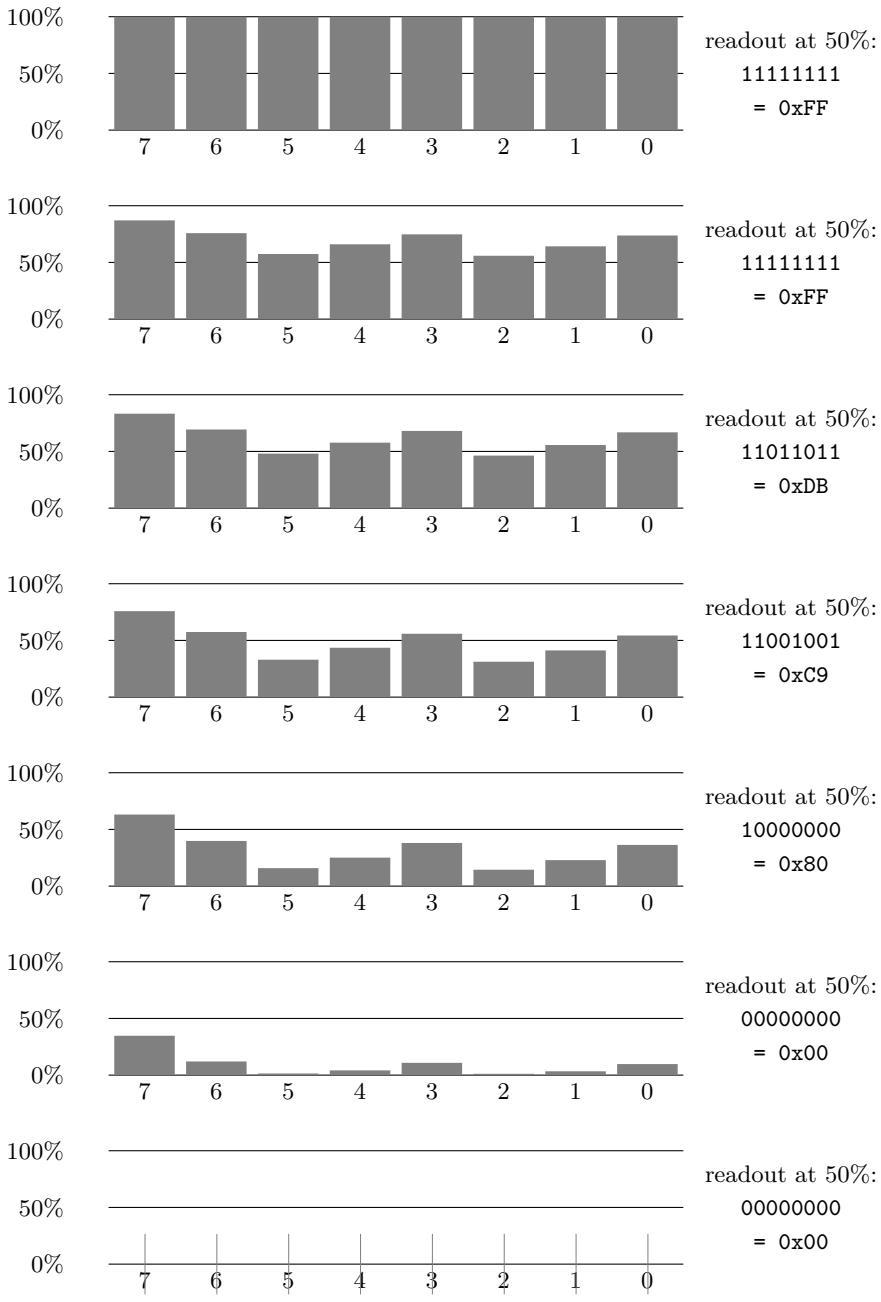


**Fig. 2.** Simulation of the internal transistors states after an interrupted erase of an EEPROM byte at different timings, from the shortest at the top to the longest at the bottom, to explain observed values on a real tag.

Our hypothesis is that during the gate update, electrons are migrating firstly at a slower rate when the applied voltage is still rising and we obtain a better precision in the charge quantity for the same temporal precision when acting during this early phase.

This is only possible for the first internal operation. For example, on a `WRITE` command composed of the succession of an *erase* and a *write*, it is only possible to slowly unload the gates in multiple passes by triggering and interrupting the *erase* operation but only one single pass can be interrupted on the *write* operation as it is always preceded by an uninterrupted *erase*.

Another trick is to put the tag as far as possible from the reader while still working properly.

**Observation 4 (Distance dependency)** *It is possible to have a better control on the number of trapped electrons in the gates by working at a greater distance from the reader.*

When the tag is far away, the EEPROM operations are apparently performed slightly slower due to the limited power available.

We could observe two more variables affecting EEPROM *erase* operation.

**Observation 5 (Content dependency)** *The speed of EEPROM* erase *operation is affected by the value to be erased: faster when very few bits need to be erased.*

Here is an example using a MIFARE Ultralight where the effect can be seen byte per byte. A first value is written with the following pattern: `0xFF3F0F03`, so 2 bits set in the first byte, 4 in the second, 6 in the third and all 8 in the fourth one. Then a second write is initiated but interrupted by a tearing. We repeat the experiment and check the content of the word after tearing at different timings. Here are our results, obtained with a Proxmark3 and the command `hf mfu otptear`[3] presented in Section 6.

— Erasing `0xFF3F0F03` → `0xFF3F0F00` after 482 $\mu$s;
— Erasing `0xFF3F0F03` → `0xFF3F0000` after 508 $\mu$s;
— Erasing `0xFF3F0F03` → `0xFF010000` after 542 $\mu$s;
— Erasing `0xFF3F0F03` → `0x00000000` after 558 $\mu$s.

One could think the bytes are simply erased from right to left so we repeat the experiment with `0x030F3FFF`.

— Erasing `0x030F3FFF` → `0x000F3FFF` after 470 $\mu$s;

3. `hf mfu otptear -b 5 -i 2 -s 460 -e 600 -d ff3f0f03`

— Erasing `0x030F3FFF` → `0x00003FFF` after 512 $\mu$s;
— Erasing `0x030F3FFF` → `0x000000FF` after 538 $\mu$s;
— Erasing `0x030F3FFF` → `0x00000000` after 562 $\mu$s.

This experiment confirms our observation: on this card, the *bytes* containing fewer bits are erased faster.

So it is very important, when you try to repeat tearing experiments on the *erase* phase, to be careful from which initial values you start.

We repeated the experiment on the *write* operation of the same card but the *write* seems not much affected by the value to be written.[4]

The other variable affecting EEPROM *erase* operation is temperature. Caution must be taken to keep the same environmental parameters when reproducing experiments.

