Reverse-Engineering the INE Credential QR Codes:

7 Layers of Cryptography in an ARM64 Library

Eduardo Dorantes
Independent Security Researcher
github.com/doranteseduardo · March 2026

ABSTRACT

This paper presents an independent security analysis of the cryptographic system embedded within the QR codes printed on Mexican INE (Instituto Nacional Electoral) voter credentials. Through static binary analysis and ARM64 emulation, the author reverse-engineered a seven-layer cryptographic pipeline implemented in a native ARM64 shared library. The pipeline combines AES-CBC-256, chained 8,192-bit RSA operations, proprietary SIMD/NEON-based key derivation, and ECDSA signature verification. A fully functional decoder was implemented using the Unicorn Engine emulator, processing a credential photograph in under three seconds. All findings are based solely on publicly accessible APK artefacts. No cryptographic keys, personal data, or private material are disclosed.

Keywords: *reverse engineering, cryptography, ARM64, AES, RSA, ECDSA, SIMD/NEON, Unicorn Engine, mobile security, identity documents, INE, Mexico*

1. INTRODUCTION

The INE (Instituto Nacional Electoral de México) voter credential carries two QR codes on its reverse side. The official application *mx.ine.rfe.lectorqr* scans these codes and renders the holder's biographical data and photograph. From the outside, the interaction appears straightforward: scan, display. Under the bonnet, however, the process involves a sophisticated cryptographic pipeline spanning seven distinct layers, implemented entirely in native ARM64 machine code.

This research was conducted independently, with no affiliation to the INE or any government body, and with no access to internal documentation or private keys. The sole artefact analysed was the publicly downloadable Android APK. The goal was to understand the architecture of the system — not to undermine it.

2. QR CODE STRUCTURE

Each INE credential carries **two binary QR codes** (left and right), each encoding exactly **858 bytes** of raw binary data. The first two bytes constitute a side identifier (0 = left, 1 = right). The remaining 856 bytes form the payload.

The payloads of both QR codes are concatenated to produce a 1,712-byte buffer, subsequently partitioned into two logical regions:

- **buf1** (688 bytes) — material used for runtime key derivation
- **buf2** (1,024 bytes) — encrypted biographical data

Both regions are passed as a single byte array to the native function `pc()`, which returns an opaque base64-encoded string containing the fully decrypted payload.

---

3. THE NATIVE LIBRARY

The cryptographic workload is handled by `libPersonalCode.so`, a 4.6 MB ARM64 shared library. Static analysis revealed that the **Chilkat cryptographic library** is statically linked into the binary, contributing over 1,000 internal functions. The library also incorporates a licence-enforcement mechanism that disables cryptographic operations in the absence of a valid activation key.

The Java-side interface exposes three JNI methods:

```
private final native String pc(Activity act, byte[] data); // decrypt pipeline

private final native String gTxt(String data); // extract text fields

private final native Bitmap gImg(String data); // extract photograph
```

---

4. THE CRYPTOGRAPHIC PIPELINE

The function `pc()` executes a strictly sequential seven-layer decryption pipeline. Each layer consumes the output of the previous one; no layer can be bypassed.

Layers 1–2: AES-CBC-256 and First RSA Round (Static Keys)

Encrypted constants stored in the `.rodata` section of the ELF binary are decrypted using a static **AES-CBC-256** key. This yields two artefacts: an RSA public key in XML format (*key #1*) and an encrypted base64 blob. Key #1 is used to decrypt the blob, revealing a second RSA key (*key #2*). All RSA keys in the system are **8,192 bits** — an unusually large size that imposes substantial computational cost on each operation.

Layer 3: RSA Decryption of buf2

Key #2 is applied to the 1,024 bytes of `buf2` from the QR payload, producing an intermediate ciphertext that is not independently interpretable.

Layer 4: Proprietary SIMD/NEON Key Derivation

This layer constitutes the most technically distinctive component of the pipeline. The library employs **ARM64 SIMD/NEON instructions** to derive a new AES key and initialisation vector from the combination of `buf1` and the result of Layer 3. The algorithm is entirely proprietary: it manipulates 128-bit vector registers in a sequence that cannot be re-implemented by reading the disassembly alone. The instructions must be *executed*.

Layers 5–6: Second AES and RSA Round (Runtime Keys)

Using the SIMD-derived keys, a second round of AES decryption retrieves RSA keys #3 and #4 from further `.rodata` constants. These keys perform a second RSA decryption pass over `buf2`. The structure mirrors Layers 1–3, but the keys are only materialised at runtime.

Layer 7: ECDSA Verification and Final Assembly

A third SIMD derivation round produces keys used to decrypt an embedded **ECC public key (secp256r1)**. This key verifies an **ECDSA signature** over the decrypted data, providing cryptographic integrity assurance. The verified plaintext is then base64-encoded and returned to the Java layer.

---

5. THE RSA KEY CHAIN

The architecture enforces a strict linear dependency between all cryptographic materials. The chain is summarised below:

```
Static AES key

-> RSA key #1 (from .so)

-> RSA key #2 (derived)

-> Partial decryption of buf2

-> SIMD key derivation (1st)

-> RSA key #3 (from .so, re-keyed AES)

-> RSA key #4 (derived)

-> Full decryption of buf2

-> SIMD key derivation (2nd)

-> RSA key #5 -> ECC public key

-> ECDSA verification
```

---

6. EMULATION APPROACH

Given that the SIMD/NEON layers cannot be faithfully re-implemented in software, the author opted to **emulate the ARM64 library in its entirety** using **Unicorn Engine v2**. The emulation environment:

- Loads the ELF binary with full GOT/PLT relocation (1,071 internal + 183 external symbols)
- Provides a Python implementation of the required libc surface: `malloc`, `memcpy`, `strlen`, `memset`, and several dozen additional functions
- Intercepts Chilkat function calls at the PLT boundary and resolves them with Python: AES via the `cryptography` library, RSA via native Python modular exponentiation (`pow(c, e, n)`)
- Executes all SIMD/NEON instructions natively within the emulator, preserving the exact behaviour of the proprietary key derivation

PLT-level interception of Chilkat calls also bypasses the library's licence-enforcement system without modifying the binary — the cryptographic operations are fulfilled by the Python implementations before the licence check is ever evaluated.

Performance Optimisation

The initial implementation required approximately 30 seconds per decode. Profiling identified a Unicorn instruction-level hook that fired on every emulated instruction — tens of millions of invocations — to maintain a counter and detect approximately 20 call sites. Two targeted changes reduced execution time to **approximately 3 seconds** (a 10× improvement):

- Removal of the global instruction-counting hook
- Restricting the tracing hook to the ~5 KB memory region occupied by `pc()`, rather than the entire address space

---

7. DECRYPTED OUTPUT FORMAT

The base64 string returned by `pc()` decodes to a binary structure with the following layout:

```
[text_length : 2 bytes, big-endian] [encoded_text : N bytes]

[image_length: 2 bytes, big-endian] [WebP image data ]
```

The text section uses a proprietary encoding based on the Spanish alphabet (including Ñ), with a dedicated separator byte acting as a field delimiter. Fields present include: credential type, CIC, OCR, CURP, full name, state, municipality, electoral section, expiry date, sex, version, generation date, and a verification signature.

The image section contains a **96×129-pixel WebP** encoding of the holder's photograph, compressed to approximately 1 KB.

---

8. SECURITY ANALYSIS

The INE system is architecturally oriented towards **offline verification**: all materials required for decryption and integrity verification are self-contained within the library and the QR codes themselves. No network connectivity is required at verification time.

The security model rests on several pillars:

- **Complexity obfuscation** — seven sequential layers create a high barrier to analysis
- **Native code** — critical logic resides in ARM64 machine code, not in the inspectable Java/Kotlin layer
- **Proprietary SIMD derivation** — key material cannot be recovered by static analysis alone; execution is required
- **ECDSA integrity verification** — tampered data will fail signature verification
- **8,192-bit RSA keys** — render brute-force attacks computationally prohibitive

The principal limitation of the model is that static AES keys and encrypted constants are embedded in the binary. Any party with access to the APK can extract the library. The practical security barrier is therefore the *complexity of analysis*, not the secrecy of the key material per se. This is a common trade-off in offline credential verification systems.

9. TOOLS AND ENVIRONMENT

`jadx`	APK decompilation (Java/Kotlin source recovery)
`Unicorn Engine v2`	ARM64 full-system emulation
`Capstone`	Disassembly and static instruction analysis
`LIEF`	ELF parsing and relocation resolution
`Python cryptography`	AES-CBC-256 and RSA operations
`zxing-cpp`	Binary QR code decoding
`rich`	Pipeline visualisation in terminal

10. CONCLUSION

What presents itself to the end user as a simple QR scan conceals a seven-layer cryptographic pipeline combining static and runtime-derived keys, 8,192-bit RSA operations, proprietary ARM64 SIMD-based key derivation, and ECDSA integrity verification. The design achieves robust offline verification at the cost of implementation complexity.

The ARM64 emulation approach proved effective: by executing the native library within Unicorn Engine and intercepting only the Chilkat and libc boundaries in Python, the proprietary SIMD layers were preserved without modification. The resulting decoder processes a credential image in under three seconds.

This research was conducted entirely independently, using only the publicly available APK. Cryptographic keys, memory offsets, and any personal data encountered during testing have been withheld from this publication.

Responsible Disclosure Notice. This paper describes the architecture of a publicly deployed system based solely on analysis of a freely downloadable Android application. No authentication mechanisms were circumvented, no personal data were accessed or retained, and no cryptographic secrets are disclosed herein. The author's intent is academic and educational.