MIFARE Classic “Offline Nested Authentication” Attack

Contents

Introduction and background 3
MIFARE Classic authentication basics 4
The nested authentication attack 7
How the attack leverages nonces, parity, and weak PRNG 12
Real-world tools: MFCUK and MFOC 14
Conclusion 15
Bibliography 16

Introduction and background

What is the offline nested attack?

MIFARE classic cards store data in sectors protected by 48-bit secret keys (two keys per sector, Key A and Key B). Normally you must know a sector's key to authenticate and access that sector's data. The offline nested authentication attack is a clever way to recover *all* the keys on a MIFARE card by using just one known key as a starting point. In simple terms, if you already have one valid key (for one sector), you can trick the card into revealing information that lets you gradually deduce the other keys without any help from a legitimate reader [1] [2]. This attack is "offline" in the sense that after you collect some data from the card via some wireless queries, the heavy work of cracking the key is done on your computer, not on the card.

To perform the nested attack, the attacker must have at least one known key for the card. Often, one key is known because some MIFARE Classic systems use default keys (like FFFFFFFFFFFF or other well-known values) or perhaps the attacker already cracked one key using another method [1] [3]. If no key is known at first, attackers can use a slower preliminary attack (for example the "Darkside" attack implemented in the MFCUK tool) to recover one key by exploiting error messages and parity bits (more on this later) [4]. Once a single valid key is obtained, the faster nested attack (often using a tool called MFOC) can be used to retrieve all the remaining keys [1] [3].

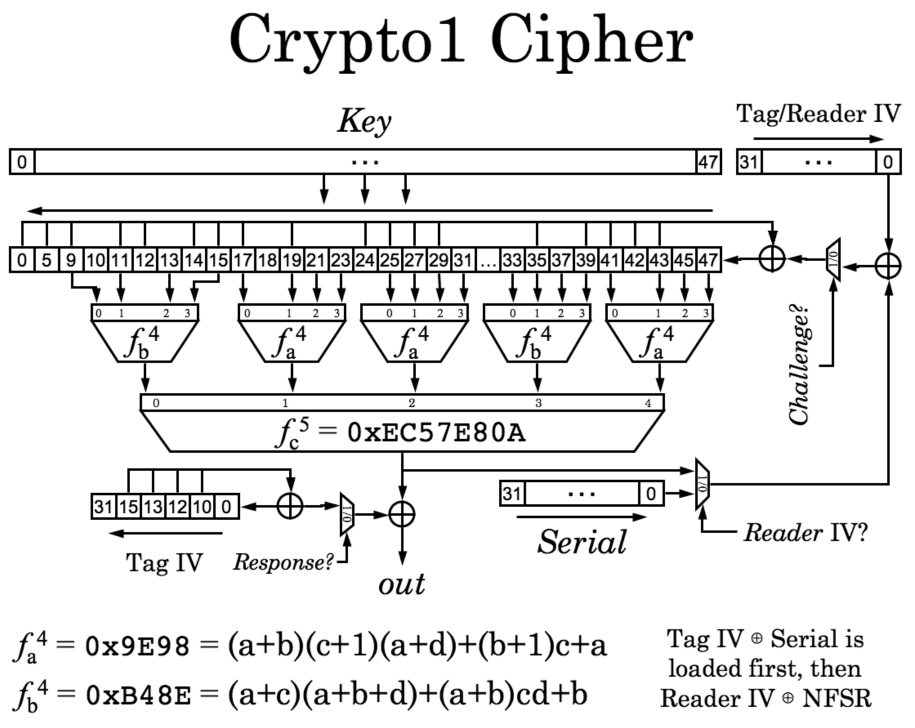
Before diving into the steps, we need to review how MIFARE Classic authentication works as well as some key concepts (nonces, keystream, parity bits) that this attack abuses.

