DUNAÚJVÁROSI EGYETEM

MÉRNÖKINFORMATIKUS BSC

SZAKDOLGOZAT

SZTEGANOGRÁFIAI SZOFTVER FEJLESZTÉSE

Tibor Attila Gábor

mérnökinformatikus jelölt

A-037-INF-2022

Kivonat

Napjainkban fel sem tűnhet számunkra, hogy milyen kreatív módjai léteznek adataink álcázásának és elrejtésének. Sajnos az ilyen célokra elérhető szoftver eszközök viszont sok esetben idejétmúltak, vagy funkcionalitásaikat tekintve kezdetlegesek, sőt alkalmanként sebezhetőségekkel is rendelkeznek. Ezen dolgozat célja egy modern elvárásoknak megfelelő, egyszerűen használható és biztonságos adatrejtő alkalmazás készítése. A dolgozatban először feltárásra, és bizonyos szempontok alapján értékelésre kerülnek a jelenleg elérhető konkurens szoftverek. Ezt követően az olvasó betekintést nyerhet egy szoftverfejlesztési projektbe, amely az elkészült munka rövid értékelésével és a korábban vizsgált szoftverekkel való összehasonlításával zárul.

Abstract

Nowadays, we may not even realize what creative ways there are to disguise and hide our data. Unfortunately, many of the software tools available for this purpose are outdated or rudimentary in terms of functionality, and sometimes even have vulnerabilities. The aim of this thesis is to develop a modern, easy to use and secure data hiding application. The paper will first explore and evaluate, from a number of points of view, the competing software currently available. From there on, the reader will be given insight into a software development project, which will conclude with a brief evaluation of the work and a comparison with previously tested software.

Tartalomjegyzék

1. Bevezetés

Napjainkban egyre égetőbb probléma a magánélet kérdése. Mobil eszközeink szüntelenül adatot gyűjtenek rólunk, centralizált üzenetküldő alkalmazásokra hagyatkozva kommunikálunk, bízva abban, hogy privát beszélgetéseinket nem használják fel. Felhőszolgáltatások igénybevételével az interneten tároljuk bizalmas adatainkat remélve, hogy senki sem fér hozzá.

Hogyan kaphatnánk vissza egy keveset a magánéletünkből? Hogyan kommunikálhatnánk és tárolhatnánk bizalmas információt teljes biztonsággal? Az olvasó azt gondolhatná, hogy triviális megoldás erre a problémára a titkosítás. Bár ez a legtöbb esetben elégséges, egy bizonyos szinten már nem célravezető. A titkosításnak ugyanis van egy nagy problémája: magára vonja a figyelmet. A titkosított kommunikáció implikálja a bizalmas információ áramlást, és kényszeríthetnek az információ felfedésére. Egy olyan megoldásra van szükség, amely által tagadható a kommunikáció léte: szteganográfia.

1.1. Mi az a szteganográfia?

A szteganográfia titkos információ nem titkos hordozóba való rejtésének a tudománya. (Görög eredetű szó, a *'steganós'* és *'graphía'* szavak képzéséből, jelentése: *'rejtett írás'*.) Szemben a kriptográfiával, amely illetéktelen hozzáféréstől védi az információt, a szteganográfia célja tehát magának az információ létezésének az álcázása.

Az informatikán belül ez általában zajtoleráns fájlokban (pl.: kép-, hang-, videófájlokban) való adatrejtést jelent olyan módon, hogy az adott fájlban való változás nem észlelhető az emberi érzékszervek, de jobb esetben szteganalitikai[1] szoftverek számára sem. A szteganográfiai szoftverek ezt általában úgy érik el, hogy az adatrejtésből származó zajt kellően alacsony szinten tartják, így megőrizve a fájl közel eredeti minőségét. Fejlettebb algoritmusok figyelembe veszik a fájl statisztikai tulajdonságait, vagy akár az emberi érzékelés sajátosságait is, növelve az észlelhetetlenséget. (Az emberi szem például érzékenyebb a zöld fényre, így érdemes lehet egy képen a zöld színeket kevésbé megváltoztatni.) A legújabb kutatások már neurális hálózatokat is próbálnak alkalmazni az adatrejtési folyamat során.

[1] A szteganalízis a szteganográfia detektálásával foglalkozó tudományág.

1.2. Motiváció

Amikor először hallottam erről a tudományágról, szembetűnő volt számomra, hogy a legtöbb ingyenesen elérhető szteganográfiai szoftver idejétmúlt. Gyakran előfordult, hogy egy adott szoftver elavult titkosítási eljárásokat használt, vagy egyáltalán nem is támogatott semmilyen autentikációt. Számos szoftver olyan fájlformátumokkal dolgozott, amelyek hétköznapi felhasználásra nem relevánsak. Nem is beszélve arról, hogy ezen szoftverek nagy része megkérdőjelezhető forrásokból volt csak elérhető. Ezáltal úgy gondoltam, érdemes lehet időt fektetni egy saját szteganográfiai szoftver elkészítésébe, amely biztonságosabb, lehetőség szerint képes újat mutatni, és egyszerű a használata akár egy átlagfelhasználó számára is.

1.3. A dolgozat felépítése

A dolgozatban először megvizsgálom a létező szteganográfiai szoftverek hiányosságait egy egyszerű értékelési rendszeren keresztül, megindokolva ezzel, mi a létjogosultsága egy új szoftver fejlesztésének. Ezt követően próbát teszek a felvázolt probléma megoldására: az olvasó betekintést nyerhet egy számos szakterületet érintő szoftver fejlesztési projektbe, amely a dolgozat fő hangsúlya. Végezetül értékelem az elkészült szoftvert, és mérlegelem a projekt sikerességét.

2. Szteganográfiai szoftverek vizsgálata

A szoftver specifikáció meghatározása előtt célszerű volt megvizsgálni, milyen funkciókat ajánlanak az elérhető szteganográfiai szoftverek, milyen hiányosságaik vannak, és esetleg milyen új funkciókkal lenne érdemes egy új szoftvert felszerelni. A keresés során csak az ingyenes szoftverek lettek kipróbálva.

2.1. Az értékelési rendszer

Mivel nagyon sok szteganográfiai szoftver létezik, készítettem egy egyszerű értékelési rendszert, hogy ki lehessen szűrni azokat, amelyeket érdemes részletesebben is áttekinteni.

A szűréshez összegyűjtöttem azokat a főbb funkciókat, amelyeket elvárnék egy adatrejtő szoftvertől, majd fontosságuk alapján súlyoztam őket. Ezek az értékelési szempontok. Minden értékelési szemponthoz jár egy pontszám az alapján, hogy az adott szoftver hogyan teljesít. A végleges pontszám a szempontok súlyozott összegéből számítható az alábbi egyenlet szerint:

$$S = \sum_i w_i x_i$$

1. egyenlet: Egy szoftver pontszámának számítása

ahol: S – a szoftver végleges pontszáma
w – a szemponthoz meghatározott súly
x – a szemponthoz kapott pontszám

2.2. Az értékelési szempontok

Az értékelési szempontok az alábbiak, súly szerinti sorrendben. A szempontokhoz tartozó súly a név mellett szerepel zárójelben.

- **Hordozhatóság (1)**

  Szükséges-e telepíteni a szoftvert?
  0 – Igen.
  1 – Nem.

- **Titkosítás (1)**

  Támogatja-e a rejteni való adatok titkosítását?

  0 – Nem, vagy nem érhető el információ erről.

  1 – Igen, de nem ismert az algoritmus, vagy van legalább valamilyen autentikáció (pl.: jelszavas védelem).

  2 – Igen, de elavult algoritmust használ (pl.: RC4, 3DES).

  3 – Igen, modern algoritmust használ (pl.: Rijndael, ChaCha20).

- **Sztego-algoritmusok (1)**

  A használt szteganográfiai algoritmusok pontozására egy algoritmusok összehasonlításával foglalkozó konferenciakiadványt vettem alapul [1]. Ez a kiadvány 4 szempont alapján vizsgálja az algoritmusokat:

  - Robusztusság:
    Az elrejtett adatok ellenállósága a hordozó fájlban történt változtatások ellen (pl.: egy kép elforgatása, levágása).
  - Észlelhetetlenség:
    Annak a mértéke, mennyire nehéz egy embernek, vagy szoftvernek detektálni, hogy elrejtett adat van a fájlban.
  - Kapacitás:
    Mennyi adatot képes elrejteni az algoritmus.
  - Komplexitás:
    Mennyire bonyolult az algoritmus.

  A szoftver az alapján kapja a pontszámot, hogy az általa használt algoritmus(ok) a fentiek közül hány kritériumnak felel(nek) meg. A komplexitás csak implementáció szempontjából lényeges, ezért ezt nem vettem figyelembe. A pontozás tehát a következő:

  0 – Nincs információ arról, milyen algoritmust használ.

  1 – Egy kritériumnak megfelelően szolgáltat algoritmus(oka)t. (Pl.: nagy rejtési kapacitás.)

  2 – Két kritériumnak megfelelően szolgáltat algoritmus(oka)t. (Pl.: van egy algoritmus, amely nagy rejtési kapacitással rendelkezik, és van egy, amely nehezen észlelhető.)

3 – A célnak megfelelően lehet választani, minden kritériumhoz szolgáltat megfelelő algoritmust.

- **Adatintegritás ellenőrzése (1)**

  Képes-e ellenőrizni a rejtett adatok épségét?

  0 – Nem, vagy nincs információ.

  1 – Igen.

- **Releváns kimeneti fájl formátum (1)**

  Támogat-e széleskörűen használt, népszerű kimeneti fájl formátumokat? (Pl.: JPG, PNG, GIF, MP3, FLAC, MP4, MKV, PDF stb.)

  0 – Nem.

  1 – Igen, egyet.

  2 – Igen, többet is.

- **Fájl rejtés (1)**

  Képes-e fájlt rejteni?

  0 – Nem. (Pl.: csak beírt szöveget lehet rejteni.)

  1 – Igen, bármilyen rejteni való fájl kiválasztható.

- **Multiplatform (0.75)**

  Több számítógép platformon is futni képes?

  0 – Nem, vagy nincs információ.

  1 – Igen.

- **Bővíthetőség (0.5)**

  Bővíthető-e új funkciókkal, új rejtési algoritmusokkal?

  0 – Nem, vagy nincs információ.

  1 – Legalább részben nyílt forráskódú szoftver.

  2 – API[2]-val rendelkezik, plug-in (beépülő modul) alapú architektúra.

[2] Application Programming Interface

- **Konzolos interfész (0.5)**

  Használható-e parancssorból? Ez hasznos lehet, ha scriptekbe szeretnénk integrálni.

  0 – Nem

  1 – Igen

- **Tömörítés (0.5)**

  Képes-e tömöríteni a rejtendő adatokat?

  0 – Nem

  1 – Igen

- **Hibajavítás (0.25)**

  Alkalmaz-e valamilyen hibajavító algoritmust az adatkorrupció elleni védelem érdekében?

  0 – Nem

  1 – Igen

- **Láncolás (0.25)**

  Képes-e az adatokat feldarabolva, több hordozó fájlba rejteni?

  0 – Nem

  1 – Igen

- **UX[3] (0.25)**

  Mennyire felhasználóbarát, mennyire igényes a program? Mivel ezt teljesen szubjektíven értékeltem, ezért alacsony súlyt adtam neki.

  0 – Rossz – nehéz kezelni, hibás

  1 – Elégséges – csak parancssorból kezelhető.

  2 – Elfogadható – rendelkezik felhasználói felülettel

  3 – Jó – van felhasználói felület, és legalább valamennyire igényes, könnyen kezelhető.

A maximálisan elérhető pontszám így 15 volt.

[3] User experience, azaz felhasználói élmény.

2.3. Eredmények

A pontozás az 1. mellékletben található, összesen 33 szoftvert értékeltem. Az eredményből a következő észrevételeket tettem.

Egyetlen vizsgált szoftver sem támogatja a hibajavítást. Ez érdekes, ugyanis bár a kapacitás kárára, de növeli a rejtett adatok robusztusságát. Bizonyos titkosítási eljárások során egyetlen bit hiba is teljesen értelmezhetetlenné teheti az összes adatot, ezért érdemes lehet valamilyen hibajavító algoritmust alkalmazni. A rejtendő adat hibajavító kódolását a felhasználó külön szoftverrel sem tudja elvégezni, ugyanis ez még a titkosítás előtt történne meg, valamint az adatrejtő szoftver által beírt metainformációkat[4] is ugyanúgy védeni kell. Az egyetlen magyarázatom ezen funkció teljes hiányára az, hogy a fejlesztők feleslegesnek tarthatták abból adódóan, hogy egy fájl módosulása a legtöbb esetben egyébként is teljes adatvesztéshez vezetne a használt rejtési algoritmusok természetéből kifolyólag. Ez megmagyarázhatja az adatintegritás ellenőrzésének széleskörű hiányát is. Mégis, egyes felhasználási esetekben, például hosszabb távú fájl tárolásból származó korrupciók javítására hasznos lehet.

Néhány pozitívum is felfedezhető. Az alapvető kritériumok közül a hordozhatóságot, a titkosítást, és a fájl rejtést a legtöbb szoftvernek sikerült valamilyen szinten teljesítenie. Általánosságban az is elmondható, hogy a vizsgált szoftverek releváns kimeneti fájlformátumokat használnak. Az értékelés átlagos eredménye viszont 40% (19% szórással), és összesen csak 3 szoftver érte el a 70%-ot. Összeségében tehát az eredmények alapján kimondható, hogy a vizsgált szoftverek többsége alkalmatlan valódi használatra.

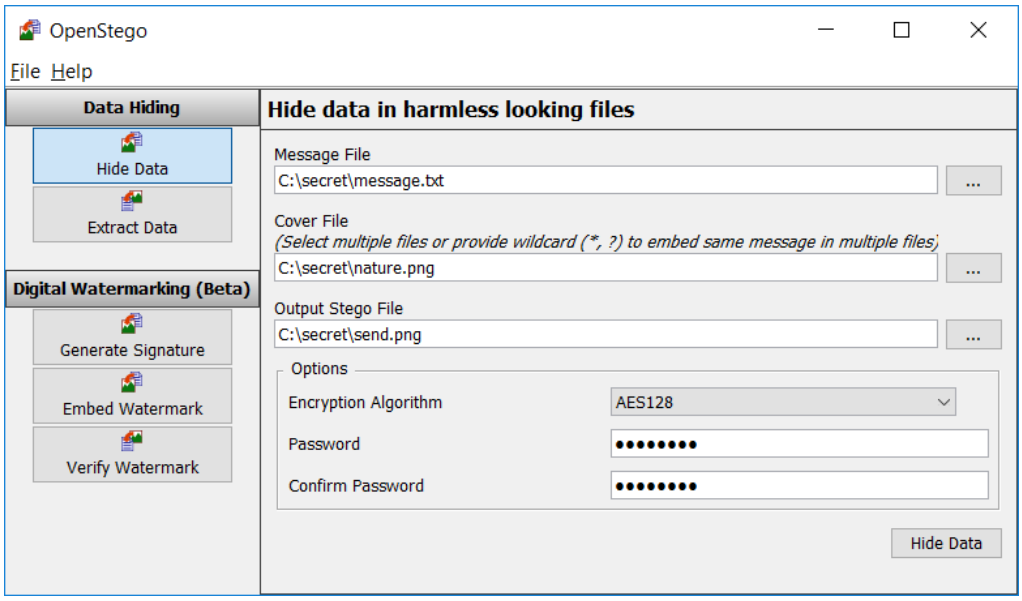
2.4. Konkurens szoftverek

A három legmagasabb eredményt elért szoftvert részletesebben is megvizsgáltam, ugyanis ezek már kifejezetten szofisztikált funkciókkal rendelkeznek.

[4] Információ az információról. Pl.: egy fájl formátuma, mérete, létrehozási dátuma.

2.4.1. OpenStego

Az OpenStego egy nyílt forráskódú, multiplatform szoftver, amely teljes mértékben Java nyelven íródott.



1. ábra: Az OpenStego felhasználói felülete [2]

Legfontosabb tulajdonsága, hogy moduláris felépítésű. A fejlesztő által szolgáltatott API dokumentáció segítségével bárki kiegészítheti egy adatrejtő, vagy titkosítási algoritmussal, és felhasználói felületet is szolgáltathat hozzá. Ez a képesség lehetővé teszi a program hosszú távú dinamikus fejlődését, miközben a kriptográfiai és adatrejtési algoritmusok elavulttá válhatnak. A felhasználói felülete (1. ábra) logikusan van felépítve, egyszerűen kezelhető. Itt észrevehető még, hogy a szoftver szolgáltat vízjelezési funkciót is. Ha scriptben szeretnénk használni a szoftvert, erre is van lehetőség, ugyanis konzolos felületről is használható. Kimeneti formátumként a PNG és BMP fájlformátumokat támogatja.

A program a Random-LSB adatrejtési módszert támogatja alapértelmezetten, amelyet bár nem feltétlen nehéz detektálni szoftverrel, de magas rejtési kapacitást tesz lehetővé. Mivel magam is ezt a technikát alkalmaztam, az algoritmusról a 3.3.1-es alfejezetben részletesebben is lesz szó, viszont a szoftver saját implementációjából kiemelném, hogy a véletlenszerű pixel kiválasztás a Java beépített `Random` osztályának felhasználásával történik, amelyet egy 8 bájt méretű `long` típusú értékkel inicializál [3, S. 86]. Ezt a

`long`-ot a jelszó MD5[5] hash értékének egy részletéből állítja elő [4, S. 58]. A probléma ezzel a megoldással az, hogy a `long` mérete nem kellően nagy, hogy kriptográfiai biztonságot nyújtson. Amennyiben egy támadó képes visszafejteni a random generátor helyes seed[6] értékét, azzal egyben a jelszó MD5 hash-ének egy részét is megkapja, amelyet felhasználva egy úgynevezett rainbow táblás támadással akár a jelszót is képes lehet visszafejteni, ezzel teljesen feltörve a rejtett adatokat.

Tovább vizsgálva a forráskódot észrevehetjük, hogy titkosítási eljárásként (a DES[7]-t figyelmen kívül hagyva) az AES[8]-t használja a program, 128, vagy 256 bites kulcsmérettel [5, S. 58]. Viszont egy program biztonsága nem csak attól függ, hogy milyen titkosítási eljárásokat használ, hanem attól is, hogy hogyan használja azokat. A program jelenleg az RFC 8018-ban definiált PBES2[9] titkosítási módszert alkalmazza [5, S. 58], [6, Szak. Cipher Algorithm Names], amely önmagában nem határoz meg adat hitelesítési eljárást [7, o. 16]. (A forráskódban, az algoritmus nevében látható `HmacSHA256` a kulcsszámító algoritmusban használt hash függvényre utal.) További probléma, hogy csak 7 iterációval alkalmazza a kulcsszámító algoritmust. (Az RFC 8018 a PBES2-höz a PBKDF2[10] kulcsszámító algoritmust specifikálja [7, o. 15].) Mivel a PBKDF2-vel SHA-256[11] hash függvényt használ, az alacsony iteráció szám jelentősen megkönnyít egy potenciális brute-force támadást a jelszó ellen. Egy modern videókártya ugyanis több millió hash-t képes kiszámítani másodpercenként. Az OWASP[12] ajánlása szerint a program által használt konfigurációhoz 310 000 iterációt kellene használni [8].

Az OpenStego tehát annak ellenére, hogy az értékelési rendszerben számos kritériumot teljesít, jelentős információbiztonsági problémákkal rendelkezik, amelyekre a saját szoftverem tervezésénél ügyelnem kellett.

[5] Message-Digest algorithm 5
[6] Az az érték, amellyel a pszeudo-random generátort inicializáljuk.
[7] Data Encryption Standard
[8] Advanced Encryption Standard
[9] Password-Based Encryption Scheme 2
[10] Password-Based Key Derivation Function 2
[11] A Secure Hash Algorithm 2 család 256 bites kimenetű verziója.
[12] Open Web Application Security Project

2.4.2. Steghide

A Steghide egy konzolos alkalmazás, többnyire C++ nyelven íródott. Nyílt forráskódú, és multiplatform, akár csak az OpenStego. Viszont nem csak kép (JPEG, BMP), hanem már hangfájlokat (WAV, AU) is támogat. Felhasználói felület hiányában nem nevezhető felhasználóbarátnak, bár létezik régebbi harmadik fél által készített felhasználói felületes verzió. Érdemes kiemelni, hogy teljes adatkinyerés helyett képes csak metainformációkat megjeleníteni egy adott stego-fájlban[13] rejtett adatokról. Ha sok hordozófájlunk van, ez leegyszerűsíti a fájlkeresés folyamatát.

Számos kriptográfiai algoritmust támogat, amelyek között még napjainkban is használatos algoritmusok is felfedezhetőek. A forráskódot áttekintve láthatjuk, hogy a Steghide, hasonlóan az OpenStego-hoz, nem alkalmaz semmiféle adat hitelesítést a titkosítás visszafejtése előtt [9, S. 165], ezzel lehetővé téve egyes aktív támadásokat a rejtjelezett adatokon (például a CBC[14] blokktitkosítási mód IV[15]-jának manipulációjával). Kulcsszámításhoz egyszerű MD5 hash-t használ 0 iterációval, amely ugyanazokat a problémákat vonja maga után, amelyeket az OpenStego esetében már tárgyaltunk.

Sajátossága, hogy egy gráfelméleten alapuló hibrid szteganográfiai algoritmust alkalmaz, amely nem csak nehezen észlelhető, de a dokumentáció szerint ellenállóvá teszi egyes statisztikai támadások ellen is [10]. Az algoritmussal kapcsolatban sajnos újra párhuzam vonható az OpenStego-val: Egy jelszóból számított MD5 hash alapján inicializált, 32 bit méretű seed-del ellátott pszeudo-random forrást használ [11, S. 26], de legalább ebben az esetben a seed kiszámításának folyamatából adódóan az eredeti MD5 hash nem nyerhető vissza.

A Steghide összességében egy kifejezetten fejlett szoftver, felhasználói felület hiányában viszont nem praktikus az átlagfelhasználó számára.

[13] Rejtett adatot tartalmazó fájl
[14] Cipher Block Chaining
[15] Initialization Vector

2.4.3. OpenPuff

A vizsgált szoftverek közül az OpenPuff a legfejlettebb.

2. ábra: Az OpenPuff felhasználói felülete [12]

A felhasználói felületén látható (2. ábra), hogy nem csak szteganográfiai, hanem vízjelezési funkcióval is rendelkezik, és ezeket a feladatokat több szálon futtatva képes elvégezni. Az adatok tömörítését nem támogatja, viszont az értékelt szoftverek közül egy a háromból, amely támogatja a hordozófájlok láncolását, azaz képes a bemeneti adatot feltördelve, darabonként külön hordozófájlba rejteni. Ez egy nagyon hasznos funkció lenne, a megvalósítás miatt viszont problémás a használata. A program ugyanis elvárja a felhasználótól, hogy megjegyezze, és a visszafejtés során megadja megfelelő sorrendben pontosan azokat a fájlokat, amelyek tördelt adatot tartalmaznak, ahelyett, hogy például csak egy mappát kellene megadnia, és a program keresné meg a saját fájljait a megadott jelszó alapján.

Összesen 17 kimeneti fájlformátumot támogat, ezek közül 6 nevezhető napjainkban is relevánsnak:

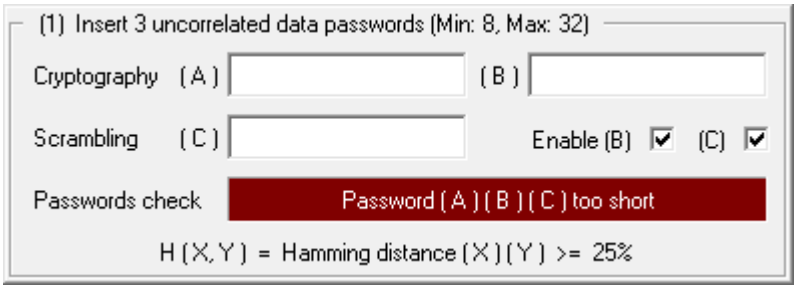
- Kép formátumok: JPG, PNG
- Hang formátumok: MP3, WAV
- Videó formátumok: MP4
- Dokumentum formátumok: PDF

A felsorolt formátumok közül a kép formátumokat kivéve gyakran csak speciális eseteket kezel megfelelően. Esetemben borítóképpel rendelkező MP3 fájlokat, egyes

WAV és PDF fájlokat, és egyetlen MP4 fájlt sem volt képes betölteni, ez kifejezetten kellemetlenné teszi a használatát. Általánosságban a rejtési kapacitása még nagy méretű fájlok esetén is alacsony, a nagyobb felbontású képek feldolgozása pedig még egy korszerű számítógépen is lassú.

Ellentétben a korábban vizsgált szoftverekkel, nem teljesen nyílt forráskódú, viszont a kriptográfiai könyvtár, amelyre alapul (libObfuscate), nyilvánosan elérhető. A dokumentációból kiderül, hogy az adatrejtési folyamat során számos titkosítási algoritmust véletlenszerűen keverve rejtjelezi az adatot [13]. Blokktitkosítási eljárásként CBC-t használ, és nincs a dokumentációban arra utaló jel, hogy az adatok visszafejtése előtt ellenőrizné azok hitelességét. Összességében az OpenPuff titkosítási folyamata feleslegesen bonyolult, és a CBC eljárásból eredően csak visszafejtés során párhuzamosítható, amely a program sebességét rontja.

Vitatható, hogy három jelszó többszörös titkosítással ötvözve számottevően biztonságosabb-e, mint egyetlen jól megválasztott jelszó egy erős kulcsszámító metódussal és titkosítási eljárással kombinálva, viszont a program erre is ad lehetőséget: opcionálisan kettő jelszó választható a titkosításhoz, és egy az obfuszkációhoz, ahogy alább látható.



3. ábra: Az OpenPuff jelszó beviteli ablaka (saját képernyőkép)

Az OpenPuff tehát az ajánlott funkcióit tekintve nem teljesen kiforrott, és a felhasználói felületén megjelennek olyan fogalmak (pl.: szál, Hamming-távolság) amelyet az átlag felhasználó nem fog érteni. Ezutóbbi viszont nem róható fel a programnak, ugyanis a weboldalán professzionális szoftverként van hirdetve [12].

3. Tervezés

Ebben a fejezetben a szoftverprojektem előkészítési fázisát mutatom be.

3.1. A szoftverspecifikáció

Mint azt az előző fejezetben tárgyaltuk, a konkurens szoftverek közül a legjobbak is rendelkeznek kisebb-nagyobb problémákkal, amelyek elbátortalaníthatnak a valódi használattól. A célom tehát egy olyan szoftver létrehozása volt, amely az említett problémákat kiküszöböli, és esetleg képes is valamilyen új funkciót mutatni.

3.1.1. Követelmények

Tapasztalataim alapján a következő elvárásokat állítottam fel a saját projektemmel szemben. Az elvárások priorizálását a MoSCoW[16] módszer segítségével végeztem el [14, o. 8].

- **Szükséges**
  - Az alkalmazott szteganográfiai algoritmus(ok) a 2.2-es alfejezetben tárgyaltak szerint legalább egy kritériumnak feleljen(ek) meg.
  - Releváns fájlformátumokkal dolgozzon.
  - Rendelkeznie kell felhasználói felülettel, és a haladó felhasználók számára parancssorból is használható kell, hogy legyen.
  - Az adatrejtést és adat-visszafejtést megszakítható módon kell végezze, és a folyamat állapotát jeleznie kell a felhasználó számára.
  - Alkalmazzon korszerű, biztonságos kriptográfiai eljárásokat, és a kriptográfiai titkokat biztonságosan kezelje a memóriában.
  - Képes legyen ellenőrizni a rejtett adatok épségét, és hitelességét.
  - Jelezze a felhasználó számára a hordozófájl(ok) adatrejtési kapacitását.
  - Legyen hordozható: telepítés és függőségek nélkül működjön, legyen kis méretű.
  - Tömörítse a rejtendő adatokat.

[16] Szükséges (Must have), fontos (Should have), lehetséges (Could have), nem valósul meg (Won't have).

- **Fontos**
  - Legyen kriptográfiailag agilis[17].
  - Legyen bővíthető új szteganográfiai algoritmusokkal.
  - A felhasználói felület igényes, átlátható és reszponzív legyen.
  - Legyen multiplatform.
- **Lehetséges**
  - Konzolos használat esetén kezelje a standard bemenet és standard kimenet átirányítását.
  - Alkalmazzon hibajavító kódolást.
  - Jelezze a felhasználó számára az elkészült hordozófájlban történt minőségi változásokat.
  - Több szteganográfiai algoritmust is támogasson.
  - Képes legyen több nyelv támogatására (lokalizáció).
  - Képes legyen a rejtési folyamat után az eredeti fájlokat véglegesen, nyomtalanul törölni a háttértárolóról.
  - Képes legyen metainformációkat megjeleníteni a stego-fájlban tárolt adatokról anélkül, hogy vissza kellene fejteni az összes rejtett adatot.
- **Nem valósul meg**
  - Képes legyen az adatot feldarabolva, több hordozófájlt láncolva elrejteni.
  - Plug-in alapú architektúrával rendelkezzen.

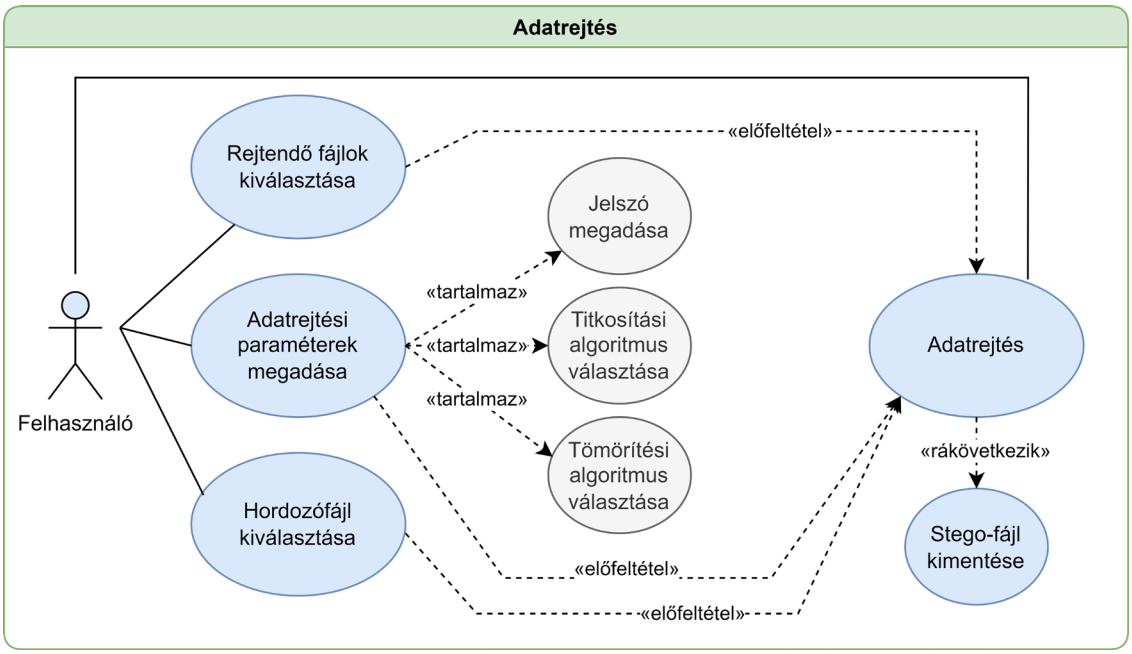
Ahogy fent látható, az időszükségletet is figyelembe vettem az elvárások prioritásának meghatározása során. A hordozófájlok láncolása például egy olyan funkció, amely a felhasználói felülettől elkezdve a szteganográfiai motorig[18] számos részét érinti a programnak, és hosszas tervezést, implementációt, tesztelést igényel. Az előbbiek elmondhatóak a program plug-in alapú felépítéssel való megtervezésére is. Egy ilyen felépítésű program elkészítése kifejezetten kockázatos, még ha csak az API tervezésére fordított időt tekintjük is, és előreláthatatlan problémákkal jár az implementáció során, amelyet, ha társítunk az alapfeladat kihívásaival, nem tekinthető reális elvárásnak.

[17] Új kriptográfiai algoritmusok adaptálásának a képessége, a régi algoritmusok kompatibilitásának megőrzése mellett.

[18] A szteganográfiai motor végzi az adatrejtés, és adat-visszafejtés feladatát.

3.1.2. Használati esetek

A részletesebb tervezés előtt tekintsük át, hogyan képzeltem el az elkészítendő program használatát.

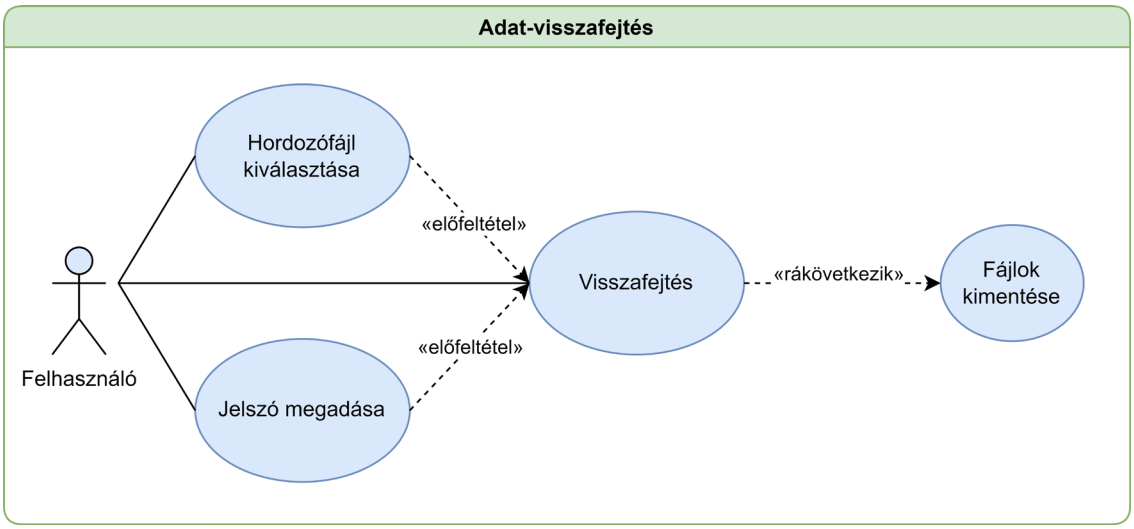


4. ábra: Használati eset diagram az adatrejtés folyamatáról (saját szerkesztés)

Ahhoz, hogy az adatrejtési folyamatot elindíthassa a felhasználó, csak a rejtendő fájlokat, a hordozófájlt, és a paramétereket kell megadnia. ahogy a 4. ábra szemlélteti. Ezt követően az adatrejtés megkezdhető. A program a hordozófájl formátumának megfelelően automatikusan választja ki az adatrejtési algoritmust. Ennek következtében természetesen fájlformátumonként csak egy algoritmust támogathat, másképpen a felhasználónak kellene megjegyeznie, hogy egy adott fájlhoz milyen algoritmust használt. Az adatrejtési folyamat végeztével bekéri a felhasználótól az elkészült fájl mentési helyét, és exportálja azt.

A rejtési paraméterek közül a titkosítási és tömörítési algoritmusok választását célszerű legördülő menüvel megvalósítani, mivel adottak. Ezen vezérlők alapértelmezetten elrejthetőek a felhasználói felületen, ugyanis valószínűleg csak a haladó felhasználók fogják használni őket. A rejtendő fájlok, és a hordozófájl kiválasztása egyszerű fájldialógussal, és ’drag and drop’ módszerrel is véghezvihető.

Az elkészült stego-fájlba a rejtendő adatokon kívül természetesen belekerülnek a rejtési paraméterek is, ezáltal a visszafejtési folyamat előkészítése még egyszerűbb, ahogy az 5. ábra szemlélteti alább.



5. ábra: Használati eset diagram az adat-visszafejtés folyamatáról (saját szerkesztés)

Az adatok kinyeréséhez a felhasználónak csak a rejtett adatokat tartalmazó fájlt kell megadnia, a hozzá tartozó jelszóval. Az adatrejtési folyamathoz hasonlóan a program végzi a megfelelő algoritmus kiválasztását, és az adatok kinyerése után kell csak a felhasználónak kiválasztani azok mentési helyét.

Fontos, hogy az adott folyamat állapotát visszajelezzük a felhasználónak, ez a legegyszerűbben egy folyamatjelzősáv alkalmazásával tehető meg. A hibák visszajelzését számos módon meg lehet valósítani. Mivel a felugró ablakok zavaróak lehetnek, alkalmazható például egy log panel. Ennek az az előnye, hogy nem csak a hibákat, hanem akár a folyamat állapotát is részletesen tudathatjuk a felhasználóval, viszont nem szép. Alternatívaként használható valamilyen státusz felirat, amely a program állapotának megfelelően változik.

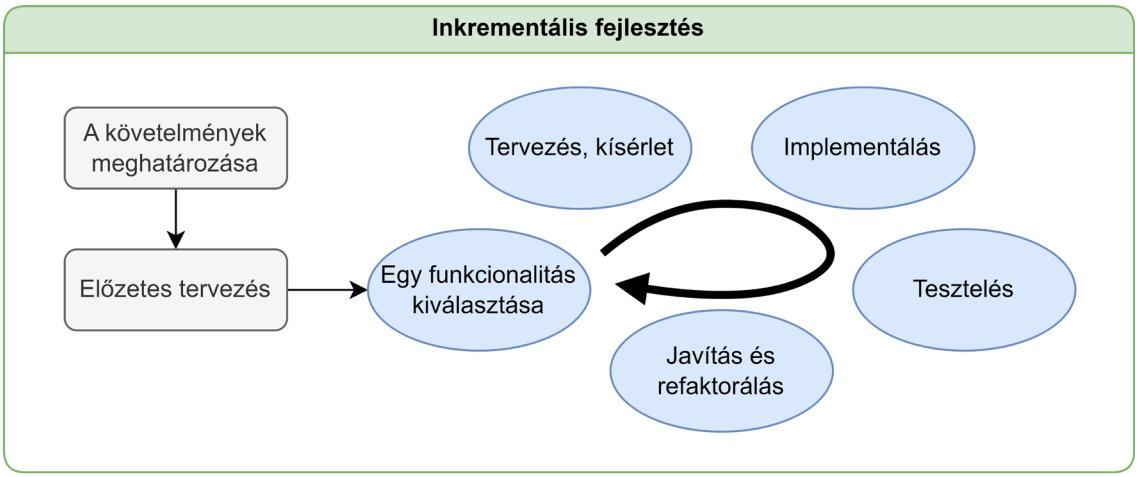
Egy ilyen program felhasználói felülete kevés mezőt igényel, és amennyiben jól van megvalósítva, könnyen kezelhető. A parancssoros használat csak annyiban változik, hogy a felhasználónak előre meg kell adnia a bemeneti és kimeneti paramétereket a program elindítása előtt.

3.2. Alkalmazott technológiák

Ebben az alfejezetben röviden, általánosan ismertetem a szoftverprojekt során használt legfontosabb technológiákat és módszereket.

3.2.1. A fejlesztési módszer

Ahogy a dolgozat felépítése is tükrözi, a feladat megoldása leginkább egy inkrementális szoftverfejlesztési modellhez hasonlítható, amelyet a 6. ábra szemléltet.

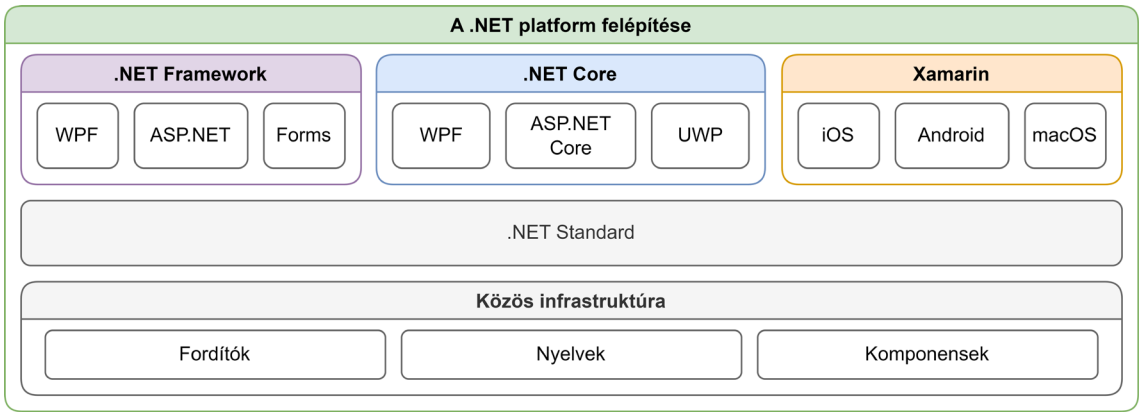


6. ábra: Az alkalmazott fejlesztési módszer (saját szerkesztés)

Egy előzetes tervezési fázis után a projekt lentről (azaz a szteganográfiai motortól indulva) felfelé valósult meg. Az egyes funkcionalitások implementálását további tervezés és prototípus kód készítése előzte meg, majd az implementálás után írtam meg a unit teszteket. Amennyiben szükséges volt, javítottam az implementáción, majd jöhetett a következő funkcionalitás. Mivel minden önálló funkció azonnal tesztelésre került, az integrációs tesztek problémamentesen zajlottak, és a legutolsó funkció implementálásával már működött is a program.

3.2.2. A .NET platform

Projektem alapjaként a Microsoft nyíltforráskódú szoftverfejlesztési platformját, a .NET-et céloztam meg. Magába foglal számos programozási nyelvet, egyebek mellett a C#-pot is, amelyet az egyetemi évek alatt alkalmam volt elsajátítani.



7. ábra: A .NET platform felépítése (saját szerkesztés [15, Ábr. 1] alapján)

Ahogy fent látható (7. ábra), több operációs rendszeren is támogatott, rugalmasságának köszönhetően alkalmazható különféle problémák megoldására a webalkalmazás fejlesztéstől a játékfejlesztésig. Alapja a .NET Standard API, amelynek három fő implementációja létezik: a .NET Framework, a .NET Core, és a Xamarin. Hatalmas közösségi támogatást élvez, ezáltal számos ingyenesen elérhető függvénykönyvtárt, és programcsomagot használhatunk a saját szoftverünk fejlesztésénél, a NuGet csomagkezelő segítségével. Ezzel nem utolsósorban gyors alkalmazásfejlesztést tesz lehetővé. Összességében tehát megfelelő választás volt a projektem megvalósításához.

A programom felhasználói felületének elkészítéséhez a WPF[19] platformot választottam (.NET Core alapon), mivel már kiforrott technológia, széleskörűen testre szabható és sok internetes forrás is elérhető hozzá. Ha viszont az egész projekt WPF-ben valósult volna meg, azzal nem tett volna eleget a multiplatform követelménynek, mivel a WPF csakis Windowson elérhető. Ezért (egyéb indokok mellet) a kódbázist elkülönítettem, és újrahasznosítható módon, .NET Standardra terveztem. Ezzel kapcsolatosan a projektem pontos felépítéséről a 4. fejezetben lesz szó.

[19] Windows Presentation Foundation

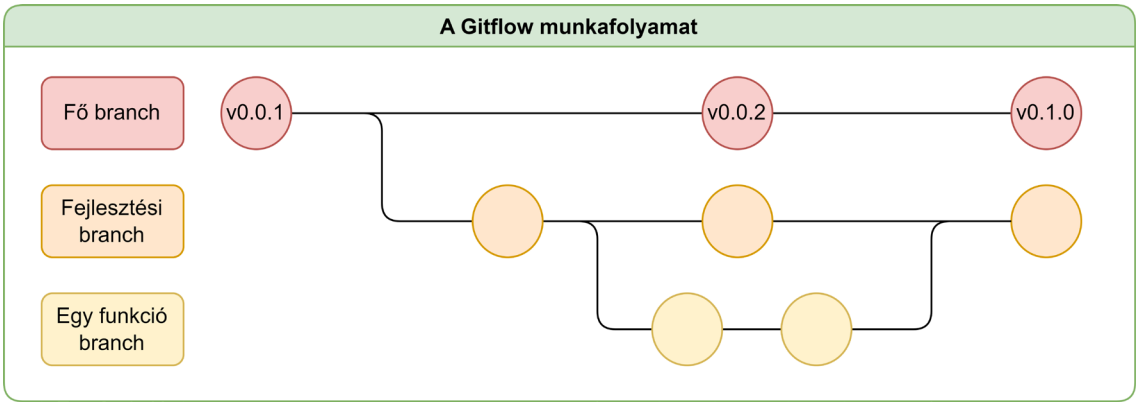
3.2.3. A fejlesztési környezet

Ha .NET alkalmazást fejlesztünk, értelemszerű, hogy a Visual Studio-t választjuk fejlesztői környezetként. A Visual Studio szükségleteinknek megfelelően konfigurálható, és számtalan beépített eszközzel rendelkezik, amellyel a programozást segíti. Tartalmaz integrált tesztelési környezetet, legújabb verzióiban már mesterséges intelligencia alapú kódkiegészítéssel is rendelkezik. A képességek listája szinte végtelen, és folyamatosan bővül.

3.2.4. Forráskód kezelés

Egy ilyen volumenű szoftverprojekt megvalósítása lehetetlen megfelelő forráskód kezelés nélkül. A módosítások visszakövethetősége, és a biztonsági mentés kritikus szükségletek. A projektem során a Git verziókezelő rendszert használtam, és a következő konvenciókat alkalmaztam.

A Conventional Commits egy ember és számítógép számára is olvasható commit konvenció. Az alapja, hogy kulcsszavakat használ minden commit üzenet elején egy meghatározott formátum szerint, ezzel segítve a gyors áttekinthetőséget, kereshetőséget, és a szoftver kiadása során automatikusan generálhatóvá teszi a változások jegyzékét. Ilyen kulcsszó lehet például új funkció implementálása esetén a `feat` vagy egy hiba javítása esetén a `fix`. Az adott kulcsszóhoz megadhatunk zárójelben egy tartományt is, ezzel behatárolhatóvá válik, hogy a projekt mely részérve vonatkozik az adott commit [16]. (Egy példa a sajátjaim közül: `fix (WPF): theme system`)



8. ábra: A Gitflow munkafolyamat (saját szerkesztés [17, Ábr. 2] alapján)

Alkalmaztam továbbá egy branch kezelési konvenciót is, a Gitflow-t, így elkülönítve a projekt különböző szegmenseit. Ahogy a 8. ábra szemlélteti, a Gitflow lényege,

hogy a projekt feloszlik egy fő, és egy fejlesztési branch-re. A különböző funkciók külön-külön branch-en kerülnek implementálásra, majd, amikor elkészülnek, integrálódnak a fejlesztési branch-be. A fő branch gyakorlatilag csak a fejlesztési branch pillanatképeiből áll. [17]

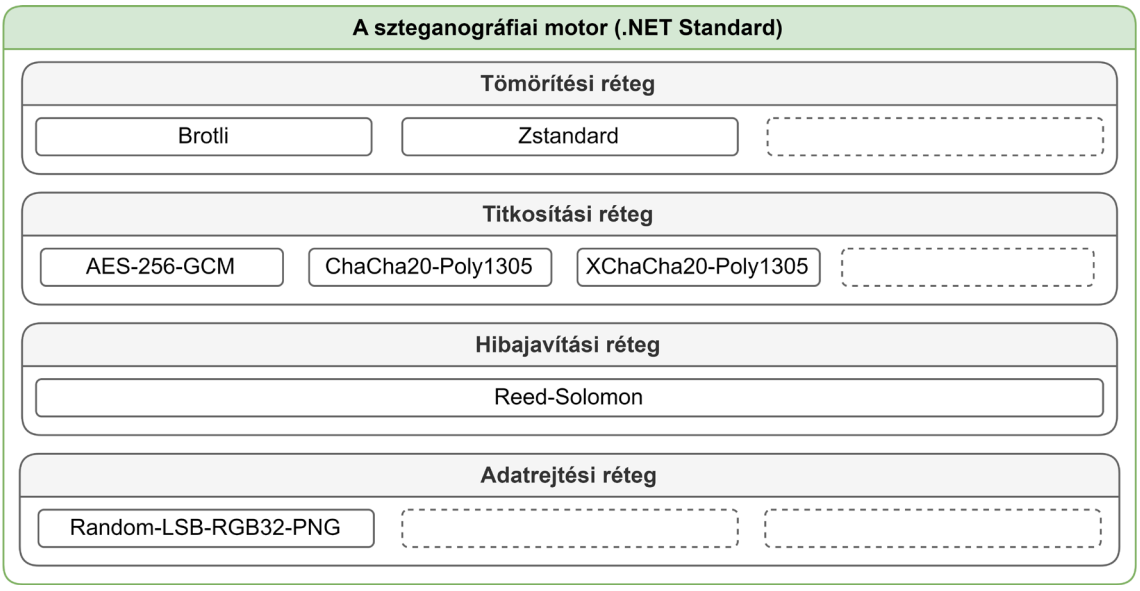
A forráskód rendszerezése mellett fontos a külalak is. A megfelelő kódolási stílus kikényszerítése érdekében létrehoztam egy szerkesztő konfigurációt a Visual Studio-hoz, valamint alkalmaztam egy automatikus kód elemzőt, amely ellenőrizte a forráskód formai helyességét az említett konfiguráció ellenében.

3.2.5. Dokumentáció generálás

Külön szoftverdokumentáció készítése helyett sokkal célszerűbb volt magát a forráskódot dokumentálni, és a forráskód alapján automatikusan generálni a dokumentációt. