# Decentralized System Report

Michele Frattini 4878744 - Luigi Timossi 4819664

## Contents

## 1 Introduction

In this project, we decided to develop a smart contract to implement NFTs and an application to interface with it.

## 2 InfUnigeCardsNFT

The NFTs we decided to create represent some students and professors from the 2019-2022 computer science course as a trading card game.

Initially, we considered using the ERC-721 standard for the implementation; however, the concept of individual NFTs evolved. In fact, the contract now includes multiple copies of each NFT. To support this feature, we decided to use the ERC-1155 standard, which is more suitable for our case.

### 2.1 Packs

The most interesting feature is that the mechanism for obtaining these NFTs is random. Users cannot choose in advance which NFTs to mint; everything is based on opening virtual packs.

Each pack, costing 0.05 Ether + fee, contains 5 random NFTs which are added to the user's collection. The collection consists of 80 cards, divided into students, professors, and a special card. The professors and the special card are rarer than the student ones. Each student card is associated with one of three categories: Defense, Attack, or Mind.

### 2.2 Metadata and Pinata

Each card obviously has its own metadata in JSON format and an image in PNG format. The JSON follows this pattern:

- `id`
- `name`
- `image`: containing the IPFS URL of the image
- `description`
- `attributes` containing:
  - `trait_type`: illustrator and type
  - `value`

As shown above, it is necessary to have the image link before deploying the JSON file.

We decided to use IPFS with Pinata as a pinning service. Pinning is necessary to make files easily retrievable on the IPFS network; otherwise, they would be hard to find.

We used the APIs provided by Pinata and a script to deploy the image folder, obtaining a folder CID; thus, the image with id 0 can be accessed via CID/0.png. We then updated the JSON metadata to insert the CID of the respective images and finally uploaded the metadata folder to IPFS as well, obtaining the final CID for the NFT BaseURI.

## 3 Smart Contract

The contract was developed in Solidity (version 0.8.29) and uses several libraries from the OpenZeppelin suite to ensure security, modularity, and upgradeability of the code. The use of the UUPS (Universal Upgradeable Proxy Standard) pattern allows the contract to be upgradable. Essentially, the proxy address (used for on-chain interactions) remains constant, while the contract logic can be updated by pointing to a new implementation.

In our case:

Proxy address: 0xA13d9d004ECC11740006dEFFc9C20Ee9cEa61353

Contract address: 0x87b466fb3F0f5d8D35FE1cdF1906d55F158680ab

Initially, the contract includes 300 available packs.

To implement the logic for determining the cards that come out of the packs, we used a dynamic bytes array of uint8 (batchOrder), which also optimizes space and thus reduces fees when deploying to the testnet.

This array is initially filled with IDs representing the card IDs from 0 to 80. As mentioned earlier, some cards are rarer (professors and the special card), and some are more common (students). To handle rarity, we decided to repeat each card ID in the array a certain number of times. For example, a professor's ID is repeated 10 times, while a student's ID is repeated 20 times.

During array initialization, IDs are repeated in sequence: first all IDs 0, then IDs 1, then IDs 2, and so on.

We then implemented the function _shuffleBatchOrder to shuffle these IDs. Essentially, a random number j is generated and used to swap positions. j is a number between i and batch lenght - 1, based on a pseudo-random hash derived from the block timestamp, block randomness (prevrandao), and the current index i.

We are aware that this choice may not be the most secure, as a validator could manipulate block.timestamp to influence the result. We also tried using the VRF (Verifiable Random Function) oracle to solve this issue. In the end, however, we decided to stick with the original solution since our use case doesn't involve sensitive data. Another reason we didn't use the oracle is that it is not natively compatible with the upgradable pattern, so we would have had to create another contract to interface with the proxy, which seemed unnecessarily complex for our use case.

The mint function, protected by the noReentrancy modifier, is designed so that users cannot open more packs if they have 250 or more NFTs from this series in their wallet, to avoid whales. The function also aggregates any duplicate cards found when opening packs, making it efficient for calling the _mintBatch() function from the 1155 standard.

The mint function cannot be called directly from outside; it is called by the two wrapper functions mint4me and mint4to, which are payable and include additional checks.

We also defined and handled possible errors including MaxCardsReached, NotEnoughCardsLeft, InsufficientETH, etc. Defining errors this way is another method to save on fees.

Using the SolidityScan tool integrated in Remix, we made some optimizations to reduce deployment costs. Examples include: adding an unchecked scope in for loops for the iteration variable update; creating temporary variables to reduce storage reads; and applying certain checks using assert().

We used Truffle to test the contract, which helped resolve some errors not caught by Remix. We then used Remix directly to deploy it to the testnet.

## 4 Site

To better interface with the smart contract, we developed a website that serves as the frontend. We used Nuxt, a framework based on Vue.js, along with the shadcn library for component styling. The site is responsive and adapts to smartphone format.

Both the website and the Pinata deployment were dockerized for simplicity and compatibility. All docker commands can be easily executed using the 'just' command. We used a '.env' file to securely store all keys and sensitive variables.

For deployment, we opted for GitHub Pages hosting on the repository's 'gh-pages' branch. This allowed us to create a GitHub Action that automatically builds and deploys the new version online with every site update.

`https://lurpigi.github.io/NFT-Unigecard/`

Right on the homepage, there is a brief project description and various hyperlinks to navigate to other pages.

To access the more interesting pages, users must connect their wallet. On the "My collection" page, users can view their NFT card collection in a grid, like a digital binder. Clicking on a specific NFT card redirects the user to the OpenSea page for that card (if it has already been minted). Here, users can buy and sell cards as well as view them.

On the ”Pack opening” page, users can expand their collection by opening packs for themselves or others.

To verify the contract on OpenSea, we needed to provide the ABI. We extracted it directly from Etherscan.

To implement the Web3 logic and call the functions from the contract’s proxy, we used the ethers.js library. For wallet login, we used defineStore() from the Pinia library, a state management library for Vue.js specifically designed to easily integrate with ethers.js. This allows users to connect with any BrowserProvider, not just Metamask.