MIFARE Classic authentication basics

MIFARE Classic uses a challenge-response authentication with a proprietary stream cipher called Crypto1 (48-bit keys). When a reader wants to authenticate to a sector of the card, the sequence (simplified) is:

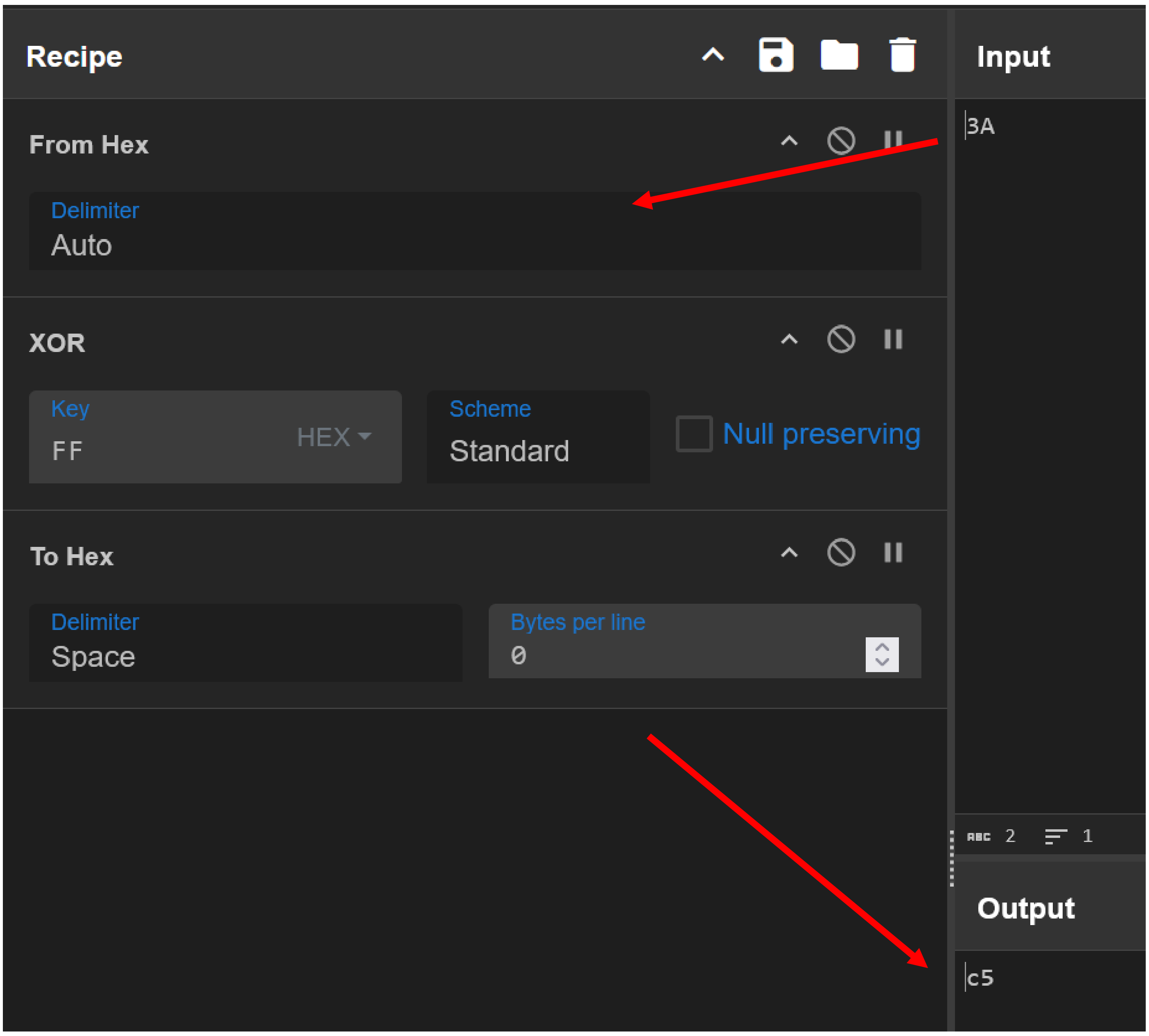
1. Reader -> Card: “I want to authenticate to Sector X with Key A/B.”
2. Card -> Reader: Card generates a random 32-bit number called a nonce (let’s call it N_{Card}) and sends it to the reader as a challenge. This nonce is essentially a random “puzzle piece” the reader must encrypt correctly to prove it knows the sector key. In the very *first* authentication of a session, this nonce is sent in plaintext (unencrypted) [4].
3. Reader -> Card: The reader computes a response using N_{Card} and its own random number N_{Reader}, encrypting these with the shared key. (Note: the reader encrypts N_{Card} and a random N_{Reader} under the sector key and sends them.)
4. Card -> Reader: The card verifies the reader’s response. If correct, it then sends back an encrypted version of N_{Reader} to authenticate itself to the reader. Now both sides trust each other and share an encrypted session. At this point, the reader is “logged in” to that sector and can read/write data. All further communications are encrypted with the keystream generated by Crypto1.

The role of the nonce: The nonce from the card N_T is a fresh random challenge each time to ensure the reader really knows the key for that session. It prevents replay attacks because the reader’s response must match that specific random challenge. In Crypto1, this nonce also seeds the cipher’s keystream generator.


Crypto1 Cipher
Key
Tag/Reader IV
Challenge?
f_c^5 = 0xEC57E80A
Tag IV
Response?
out
Serial
Reader IV?
f_a^4 = 0x9E98 = (a+b)(c+1)(a+d)+(b+1)c+a
f_b^4 = 0xB48E = (a+c)(a+b+d)+(a+b)cd+b
Tag IV ⊕ Serial is loaded first, then Reader IV ⊕ NFSR

A critical fact about MIFARE Classic is that its random number generator is not truly random – it's a simple linear feedback shift register (LFSR) that cycles through values in a predictable way [2]. In fact, there are only 2^{16} (65,536) possible nonces, and the sequence repeats every ~618 milliseconds if you continuously power-cycle the card [2]. This pseudo-random number generator (PRNG) is one key weakness the attack will exploit.

Keystream and encryption: Crypto1 produces a keystream, which is a sequence of pseudo-random bits derived from the secret key and the nonce. The plaintext messages are XORed with this keystream to produce the ciphertext. XOR has the useful property that if you know the plaintext and the ciphertext, you can derive the keystream. For example, if a plaintext byte was $0x3A$ and the ciphertext byte is $0xC5$, then the keystream byte must be $0xFF$ because $0x3A \oplus 0xFF = 0xC5$.


Recipe

From Hex
Delimiter: Auto

XOR
Key: FF (HEX)
Scheme: Standard
Null preserving: ☐

To Hex
Delimiter: Space
Bytes per line: 0

Input
```
3A
```

Output
```
c5
```

In MIFARE Classic’s auth, once the card and reader are synchronized, all further messages (including subsequent nonces) are encrypted by XORing with the keystream.

Parity bits: Every byte transmitted in MIFARE Classic comes with an extra parity bit for error detection (ensuring the number of ‘1’ bits in that byte is either even or odd as expected). Normally, parity bits are just computed over plaintext and not encrypted. However, MIFARE Classic’s designers made a mistake: the parity bit is computed on the plaintext but then the parity bit itself is transmitted without proper encryption – the first bit of the keystream for the next byte is used to encrypt the parity bit of the current byte [1]. This means if an attacker knows or guesses something about the plaintext, they can often detect it via parity, or vice versa. In short, the parity mechanism “leaks” a tiny bit of information about the encrypted data [1]. Attackers will use this leak as a side-channel to narrow down guesses (both for nonces and for keys), as we’ll see.

The nested authentication attack

Step 1: Use a known key to authenticate (initial sector)

The attacker begins by authenticating to a sector of the card for which they already know the key (either a default key or one recovered earlier). This is the “entry point.” For example, suppose the attacker knows Key A for Sector 0 (commonly sector 0 has default keys in some systems). The attacker poses as a reader and sends an auth request for Sector 0 using Key A. The card responds with a random nonce $N1$. Because this is the first authentication of the session, $N1$ is sent in plaintext over the air [4]. The attacker captures this $N1$.

- *Why do this?* We need to establish an encrypted session with the card using a known key so that we can perform the nested auth next. Capturing $N1$ also gives us a reference point for the card’s random number state. Since the card’s PRNG is predictable, knowing one nonce will help us predict the next one.

At this point, the attacker (as a reader) completes the handshake for Sector 0 using the known key, proving knowledge of the key, and the card and attacker now have an encrypted channel based on Key A of Sector 0. Any further commands the attacker sends will be encrypted with the keystream derived from Key A.

Step 2: Send a nested authentication request for target sector (unknown key)

Now the real trick: The attacker immediately initiates another authentication, but this time for a different sector (say Sector Y) whose key they want to recover. Crucially, the attacker does *not* yet know the key for Sector Y – that’s the target of the attack. However, because the attacker is still in the middle of an encrypted session from Step 1, they don’t just send a normal auth request. Instead, they send the auth command for Sector Y encrypted under the current session (which is using the known Key A of Sector 0).

What happens inside the card is interesting: The card receives the encrypted command from the attacker: “authenticate to Sector Y”. Since the card is currently decrypting everything with the Sector 0 key, it will decrypt that command and see “Oh, the reader wants to authenticate to Sector Y now.” The card then resets its internal cipher state to use Sector Y’s key for the next authentication round [1]. In other words, the card prepares to perform a fresh auth for Sector Y, and from this point, the Crypto1 cipher is now re-initialized with the (unknown) key for Sector Y. This is by design in MIFARE – a reader can authenticate to a new sector without fully powering down the card, and the card will switch keys internally.

Importantly, because this auth command was issued inside an encrypted session, the card will treat the new authentication as if it's also within encryption. As a result, the challenge nonce for Sector Y, call it $\boldsymbol{N2}$, will be sent encrypted rather than in plaintext [5]. This is the core of the "nested" trick: we force the card to encrypt its next challenge using a key we *don't* know, and we'll exploit that.

So in summary for this step: The attacker sends an encrypted "auth to Sector Y" command; the card switches to Key Y and generates a random challenge $N2$ for the new auth, and sends encrypted $\{\boldsymbol{N2}\}$ (braces denote encryption) to the attacker. The attacker now has an encrypted blob that represents the card's random number $N2$ XORed with some keystream derived from Key Y. At this moment, the attacker does not know $N2$ (it's scrambled) and doesn't know Key Y either – on the surface this looks hopeless. But the next steps show how the attacker pries out the value of $N2$ and, eventually, the key.

Step 3: Predict the card's random nonce ($N2$) using the weak PRNG

Because of the weak random number generator, the attacker has a very good chance of figuring out what $N2$ actually is (even though they only received it in encrypted form). Here's how:

- The attacker knows the previous nonce $N1$ (from Step 1, in plaintext). They also likely know when $N2$ was generated relative to $N1$ (by precise timing of their commands). In MIFARE Classic's PRNG, if you know one output and the time between outputs, you can often predict the next output because the RNG state advances in a simple, deterministic way each clock tick [2]. Essentially the "distance" (number of LFSR steps) between $N1$ and $N2$ depends on how quickly the attacker issued the second auth. Attackers can deliberately control or estimate this timing to narrow down the possibilities for $N2$. For example, if the attacker starts the second auth almost immediately, $N2$ might just be the next value in the RNG sequence after $N1$ (or after a fixed small number of cycles), making it highly predictable.
- Moreover, the PRNG only has 2^{16} possible values [2]. This is a tiny space by cryptographic standards. The attacker can brute-force all 65,536 possibilities if needed and test which one makes sense. In practice they don't even have to try them all – they can combine this with parity checks to eliminate many of them quickly.
- Parity bit clues: Remember those parity bits attached to each byte? The encrypted nonce $\{N2\}$ comes with parity bits that were calculated from the *plaintext* $N2$. The attacker can see these parity bits over the air. Because the parity is computed on the plaintext but one bit of keystream was used to encrypt each parity bit, not all bit patterns of $\{N2\}$ are possible for a given plaintext $N2$. In fact, by examining the parity of the encrypted bytes, the attacker gains 3 bits of information about $N2$, reducing the uncertainty by a factor of 8 [1] [2]. Put

simply, some candidates of $N2$ are inconsistent with the observed parity bits, so the attacker can throw those out.

Combining these factors, the attacker can deduce the actual nonce **$N2$** (or at worst, narrow it down to a very small handful of possibilities). In many cases, the timing plus parity analysis makes $N2$ essentially predictable to the attacker [4] [2].

For example, suppose after timing and parity filtering, the attacker believes the card's second random challenge $N2$ is likely $0x5A3C1F08$ (just as a hypothetical value). If they're correct, they now effectively know the plaintext that the card generated for the second challenge.

Step 4: Derive the keystream used for the encrypted nonce

Now comes a crucial payoff: if the attacker knows the plaintext $N2$ (from Step 3's prediction) and they have the ciphertext (the encrypted $\{N2\}$ captured from the card), they can compute the keystream that was used to encrypt $N2$. This is done by a simple XOR:

$$\text{Keystream bits} = N2 \oplus \{N2\}$$

Because plaintext ⊕ keystream = ciphertext, rearranging this gives keystream = plaintext ⊕ ciphertext. This operation yields 32 bits of the keystream produced by the unknown Key Y during that authentication attempt [2]. In other words, the attacker now knows the exact sequence of 32 bits that Crypto1 generated when it was initialized with Key Y and when the card nonce was $N2$.

Why is this valuable? Because those 32 keystream bits carry a lot of information about Key Y. Crypto1's keystream generator is a 48-bit LFSR with a filtering function, and knowing 32 output bits (especially output bits that correspond to the start of an authentication session) puts heavy constraints on the internal state (which includes the secret key). In fact, at this point the search space for the key has been reduced from 2^{48} down to 2^{16} possibilities [2]. In terms of scale, it's like having solved 99% of a thousand-piece puzzle with only a single small piece missing.

To clarify: the attacker has not directly "decrypted" anything yet (they don't have the sector's data), but they have a slice of the cipher's output (keystream) that they shouldn't normally have. This is like knowing how a padlock's pins align; it dramatically cuts down the effort to replicate the actual key.

Technical note: The Crypto1's cipher design allows an attacker who knows one nonce and the corresponding 32-bit keystream to perform a "key recovery" attack. This can involve things like solving for the LFSR state or simply brute forcing the remaining unknown key bits. Since 16 unknown bits remain (2^{16} combinations), a brute force is very feasible on a modern PC (on the order of milliseconds to minutes, depending on optimizations)

[2] [5]. There are also analytical techniques to derive the key faster than brute force using the structure of Crypto1 (e.g. direct inversion using the known keystream bits [1]), but the end result is the same: the attacker can find the key relatively quickly.

Step 5: Recover the unknown sector key (brute-force)

Armed with the 32-bit keystream segment (and the knowledge of $N2$), the attacker now works offline (on their computer) to find which 48-bit key could have produced that keystream. Essentially, the attacker is solving for Key Y. Because of the information gained, this is a much smaller search problem than trying 2^{48} keys blindly.

The attacker can either:

- Brute force test: Try each candidate 48-bit key, simulate the Crypto1 cipher's first 32 bits of keystream for nonce $N2$, and see if it matches the captured keystream. The correct key will produce exactly the keystream that was observed. This is a check that at most 65,536 keys need to be tried, which is extremely fast (on the order of 0.05 seconds in software, or even faster with specialized hardware) [5].
- Direct computation: Use the mathematical structure of Crypto1 to deduce the key bits. For instance, researchers found that only the odd-numbered bits of the LFSR are involved in generating those first 32 keystream bits, allowing a direct recovery of those bits of the key and leaving only 2^{16} possibilities for the even bits [1]. The result again is about 65k candidates to test.

In practice, attackers often repeat the nested auth process a few times with *different nonces* to absolutely confirm the key and eliminate any remaining ambiguity [2]. For example, they might perform Steps 2–4 two or three more times (getting new random $N3$, $N4$, etc., each time deriving another 32-bit keystream slice under the same unknown key). The true key will be the one candidate that is consistent with *all* the observed keystream pieces. Each new attempt dramatically reduces the candidate pool – after even 2 attempts, the correct key is usually obvious. In fact, the literature shows that about 6 successful nonce captures (with parity guesses) are enough to uniquely determine the 48-bit key with high confidence [1] [5]. Often, it's even faster; one high-level description notes that *"repeating the process two to three more times"* is enough to conclude the key [2].

At this stage, the attacker has recovered the secret Key for Sector Y. They can verify it by authenticating normally to that sector (outside the nested context) to ensure the card accepts it. Once verified, the attacker now knows another sector's key without ever having had it given to them.

Step 6: Repeat the attack for other sectors

With Key Y recovered, the attacker can rinse and repeat the process for other sectors on the card [1]. Each newly recovered key becomes a new “known key” that can be used to mount a nested attack on yet another sector. The attacker can systematically go through all 16 sectors of a MIFARE Classic 1K card (for example) until all keys are known. This means the attacker has effectively “dumped” the entire card’s secrets. At this point, they can read or modify any data on the card or even clone the card completely since they possess every key.