**Observation 6 (Temperature dependency)** *The speed of EEPROM* erase *and* write *operations is sensitive to temperature. Operations may be slowed down by choosing adequate temperatures.*

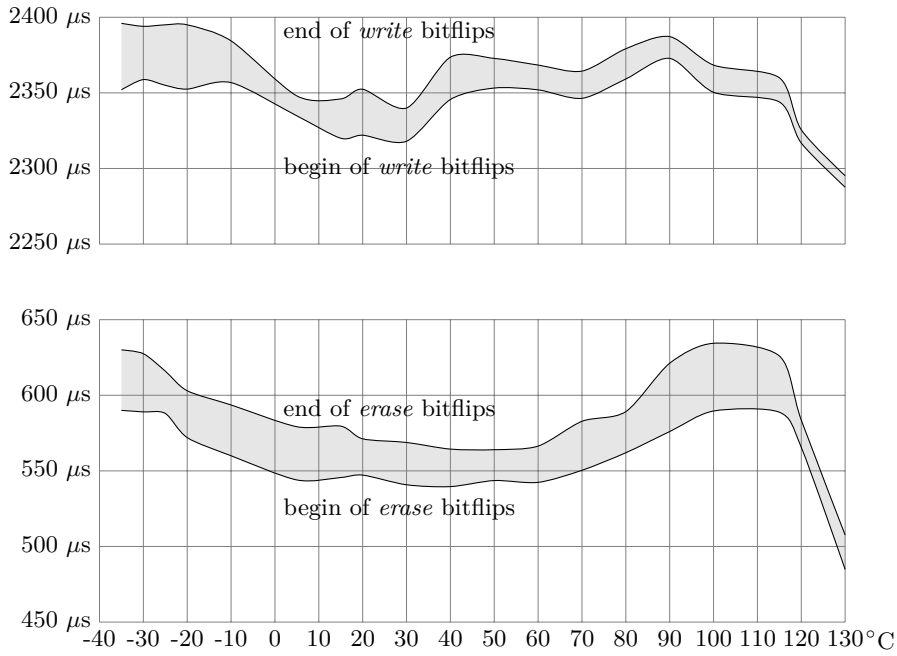


**Fig. 3.** Temperature dependency of tear-off timings for which bitflips are observed during EEPROM *erase* and *write* operations on a MIFARE Ultralight.

4. `hf mfu otptear -b 5 -i 2 -s 2320 -e 2360 -t 030F3FFF`

Using again a MIFARE Ultralight and interrupting a `WRITE` command at various times during the *erase* and *write* operations, we check at which timings the bitflips are occurring and we reproduce the experiment at different temperatures. Experimental measures are shown in Figure 3, with bitflips occurring in both gray areas.

On this card, the temperature has quite some impact on the temporal window during which bitflips are occurring. As in most cases we want EEPROM operations to be performed slowly to have a better opportunity to interrupt them at critical timings, cooling down the chip or heating it around 100°C seems a promising technique to slow down the *erase* phase. The *write* phase seems to offer fewer opportunities outside room temperature unless using temperatures below -20°C. Around 140°C, the card stops working properly. Of course such experiments must be repeated on other types of cards to determine their own temperature dependency.

We've seen in this section various methods to get quite some control on EEPROM *erase* and *write* results by controlling the environment and triggering one or more tearing events. If all conditions are fulfilled, we can even do some steganography! For example, interrupting *write* operations to fill gates at either 60% or 100% can be a way to hide zeroes and ones. A *read* operation will return 1 in both cases. But interrupting an *erase* operation can bring them to e.g. 30% and 70% and a *read* operation will now reveal the hidden content.

Storing more information than a single bit in a gate [4] is a technique called MLC (*Multi-Level Cell*) and already used by most flash memory manufacturers to achieve nowadays multi-gigabyte sizes. For example, for the QLC (*Quad-Level Cells*) to store 4 bits per transistor, it requires to be able to make a clear and robust distinction among 16 different charge levels! They typically achieve these performances by combining several *read* operations at different voltages and adding strong error correction techniques.

### 3.2 Interpretation of Trapped Electrons

In a first approximation, we said that a bucket less than half full is a 0 and more than half full a 1. Actually even repeating a *read* operation on the same memory location can lead to different values.

**Observation 7 (Weak bits)** *A gate loaded with a number of electrons very close to the threshold can be read as a* 1 *or a* 0 *across several reads, with some probability gradient related to the actual amount of charge.*

So, combining Observations 2 (Biased bits) and 7 (Weak bits), when reading several times the same half-written memory word, we will see some bits always read as 0, some always read as 1 and some flipping occasionally. Figure 4 illustrates the same byte being read at slightly different thresholds, with two flipping bits. Such weakly programmed flipping bits are also referred as *weak* bits.

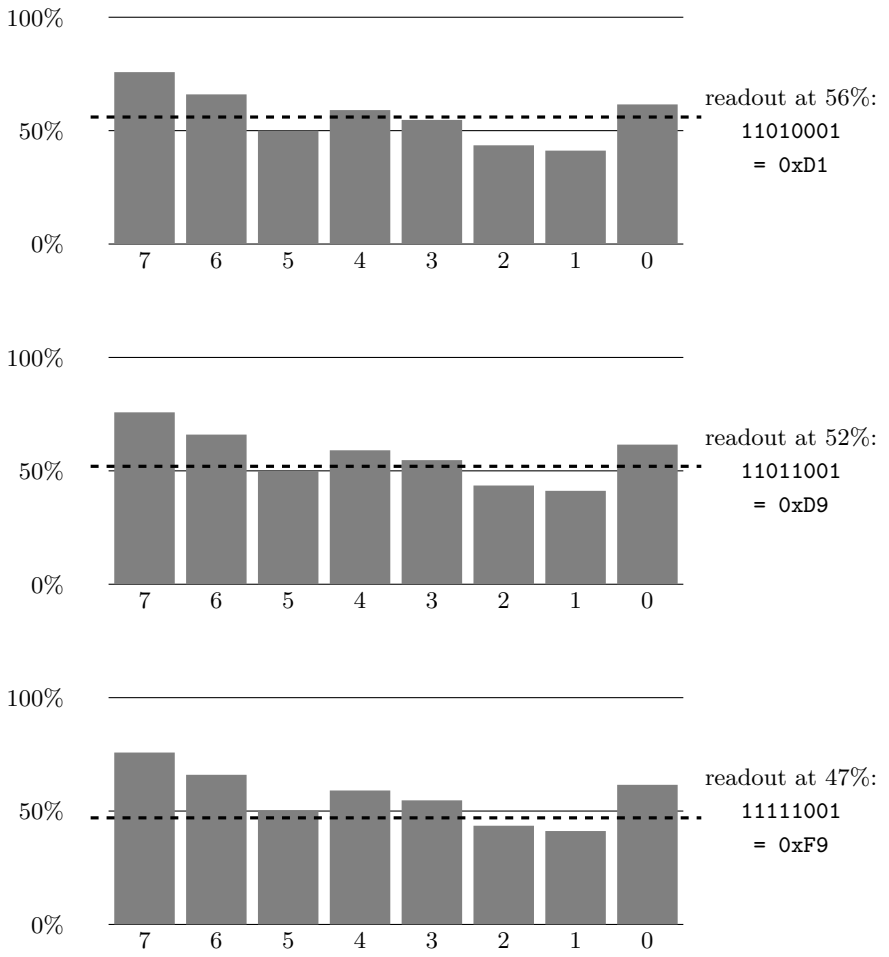


**Fig. 4.** EEPROM weakly written byte read at slightly different thresholds.

These differences can be used to fingerprint EEPROM memory locations quite similarly to SRAM PUF (*Static Random Access Memory Physically Unclonable Function*) [13], except that any point in the transi-

tion from a word value to another can be observed. Using this property, one could turn cheap memory tags into unclonable (and quite challenging to emulate) tags.