A few notes on efficiency and real-world use:

- Often, many sectors might share keys (some deployments use the same key for multiple sectors), so in reality you might not need to do separate attacks for every single one – a few key recoveries can unlock multiple sectors.
- The nested attack is very fast for each sector once one key is known. Literature reports that recovering a new sector key via nested attack *“only requires about 8 authentication attempts”* on average [6] – this is a matter of milliseconds. In practice, some tools have been able to retrieve all of a single MIFARE Classic card’s keys in seconds. For example, one demonstration combined first-key recovery and nested attacks to clone a card in under 10 seconds [7].

How the attack leverages nonces, parity, and weak PRNG

Here we will recap the key points of why this attack works, emphasizing specific roles:

- Nonces (card-generated random numbers): The nonce is supposed to add unpredictability. In this attack, however, the nonce becomes a tool for the attacker. By capturing one nonce in plaintext and forcing the next one to be encrypted, the attacker creates a known-plaintext scenario (they eventually know $N2$ and see $\{N2\}$). The nonce essentially gives the attacker a chunk of *cipher input vs output* to analyze. The fact that MIFARE's nonce space is small (16-bit effective entropy) and repeats predictably is a huge weakness [2]. The attacker capitalizes on this by predicting the nonce and leveraging it to get a keystream. Without the nonce, the attacker would have no starting point; without the weak RNG, the attacker couldn't guess it so easily.
- Encrypted responses and keystream: When the card sends an encrypted challenge $\{N2\}$, it's using the secret key's keystream – effectively "showing" the attacker the result of XORing something secret with something known. Once the attacker figures out the plaintext behind that encrypted response, the keystream is revealed [2]. The keystream is crucial because it ties directly to the secret key's state. Think of the keystream as a fingerprint of the key: if you have enough of it, you can identify the key (since only the correct key would produce that exact keystream). The offline nested attack is essentially a way to extract a segment of keystream from a card without knowing the key, by using a known plaintext nonce.
- Parity bits: Parity bits play two roles in these attacks.
  a) During the nested attack's nonce prediction, the parity of the encrypted nonce leaks partial information about the plaintext nonce [1], cutting down the guesswork significantly. It's like a crossword puzzle hint – you know some letters must fit a pattern. Those 3 bits of leakage (for a 32-bit nonce) might not sound like much, but reducing possibilities from 65k to about 8k is a big jump in efficiency [1].
  b) During key recovery (especially the Darkside variant or if the attacker intentionally sends wrong authentication data), parity bits can be used to provoke the card into giving an encrypted error code. If the attacker guesses the parity bits correctly for a response but the response itself is wrong, the card will send a 4-bit NACK (negative acknowledgement) encrypted with the keystream [4]. Since the NACK is a small known value (typically $0x5$ denoting auth failure), the attacker can XOR the encrypted NACK with the known plaintext NACK to get 4 bits of keystream. This actually leaks 12 bits of the secret key's state at a time because those 4 keystream bits correspond to 12 bits of entropy of the 48-bit key in Crypto1

[1]. By repeatedly attempting auth with randomly guessed (incorrect) data but correct parity, eventually (on average 1/256 tries) the parity will by chance be right and yield an encrypted NACK [1]. Do this a few times (approx. 6 successful NACKs) and you accumulate enough keystream bits to deduce the entire key with a brute force [1]. This *“parity-NACK”* approach is exactly how the MFCUK tool recovers a key when none are known: it’s slower, but it works by leakage. In summary, parity bits – meant for error checking – ironically become an information leak that both speeds up nonce guessing and even allows direct key leakage via error messages.

c) Weak PRNG: The pseudo-random number generator in the card is an LFSR that always starts from the same state on power-up [4]. This means if you can time when the card was powered (or you power it yourself), you have essentially a deterministic sequence of “random” numbers. Even without power-cycling, the RNG has a short cycle. The attack takes advantage of this by tightly timing the two authentications. As J. Feng’s card hacking summary puts it, *“the distance between challenge nonces used in consecutive attempts strongly depends on the time between those attempts”*, making the nonce predictable [2]. The short period (2^{16}) further ensures the card’s random output isn’t very surprising [2]. In cryptography, a good RNG should never be predictable – here, predictability breaks the security. The attacker effectively “knows what the card will do next” in terms of random challenges, which undermines the whole premise of a challenge-response.

Real-world tools: MFCUK and MFOC

In practical scenarios, attackers don’t perform the above steps manually; they use specialized hardware and software:

- MFCUK (MiFare Classic Universal Toolkit): This is an open-source tool that implements the Darkside attack (among others) to recover at least one key from the card when you start with nothing. It automates the process of sending many authentication attempts with guessed parity to induce encrypted error codes and uses those to brute-force a key [4]. In essence, MFCUK will find one sector key by exploiting the parity/NACK vulnerability. This may take a bit of time (several minutes in some cases, as it involves thousands of attempts), but it requires only the card and no prior knowledge of a key. Once one key is obtained, the attacker can switch to the faster nested attack for the rest.
- MFOC (MiFare Classic Offline Cracker): This tool automates the offline nested authentication attack. Given one known key and access to the card, MFOC will perform the nested auth on other sectors, gather nonces and encrypted responses, and calculate the unknown keys. MFOC is extremely fast at recovering keys from MIFARE Classic cards [3].

In practice, tools like the Proxmark3 (a powerful RFID/NFC device) have scripts or commands (e.g. `hf mf nested`) that essentially invoke these techniques. For example, using Proxmark3, an attacker can run a command that first uses MFCUK logic to grab a key, then immediately uses MFOC logic to dump the rest of the keys. The output is a list of all sector keys and can often be obtained in a matter of seconds.

To illustrate, security researchers demonstrated that combining a quick Darkside attack for the first key (~300 queries to the card) with the nested attack for subsequent keys allows cloning a MIFARE Classic card in under 10 seconds [7]. This was far faster than older methods and raised awareness that no MIFARE Classic card is safe from a determined attacker.

Conclusion

The offline nested authentication attack shows how a combination of minor weaknesses can be exploited in clever ways to break a cryptographic system. In this document we looked at simple analogies and step-by-step reasoning to see how an attacker can start with one sector key and end up with *all* the keys to the card.

In summary, the attacker leverages a known key to get inside an encrypted session, nests a new auth to trick the card into encrypting a fresh random challenge with an unknown key, then uses knowledge of the card’s poor RNG and parity leakage to figure out that challenge. This gives a slice of the keystream, which is the golden thread leading back to the unknown key. With a bit of calculating, the key is recovered. Repeating this process opens up the entire card.

To someone with limited cryptographic background, it’s amazing (and perhaps alarming) to realize that by simply asking the card a few clever questions, one can completely break its security. The takeaway is that even devices marketed as secure (like MIFARE Classic) can have design flaws that render their security ineffective. Attacks like this show the importance of properly designed encryption protocols and not just relying on “security by obscurity.” MIFARE Classic has been deprecated in favour of more secure applications and newer card standards like MIFARE Plus and MIFARE DESFire which use stronger protocols to prevent these kinds of attacks.

Bibliography

[1] kaiya, "How Mifare classic nested attack works?," 09 06 2021. [Online]. Available: https://security.stackexchange.com/questions/107450/how-mifare-classic-nested-attack-works. [Accessed 27 03 2025].

[2] J. Feng, "Hacking a Stored Value Card," 05 08 2020. [Online]. Available: https://medium.com/csg-govtech/hacking-a-stored-value-card-4ee0a9934d2b. [Accessed 30 03 2025].

[3] gelotus, "MFOC Readme," 07 07 2020. [Online]. Available: https://github.com/nfc-tools/mfoc-hardnested/blob/master/README.md. [Accessed 30 03 2025].

[4] L. d. Araújo, "Study of vulnerabilities in MIFARE Classic cards," 22 09 2023. [Online]. Available: https://www.sidechannel.blog/en/mifare-classic-2/. [Accessed 30 03 2025].

[5] e. a. Flavio D. Garcia, "Wirelessly Pickpocketing a Mifare Classic Card," 17 05 2009. [Online]. Available: https://www.cs.umd.edu/~jkatz/security/downloads/Mifare3.pdf. [Accessed 01 04 2025].

[6] e. a. Ya Liu, "Legitimate-reader-only attack on MIFARE Classic," *Mathematical and Computer Modelling*, vol. 58, no. 1-2, pp. 219-226, 2013.

[7] "MIFARE," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/MIFARE. [Accessed 02 04 2025].