But there is more. We observed that we can have some control on these flipping bits.

**Observation 8 (Distance effect on weak bits)** *The probability to read a flipping bit as a* 0 *or a* 1 *can be influenced with the distance to the reader.*

If e.g. a card can be read at a distance between 0 and 4 cm, we place the card at 2 cm and at some point after a tearing event we obtain a flipping bit when reading several times the same memory, then bringing the card closer to the reader (0 cm) will stabilize the reading as 1, while moving the card away (4 cm) and the same bit will be read as a 0!

One hypothesis is that distance influences the actual voltage, and voltage influences comparators in charge of measuring memory cells.

## 4 Opportunities for Security Vulnerabilities

Back in 2008, Teepe mentioned already the possibility to trigger tearing events to corrupt data voluntarily [16].

Before investigating more complex commands, testing tear-off on simple commands such as a `WRITE` or an `ERASE` allows to better understand and characterize a given EEPROM technology, even if useless from a security perspective (if you can already write any value you want, what is the point?)

- What is the granularity of the memory being written?
- Does a `WRITE` firstly imply an internal *erase*?
- What is the default value of an erased EEPROM word? Full of zeroes or full of ones?
- What are the appropriate timings to target internal operations?
- Can we create flipping bits? Does distance to the reader affect their reading?

If the targeted system is indeed susceptible to EEPROM tear-off attacks, which functionalities could be affected from a security perspective?

The rule of thumb is that we need to find a security functionality involving EEPROM *erase* or *write* operations that an attacker can trigger but on which by design he does not have a full control on the final result. Tear-off will then bring some hope that we can nevertheless influence the final result in a favorable way.

Think of a poorly implemented PIN (*Personal Identification Number*) trial counter we want to reset. If a `VERIFY_PIN` command starts with increasing temporarily the PIN counter, which implies in turn a first internal *erase* operation, bad things will happen.

One thing to consider is the possibly destructive nature of the tests and the availability of samples on which you have enough control to *reset* them in initial conditions, at least during the discovery phase, before switching to real targets.

Taking again the PIN trial counter, if you have only one single card with an unknown PIN, it will be very hard to find the sweet spot between *erase* and *write* without overshooting and triggering at least several *writes*, locking permanently the card. But if you have a second card whom you know the PIN, you can test it without fearing overshoots as you can *reset* the counter by typing the proper PIN between experiments.

It is also important to avoid blind situations. Elaborate strategies, scenarios, where you can reason about the internal state of the target and possibly infer information about the internal state from experiments. This will all become clearer with real examples.

## 5 Four Case Studies

### 5.1 Erasing OTP Bits

**Target.** OTP (*One-Time Programmable*) bits are a common security feature in RFID and NFC tags. An OTP memory is a memory where one can set some bits to 1 but never clear them back to 0. This is used for example to punch a 10-journey transportation ticket. If secure chips can indeed use polyfuses, it is a costly manufacturing process and tags implement the OTP feature using regular EEPROM and some logic such as writing the bitwise *OR* between the old value and the value asked to be written.

Nahuel Grisolía and Federico Gabriel Ukmar demonstrated in their thesis [2] tear-off attacks against the OTP feature of an HF (High Frequency: 13.56 MHz) tag, the MIK640M2D [9], a variant of MIFARE Ultralight, developed by Mikron.

Indeed this variant did not do anything to protect its five OTP (One-Time Programmable) blocks of 32 bits and a tear-off between internal *erase* and *write* operations is enough to set OTP bits back to zeroes.

**Strategy.** To explore and characterize tearing effects on a tag, one can follow these steps:

— Select firstly a non-OTP memory area;
— Write initial data, e.g. `0xAAAAAAAA`;
— Write different data, e.g. `0x55555555` and interrupt the operation;
— Read back. Given the pattern read back, we know if the interrupt was too early or too late;
— Adjust timings in a binary search manner and try again.

Once a good timing is found, move on the OTP memory. Timings may vary slightly but you have already a good approximation to start with.

— Select the OTP memory area containing some bits already set;
— Start from a slightly shorter timing;
— Write data to set one single bit and interrupt the operation;
— Read back;
— Increment slightly timings and try again.

**Results.** Because tearing anywhere between *erase* and *write* operations, it is very easy to find a proper timing and reset OTP bits on a MIK640M2D tag. A tool is available to automate this attack, as explained in Section 6.

**Mitigations.** There is not much option to mitigate this attack, besides not relying on MIK640M2D OTP feature. But we can have a look at other manufacturers and see how they deal with it. NXP MIFARE Ultralight tags are immune to this attack. Other Ultralight variants such as my-d move tags by Infineon [6] and F8213 NFC-Forum Type 2 tags by Shanghai Feiju Microelectronics Co [14] also have OTP bits and no protection against tear-off but – remember Observation 1 (Logic implementation) [B] – on these tags, the default value of an erased word is `0xFFFFFFFF`, so it is impossible to reset bits back to zero with this method. Simply inverting the logic of the stored bits in EEPROM provides an efficient security protection against tear-off! On the other side, if such a tear-off happens by accident, you may burn all the OTPs in the word being updated. Some other cards such as ST SRI512 [15] simply skip the *erase* operation when writing on OTP bits, which is probably easier and safer.

### 5.2 Recovering a Protection Password

**Target.** The ATA5577C [8] is a general-purpose reprogrammable LF (Low Frequency: 100-150 kHz) RFID tag with many features but we will focus on a subset of them.

— Block 0 contains configuration data, including modulation settings and one bit indicating if a password protection is activated;
— Block 7 contains the password;

— An undocumented *test-mode* allows writing specific patterns on the whole tag, ignoring the password protection.

When the tag is password-protected, it refuses any command unless the command includes the expected password. Except for this hidden test-mode, which has been investigated initially by *Marshmellow* on Proxmark3 forum [7].

**Field reconnaissance.** Launching a test-mode command will first trigger a mass erase then the writing of specific patterns. We have seen in the previous example that we can execute a tear-off between the *erase* and the *write* but besides recycling locked tags, there is not much advantage to do so. The tear-off teaches us that this tag follows the convention of Observation 1 (Logic implementation) [A].

But if we tear off sooner, according to Observation 2 (Biased bits) only some bits will be flipped towards zero. As we are tearing the first internal *erase* operation, we can also use Observation 3 (Progressive tear-off) to have more control on the process and flip slowly bits in several passes. At some point, the password protection bit will be cleared and the tag will be unlocked.

Of course its configuration and its content will also be affected by bitflips, including the password value itself, but we are sure that all bits still at 1 were at 1 and each bit set in the recovered password reduces the keyspace for a bruteforce by half. Repeating the same attack on a few other tags protected by the same password will quickly allow to recover the full password.

Reading such tag, if we do not know its current configuration, is always a bit tricky because we have to guess its modulation settings to get the bitstream. On the contrary, sending commands to the tag always relies on the same type of modulation: OOK (*On-Off keying*), short interruptions of the reader field (not too long, otherwise the tag will not have enough power to run!). This will be important when defining the strategies.

A first strategy will explore the fine details of tearing off a test-mode on a tag under control.

— Fill the memory blocks with `0xFF` bytes except for the configuration blocks;
— Start and interrupt a test-mode command;
— Write a valid configuration block;
— Dump the memory;
— Adjust timings and try again.

Observation 3 (Progressive tear-off) tells us that the approach of slowly unloading gates through multiple short writes works better than trying to

find the perfect timing to achieve some bitflips in one single write (erase) attempt. Remember that the working frequency of the power source is very slow - 125 kHz - and each cycle takes 8 $\mu$s. Moreover, the tag must have enough internal capacity to survive during OOK gaps, especially the start gap that can be as long as 50 cycles, 400 $\mu$s! That is under relatively idle conditions. When the tag is programming its EEPROM it is consuming its energy budget much quicker. Nevertheless, do not expect to achieve repeatable results with one-shot writing timings. The situation is quite different on 13.56 MHz tags.

On a first tag, we got interesting results by doing a first tear-off after 558 $\mu$s and then repeating the test-mode command several times with tear-offs after 544 $\mu$s and we could observe the entire memory being slowly flipped towards zero. To understand such timings, imagine that the quantity of electrons removed from the gate depends on that timing, so e.g. maybe 558 $\mu$s removes 40% and 544 $\mu$s removes 2%. If we used twice 558 $\mu$s, 80% of the electrons would be gone and the memory would be seen as fully erased. By using 558, 544, 544, 544... we are reducing the quantity of electrons from 100% to 60%, 58%, 56%, 54% etc and around these 45-55% we observed different quantities of bitflips. The values given here are only for illustration of the underlying phenomenon and far more tests would be required to characterize more precisely the behavior of this specific tag. But that is not the point and this imaginary example should be enough to understand the timing strategy in use.

The tests reveal that under test-mode, the initial *erase* operation happens on all memory blocks at once; all gates are affected concurrently and physics decides which bits will flip first.

**Strategy.** Once appropriate timings are found, we are ready for the second phase: attacking password-protected tags, with the following strategy.

— Start and interrupt a test-mode command;
— (Try to) write a valid configuration block;
— (Try to) read the (partially bitflipped) password;
— Adjust timings and try again until tag gets unlocked;
— Repeat on few other tags and collect partial passwords;
— Compute logical *OR* between partial passwords;
— Try to unlock a genuine tag with the recovered password;
— If it still fails, bruteforce by flipping few bits back to 1.

Timings for a new tag can be selected as follows: select a slightly shorter initial timing than what was tested in the discovery phase. For next rounds, use a cautious timing for a while. If some modulation change is observed, this reflects that some bits have been flipped in the configuration block

and we can continue with a slightly shorter timing to slow down bitflips as the goal is to get as few additional bitflips beside the password protection bit. If, after a while, nothing happens, increase slightly the timing and try again for a few rounds.

**Results.** To evaluate the feasibility of this attack, we tried 10 password recovery attempts on 4 different tags initialized with the password `0x0F0F0F0F`. Slightly different sets of timings had to be used for each tag. Obtained results are compiled in Table 1.

| Tag | Recovered password | Occurrences | Bitflips |
|---|---|---|---|
| #1 | `0x0F0F0F0F` | 7 | 0 |
| | `0x070B0F0F` | 3 | 2 |
| #2 | `0x040D0505` | 2 | 8 |
| | `0x040C0105` | 4 | 10 |
| | `0x04040105` | 1 | 11 |
| | `0x04040005` | 3 | 12 |
| #3 | `0x0105050D` | 1 | 8 |
| | `0x0105050C` | 5 | 9 |
| | `0x01050508` | 4 | 10 |
| #4 | `0x0F0F0F0F` | 4 | 0 |
| | `0x0F0F0E0F` | 2 | 1 |
| | `0x0B0F0E0F` | 1 | 2 |
| | `0x0B0F0C0F` | 1 | 3 |
| | `0x090B040F` | 2 | 6 |

**Table 1.** Study of recovery attempts on 4 ATA5577C tags programmed with password `0x0F0F0F0F`: showing for 10 attempts on each tag the number of occurrences of each partially recovered password values.

We have shown that even a global *erase* of the entire memory could be misused to defeat a security feature.

A specific command to support this attack has been implemented in the Proxmark3 RRG code repository as explained in Section 6.

**Mitigations.** As a countermeasure, it is possible to irreversibly lock the test-mode with the consequence that the modulation configuration of the tag will also be locked forever.

Some ATA5577C peculiarities were left out in this chapter. A more complete description is available in [17].

### 5.3 Unlocking a Read-Only UID

What about applying tear-off against a feature specifically designed to deter tear-offs?

**Target.** EM4305 [1] is another general-purpose reprogrammable LF RFID tag. EM4305 chip features 16 words of 4 bytes, labeled from 0 to 15. Protection Words are located in words 14 & 15. Ignore for a moment that there are two words as only one word is active at a time. The first 14 bits of a Protection Word represent the first 14 words. Once a Protection Word bit is set, the corresponding word is locked as read-only forever. By default, all words are writable except the second one, the UID Word, which is read-only.

The $15^{\text{th}}$ bit locks both words 14 & 15. If set, it forbids further modifications of the Protection Words themselves.

The chip has a feature designed specifically to avoid tear-off events: Protection Word is shared across words 14 & 15. The $16^{\text{th}}$ bit indicates which word is the active one. The remaining bits are unused.

The initial Protection Words content of a new tag is shown in Table 2.

| Word | Data | Active |
|---|---|---|
| 14 | `0x00008002` ≡ `0b...1000000000000010` | ★ |
| 15 | `0x00000000` ≡ `0b...0000000000000000` | |

**Table 2.** Protection Words, initial values.

Protection Word 14 is the active one (its $16^{\text{th}}$ bit is set) and protects the UID Word (its $2^{\text{nd}}$ bit is set). Word 15 is zeroed and ready to be programmed with a new Protection Word, which means it follows the convention of Observation 1 (Logic implementation) [A].

Protection Words 14 & 15 cannot be directly written with a `WRITE` command. To modify their content, a `PROTECT` command must be issued. Upon receipt of such `PROTECT` command, the EM4305 will do the following:

— Check if a password must be provided. If yes, check if a `LOGIN` command has been issued with a correct password;
— Check which Protection Word is active, i.e. which has its $16^{\text{th}}$ bit set;
— Write in the other Protection Word the new value, being the bitwise OR between the content provided in the `PROTECT` command and the existing content of the active Protection Word;

— Potentially check the written word or a sensor detecting under-power event;
— Erase the active Protection Word.

For example, after issuing a `PROTECT(0x00000001)` we will end up with the values displayed in Table 3 and the first two words are now read-only.

| Word | Data | Active |
|---|---|---|
| 14 | 0x00000000 ≡ 0b...0000000000000000 | |
| 15 | 0x00008003 ≡ 0b...1000000000000011 | ★ |

**Table 3.** Protection Words after `PROTECT(0x00000001)`.

By this mechanism of shadow Protection Word, we should never end up with a Protection Word with fewer bits set than before the command execution.

Abstract from the datasheet [1]:

> *The above implementation, using two physical words in a read/write EEPROM to represent a single Protection Register, was chosen as an additional security feature. This double buffered mechanism caters to the fact an EEPROM-write operation internally generates an erase-to-zero operation followed by the actual write operation. Should the operation be interrupted for any reason (e.g. tag removal from the field) the double buffer scheme ensures that no unwanted "0"-Protection Bits (i.e. unprotected words) are introduced.*

So the goal will be to end up with a new valid Protection Word but with its lock bits cleared, as shown in Table 4.

| Word | Data | Active |
|---|---|---|
| 14 | 0x00000000 ≡ 0b...0000000000000000 | |
| 15 | 0x00008000 ≡ 0b...1000000000000000 | ★ |

**Table 4.** Desired Protection Words to unlock UID.

**Field reconnaissance.** As in the previous examples, a first test phase will help understanding some elements required for the attack itself.

We have already seen that when cleared, the EEPROM is set to zeroes, but several other elements must be figured out for the tear-off to be successful.

What happens if a tear-off occurs before the erase of the current Protection Word? Both words will have their $16^{\text{th}}$ bit set so which one will be considered as active? Can we issue a `PROTECT` that does not set more bits than the current value? Even if the second bit is cleared, maybe the UID is always hardcoded?

To discover which word gets priority, we can set a lock bit, e.g. issue a `PROTECT(0x00000001)` command and tear-off before the erase. Then we test if the corresponding word 1 is locked or not. Word 1 is not protected so the answer is that if both Protection Words are valid, word 15 is the active one, as illustrated in Table 5. It is important to know it because if we start with an active word 15, no matter what we manage to write into 14 and perform a tear-off, word 15 will always be the active one. So we must start our experiments from an active word 14 configuration.

| Word | Data | Active |
|---|---|---|
| 14 | `0x00008003` ≡ `0b...1000000000000011` | |
| 15 | `0x00008002` ≡ `0b...1000000000000010` | ★ |

**Table 5.** Protection Words priority in case both words are valid.

A simple test shows that, even if unneeded, the full `PROTECT` cycle is performed and the Protection Words get swapped. It is interesting as we can swap the words at will: e.g. every time the configuration we are writing to word 15 becomes active but does not suit our goal, we can retry from an active word 14 by issuing a `PROTECT(0x00000000)` command.

**Strategy.** We have now everything in hands to develop a plan: issue a `PROTECT` command, perform a tear-off and hope that the $16^{\text{th}}$ bit will be written before the $2^{\text{nd}}$ bit, i.e. one gate being loaded above some threshold while the other one being still under the threshold.

— As the final word needs to be word 15, first check which one is active and swap them with a `PROTECT(0x00000000)` if needed, to start from word 14;
— Issue a `PROTECT(0x00000000)`, tear-off during writing then read both Protection Words to see where we stand;
— Adjust timings if needed and go back to step 1.

There are several possibilities to deal with after the tear-off.

— The 16th bit is not set, it was too soon. Try again with a longer delay;
— The 16th and 2nd bits are set. Swap words and try again with a shorter delay.

But weird things can happen too, as mentioned in Observation 7 (Weak bits). The 16th bit e.g. could be set but weakly. You see it as set but when you issue a `PROTECT` to swap, it is seen internally as cleared and you end up with an active word 15, copy of word 14, after the swap. These are typical corner cases you have to take into account and recover from when automating such an attack. Conversely the 2nd bit could be weak.

Therefore, once the desired value seems to be obtained by tear-off, it is important to *commit* it. In this case, issuing a `PROTECT(0x00000000)` without tear-off can be used as commit: it will read and interpret the current value then write it properly again in the other slot. This write is still comprising the elementary *erase+write* operations so we don't have to worry about not having erased properly that other slot during the tear-off.

Only then, we will see if we managed to clear the 2nd bit definitively or if we will have to try again.

Table 6 shows the succession of states leading to the unlocked state.

We have illustrated the strategy starting from a default configuration `0x00008002` but the same logic applies to configurations with more lock bits set.

So, starting from a blank EM4305 tag, can we obtain an EM4305 fully unlocked, with writable UID, a kind of equivalent to the UID-writable Chinese clones of MIFARE tags? Yes!

| Word | Data | Active |
|---|---|---|
| 14 | `0x00000000` | |
| 15 | `0x00008002` | ★ |

issue `PROTECT(0x00000000)` to get 14 active

| | | |
|---|---|---|
| 14 | `0x00008002` | ★ |
| 15 | `0x00000000` | |

issue `PROTECT(0x00000000)` with tear-off

| | | |
|---|---|---|
| 14 | `0x00008002` | |
| 15 | `0x00008000` | ★ |

issue `PROTECT(0x00000000)` to commit

| | | |
|---|---|---|
| 14 | `0x00008000` | ★ |
| 15 | `0x00000000` | |

**Table 6.** Protection Words states during a successful attack.

**Results.** By experiment, about 85% of the 26 tags we tested from various sources, starting from a default configuration, could be unlocked. Sometimes in a dozen seconds, sometimes in a few minutes, trying different positions on the antenna.

One could expect the probability to be 50% as either the $16^{\text{th}}$ bit or the $2^{\text{nd}}$ bit will always be written first for a given tag. The reality is that it is a process comprising stochastic fluctuations, so e.g. after a tear-off the $16^{\text{th}}$ bit will be filled at $47 \pm 5\%$ and the $2^{\text{nd}}$ bit at $53 \pm 5\%$. It means that by repeating the experiment many times, at some point $16^{\text{th}}$ bit can be filled at 51% and the $2^{\text{nd}}$ bit at 49%, as illustrated in Figure 5. But if the bias between our two bits is really strong and unfavorable, we will be unable to unlock the tag.

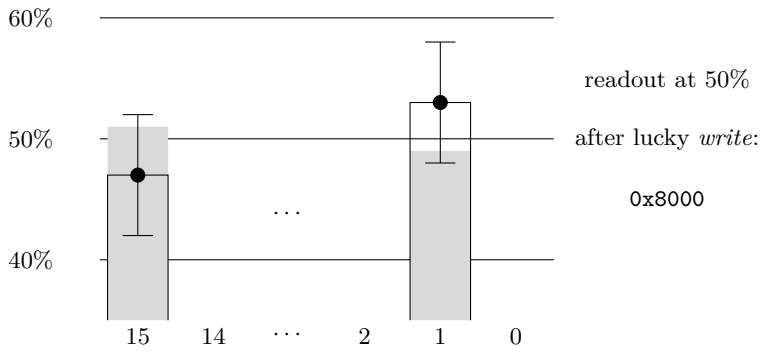


**Fig. 5.** Unfavorable odds (confidence intervals) after an interrupted *write* can still occasionally produce a favorable result (gray bars).

As stated in Observation 4 (Distance dependency), we noticed that there is more variability when the tag is put at a greater distance from the reader. The write process is slightly slower as the power budget is smaller and we had success on tags which we were unable to unlock when put directly on the reader.

The results are specific to each tag but they are quite reproducible.

An automatization of the attack has been implemented in the Proxmark3 RRG code repository as explained in Section 6.

**Mitigations.** Nothing can protect virgin tags, but a possible mitigation to protect a tag to be used in the field is to protect the tag with the password security feature of the EM4305. A possible design countermeasure would

have been to change the inner logic: when reading Protection Words to interpret them, the chip could compute a logical *OR* of both words and when deciding which one to replace during a `PROTECT`, choose the one with fewer bits set.

One may think locking the Protection Words themselves by setting the $15^{\text{th}}$ bit can prevent the attack. But even if `PROTECT` commands are denied, surprisingly words 14 & 15 are nevertheless swapped! So it's still possible to run the tear-off strategy and to clear lock bits, possibly including the $15^{\text{th}}$ bit.

Some EM4305 peculiarities were left out in the description above. A more complete description is available in [18].

### 5.4 Resetting a Secure Monotonic Counter

**Target.** The MIFARE Ultralight EV1 [10] is an HF memory tag with a few security features including three 24-bit monotonic counters with anti-tearing support. You can increment a counter with the command `INCR_CNT` by the value you want but you can never decrease it or reset it back to zero.

Each counter can be read with `READ_CNT` and is complemented by an 8-bit validity flag which can be read with the `CHECK_TEARING_EVENT` command. The flag should be equal to `0xBD`. Other values indicate that a tearing event occurred during an increment command but the (previous) counter value can still be read safely, so the counter is never corrupted. When a reader checks the counter and the flag, it can tell if the counter is up to date or if the last increment attempt failed, and act accordingly. It is explicitly allowed to send a dummy *increment by zero* to force a fresh write of the counter, e.g. after a tearing event.

The fact that the counter is never corrupted seems to indicate that the counter is stored in two memory slots and the flag indicates which counter is up to date. So the anti-tearing mechanism is somehow similar to the EM4305 and its Protection Words except that we cannot observe directly both slots and that the validity flag is not a single bit but a full byte with a specific value `0xBD`.

**Field reconnaissance.** As in the previous examples, launching a few tear-off experiments allows speculating about how the feature works internally.

To ease the reading, we firstly present what we understood but this came from conclusions on all the behaviors we observed through our experiments.

A counter is made internally of two slots, two EEPROM words. Each one comprises a 24-bit field to store the counter and an 8-bit flag which acts as a canary. Upon receipt of an increment command `INCR_CNT`, something along the following lines happens.

— Find the current counter value: if one flag is corrupted, read the other counter slot, else consider the highest of the two counter slots;
— Add to the highest counter the value sent as parameter of `INCR_CNT`;
— Erase the other counter slot – both the flag and the counter value – to zero;
— Write the erased counter slot with the new value and the flag `0xBD`.

A `READ_CNT` is similar to the first step of the `INCR_CNT` and returns the highest internal counter with a valid flag as being the current counter value.

A `CHECK_TEARING_EVENT` checks both flags, returns the corrupted one if any, else returns the flag of the smallest internal counter slot.

Table 7 illustrates the situations when the flag is correct or corrupted.

| Slot | Flag | Value | Active | ⟹ | READ_CNT | CHECK_TEAR... |
|---|---|---|---|---|---|---|
| A | 0xBD | 0x123456 | | | | 0xBD |
| B | 0xBD | 0x123457 | ★ | | 0x123457 | |
| A | 0xBD | 0x123456 | ★ | | 0x123456 | |
| B | 0x98 | 0x123457 | | | | 0x98 |

**Table 7.** MIFARE Ultralight EV1 counter internals and readouts in normal situation (top) and after a tearing event (bottom).

When both slots are identical, e.g. after a dummy increment, one slot gets priority over the other one depending on the internal architecture and is used to return the counter value. But when reading the flag, we presumably get the flag of the other counter slot.

As we have less visibility than for the EM4305 Protection Words, it would be nice to have a trick to know which slot is active.

We will refer to the two copies as `A` and `B`. Considering a counter set to value `0x123456` and issuing a dummy increment, internally the counter state will be as shown in Table 8.

Both flags are correct, so when the counter is read with `READ_CNT`, one gets priority. We will assume arbitrarily that we get B back, so `B:123456`. When the flag is read with `CHECK_TEARING_EVENT`, we get `A:BD`, the flag

| Slot | Flag | Value | Active |
|---|---|---|---|
| A | 0xBD | 0x123456 | |
| B | 0xBD | 0x123456 | ★ |

⟹

| READ_CNT | CHECK_TEAR... |
|---|---|
| | 0xBD |
| 0x123456 | |

**Table 8.** MIFARE Ultralight EV1 counter internals and readouts after a dummy increment.

of the unused counter slot as it is the one supposedly programmed during an increment event that could be torn.

Then we tear an increment with increasing timings and we observe how the flag gets progressively programmed. It is slowly flipping its bits from zero towards `0xBD`. So we can tell the EEPROM follows the convention of Observation 1 (Logic implementation) [A]: an erased word contains zeroes.

The intermediate values when a byte is written are specific to that memory area according to Observation 2 (Biased bits). So when we are in a situation where one flag is returned and we tear an `INCR_CNT` during its *write* operation at different timings, we will observe a set of intermediate values when reading the flag with `CHECK_TEARING_EVENT`. Here are the values observed on a real tag: `00`, `80`, `10`, `90`, `94`, `98`, `9C`, `9D`, `BC`, `BD`. While when the other flag is returned, we will observe another specific set of intermediate values such as, for the same tag, `00`, `80`, `88`, `8C`, `AC`, `AD`, `BD`. We can force a switch from one slot to the other by incrementing by one and we can force a switch to the priority slot by a dummy increment by zero. If after that dummy increment we see `00`, `80`, `88`, `8C`, `AC`, `AD`, `BC` and we assumed arbitrarily that `B` value gets priority, it means this series is the one of flag `A` (as `CHECK_TEARING_EVENT` always returns the *other slot* flag). Table 9 summarizes the observed values.

| Slot | Flag fingerprint | Priority slot |
|---|---|---|
| A | 80.88.8C.AC.AD | |
| B | 80.10.90.94.98.9C.9D.BC | ★ |

**Table 9.** Counter flags fingerprints observed for a given tag.

So we got a fine way to determine which slot is active, `A` or `B`, by tearing a dummy `INCR_CNT` and observing the intermediate flag values.

We made more experiments and when tearing an increment by one at the limit of getting a valid flag, the first issue we observed on a tag by repeating reads is that sometimes a `READ_CNT` returns the previous counter value, while `CHECK_TEARING_EVENT` always returns a valid flag!

We know by Observation 7 (Weak bits) that some bits can be weakly programmed. So a possible explanation is that one bit of the new counter gets weak and when it's seen as a 0, the other slot with old counter value becomes the largest one and is returned as the current counter value. This means that the flag is not written *after* the counter value but at the same time. The most probable explanation is that the 8-bit flag and the 24-bit counter are sharing the same 32-bit EEPROM word.

Moreover, we can control that weak bit by using Observation 8 (Distance effect on weak bits) and varying the distance to the reader: close to the reader we see the new counter value and far away (but still powered) we see the previous counter value.

**Strategy.** How about developing a plan to reset completely the counter back to zero? The idea is to find one single bit of the counter that gets written after all the flag bits and to program it weakly. So when the counter has a power of 2 as value with this sole bit set, if it flips to 0 during a *read*, the whole slot value becomes equal to 0.

- Increase the counter to the closest $2^N - 1$ value.
  If e.g. `0x00008F: INCR_CNT(0x70)` $\rightarrow$ `0x0000FF` ;
- `INCR_CNT(0)` so both slots are equal and `B` gets priority.
  e.g. `A:0x0000FF`, `B:0x0000FF` ;
- `INCR_CNT(1)` and tear near the end of the *write* operation. Hope for a counter set to $2^N$ with a weak bit.
  e.g. `A:0x000?00`, `B:0x0000FF` ;
- `INCR_CNT(0)` and tear before the *write* operation. Hope `A` is read as $2^N$ so `B` gets corrupted.
  e.g. `A:0x000?00` and B flag indicates B is corrupted ;
- `INCR_CNT(0)`. Hope `A` is read as `0x000000` so `A` is copied to `B`.
  e.g. `A:0x000?00`, `B:0x000000` and both flags are again valid ;
- `INCR_CNT(0)`. Hope `A` is read as `0x000000` so `B` is copied to `A` and they become both stable.
  e.g. `A:0x000000`, `B:0x000000` ;
- If it fails, try again. If the counter increases slightly with attempts, it's not a problem as once the attack is successful we can bump the counter back to $2^N - 1$ and try again ;
- If after a while there is no indication that bit $N$ can be weakly programmed, move to the next bit and bump the counter to $2^{N+1} - 1$.

Then complement this strategy to adjust automatically the timings and cover all the possible outcomes, including corrupted flags and weakly programmed flags.

It is also possible to alternate the targeted slot in the strategy by reaching $2^N - 2$ then issuing `INCR_CNT(0)` and `INCR_CNT(1)`, in order to scan the other slot as well in the quest for a bit written later than the flag bits, doubling the chances of success.

At some point, it is possible to get errors on `CHECK_TEARING_EVENT`, `READ_CNT` and `INCR_CNT` commands because both flags are internally read as invalid and the counter has become unusable. When it happens, trying different positions on the antenna and insisting on issuing an `INCR_CNT(1)` should fix the issue after a while. Such situation means one of the flags was seen as valid at some point and this should be possible to see it as valid once again, enough to fix the counter.

**Results.** We have demonstrated the possibility to reset completely a MIFARE Ultralight EV1 counter despites its anti-tearing features and we reported the issue to NXP. They confirmed the issue and we mutually agreed on a disclosure calendar.

According to NXP, the list of affected products is the following.

- In MIFARE Ultralight family:
  - MIFARE Ultralight EV1, MF0UL;
  - MIFARE Ultralight C, MF0ICU;
  - MIFARE Ultralight NANO, MF0UN.
- In NTAG 21x family:
  - NTAG 210($\mu$)/212: NT2L1, NT2H10, NT2H12;
  - NTAG 213 (TT/F) /215 /216 (F): N2H13, NT2H15, NT2H16.

None of these products are Common Criteria certified.

Other security features likely based on hidden slots might be affected by tearing events and weak bits too, such as OTP bits or Lock bits. But there is no visible flag to check which slot is active and to be used as canary to adjust timings. So it seems much harder to corrupt them and if the attack succeeds, it will probably clear only a few bits.

**Mitigations.** Consequently, an update of *Application Notes* AN11340 [11] and the new AN13089 [12] are proposing mitigations, such as doubling writes on OTP and Lock bits to update all internal slots and using the upper range of the counters. If e.g. an application needs a counter from `0x000000` to `0x000100`, use a counter from `0xfffeff` to `0xffffff`. And when it reaches `0xffffff`, issuing a dummy increment by zero to update the other internal slot will prevent any further attempt to affect the counter as it will only affect slot `A` while slot `B` will always have priority when the counter is read. Adding a MAC (*Message Authentication Code*) may help too but beware of rollback attacks.

The described attack has been realized with the generic tear-off tooling (`hw tearoff` combined with `hf 14a raw`) we added to the Proxmark3 RRG code repository as explained in Section 6.

To conclude this chapter we want to highlight that we appreciated the cooperation of NXP for a coordinated disclosure.

## 6 Proxmark3 Tear-Off Tooling

The various experiments described in this paper were performed with Proxmark3 RDV4 devices – a powerful general-purpose RFID tool designed to snoop, listen and emulate everything from Low Frequency (125 kHz) to High Frequency (13.56 MHz) tags – and the Proxmark3 RRG code repository [5]. Numerous commands were added for specific and generic tear-off experiments, as shown in Table 10.

| Chip/Standard | Command |
|---|---|
| ATA5577C | `lf t55xx dangerraw` |
| MIK640M2D | `hf mfu otptear` (automated) [a] |
| EM4x05 | `lf em 4x05_unlock` (automated) |
| EM4x05 | `hw tearoff` combined with `lf em 4x05_write` |
| EM4x50 | `hw tearoff` combined with `lf em 4x50_write` |
| ISO14443A | `hw tearoff` combined with `hf 14a raw` |
| ISO14443B | `hw tearoff` combined with `hf 14b raw` |
| ISO15693 | `hw tearoff` combined with `hf 15 raw` |
| iClass | `hw tearoff` combined with `hf iclass wrbl` |

**Table 10.** Proxmark3 RRG Tearing-Off Cheatsheet (as of 06/2021).

*a.* implemented by the authors of [2]

Besides the specific commands enabling the first three types of attack we described in the previous chapters, we also added a generic tear-off support as tear-off is becoming such an interesting vector in the Proxmark3 world.

To be able to experiment tear-off on various types of tags, there is now a `hw tearoff` command available to set a delay and to schedule a tear-off event during the next command supporting such tear-off. The list of commands is growing and today you can experiment tear-off against ISO14443-A, ISO14443-B and ISO15693 raw commands as well as iClass and EM4x05 writes. Listing 1 contains an example of how to use the generic tear-off command against the MIK640M2D.

```
% Block 3 current value:
  [usb] pm3 --> hf mfu rdbl -b 3
[=] Block#  | Data        | Ascii
[=] ----------------------------
[=] 03/0x03 | FF FF FF FF | ....

% Trying to write zeroes in it...
[usb] pm3 --> hf mfu wrbl -b 3 -d 00000000

% It fails as block 3 is OTP
[usb] pm3 --> hf mfu rdbl -b 3
[=] Block#  | Data        | Ascii
[=] ----------------------------
[=] 03/0x03 | FF FF FF FF | ....

% Now scheduling a tearing event
[usb] pm3 --> hw tearoff --delay 300 --on
[=] Tear-off hook configured with delay of 300 us
[=] Tear-off hook enabled

% Trying to write arbitrary value on block 3 in raw mode
[usb] pm3 --> hf 14a raw -sc a203ffffffff
[#] Tear-off triggered!

% Tadaam, OTP block is reset!
[usb] pm3 --> hf mfu rdbl -b 3
[=] Block#  | Data        | Ascii
[=] ----------------------------
[=] 03/0x03 | 00 00 00 00 | ....
```

**Listing 1.** Generic tear-off against MIK640M2D.

If nevertheless you want to execute a tear-off on a command not yet ready for it, adding support becomes as easy as adding a couple of lines in the firmware code, just after the command triggering an EEPROM operation has been sent to the tag, as shown in Listing 2.

```
if (tearoff_hook() == PM3_ETEAROFF) {
    // tear-off performed. Clean stuff and return
} else {
    // do as usual: read reply etc.
}
```

**Listing 2.** Generic tear-off hook to insert in the firmware.

Typically one can use the generic tear-off command and hook when poking at a new target to characterize it and when testing attack strategies on it, before writing more specific and automated tools.

## 7 Conclusion and Future Research

Based on our observations against many different tags, we have detailed several effects that may happen when an EEPROM *erase* or *write* operation is interrupted. We have explained how these effects can be leveraged to understand various security features internals and to defeat them. We have described real life examples on four different HF and LF RFID tags, from different manufacturers.

In the most secure example, the MIFARE Ultralight EV1, we have seen the difficulty to predict some weird behaviors due to weak bits especially when a security function relies on these bits and reads different values depending on the context. This is a challenge for the designers and an opportunity for the attackers even on secure chips if the designer did not think of all the possibilities.

A better understanding and model of the probabilities behind the reading threshold mechanism of a bit, their distribution across the bits of a component, across components of a same family and under different environmental conditions would be a valuable tool for further investigations.

The same type of issues might probably be found on EEPROMs contained in other non-RFID devices as well. This requires transposing the physical tearing-off variables to connected chips: under-powering the chip as an equivalent to bringing the RFID tag as far as possible from the reader, a cut in the EEPROM voltage supply would mimic a tear-off, etc.

These research activities were also the opportunity to add a generic tool in the Proxmark3 code, in order to enable fast prototyping of tearing-off experiments on the whole range of LF and HF tags supported by the Proxmark3.

Now, your turn: find other interesting targets, in RFID/NFC or in other domains, get inspired by the tear-off strategies we presented, make use of our generic tear-off tooling and hopefully get some interesting results to share as well!

We want to thank all the reviewers for their very valuable comments.

## References

1. EM Microelectronic. EM4205/4305 LF Animal & Access ICs. `https://www.emmicroelectronic.com/product/lf-animal-access-ics/em42054305`.
2. N. Grisolía and F. G. Ukmar. Estudio del comportamiento ante fallas en memorias EEPROM aplicadas en dispositivos RFID/NFC, 2020. `http://proxmark.org/files/Documents/13.56%20MHz%20-%20MIFARE%20Ultralight/PFI%20-%20Federico%20Gabriel%20Ukmar%20LU1052979%20-%20Nahuel%20Grisolia%20LU1038395%20-%20Ingenier%c3%ada%20Inform%c3%a1tica.pdf`.

3. P. Gutmann. Data remanence in semiconductor devices. In *10th USENIX Security Symposium (USENIX Security 01)*, Washington, D.C., August 2001. USENIX Association. https://www.usenix.org/conference/10th-usenix-security-symposium/data-remanence-semiconductor-devices.
4. T. Hanyu, N. Kanagawa, and M. Kameyama. Design of a one-transistor-cell multiple-valued cam. *IEEE Journal of Solid-State Circuits*, 31(11):1669–1674, 1996.
5. C. Herrmann, P. Teuwen, O. Moiseenko, M. Walker, et al. RRG / Iceman repo – Proxmark3. https://github.com/RfidResearchGroup/proxmark3.
6. Infineon. my-d™ move and my-d™ move NFC. https://www.infineon.com/cms/en/product/security-smart-card-solutions/contactless-memories/my-d-move-and-my-d-move-nfc/.
7. *Marshmellow* et al. Proxmark3 developers community – ata55x7 test mode. https://web.archive.org/web/20200919223527/http://proxmark.org/forum/viewtopic.php?id=4717.
8. Microchip. ATA5577M1C Device Overview. https://www.microchip.com/wwwproducts/en/ATA5577M1C.
9. Mikron. RFID Chip MIK1312ED Model MIK640M2D. https://mikron.ru/products/rfid-chip-inlays-maps/rfid-chips/product/rfid-chip-mik1312ed/.
10. NXP. MIFARE Ultralight® EV1. https://www.nxp.com/products/rfid-nfc/mifare-hf/mifare-ultralight/mifare-ultralight-ev1:MF0ULX1.
11. NXP. AN11340 MIFARE Ultralight EV1 Features and Hints - Rev. 3.2, May 2021. https://www.nxp.com/docs/en/application-note/AN11340.pdf.
12. NXP. AN13089 NTAG 21x Features and Hints - Rev. 1.0, May 2021. https://www.nxp.com/docs/en/application-note/AN13089.pdf.
13. C. Pavlina, J. Torrey, and K. Temkin. Abstract: Characterizing eeprom for usage as a ubiquitous puf source. In *HOST 2017*, pages 168–168, 2017.
14. Shanghai Feiju Microelectronics Co. Ultra-low-cost medium-capacity NFC chip F8213. http://www.nfcic.com/index.php?_m=mod_product&_a=view&p_id=102.
15. STMicroelectronics. ST512 ISO14443-B Tag IC with 2 binary counters, 5 OTP blocks and anti-collision with 512-bit. https://www.st.com/en/nfc/sri512.html.
16. W. Teepe. Making the best of MIFARE Classic, 2008.
17. P. Teuwen. EEPROM: When Tearing-Off Becomes a Security Issue, 2019. https://blog.quarkslab.com/eeprom-when-tearing-off-becomes-a-security-issue.html.
18. P. Teuwen and C. Herrmann. RFID: New Proxmark3 Tear-Off Features and New Findings, 2020. https://blog.quarkslab.com/rfid-new-proxmark3-tear-off-features-and-new-findings.html.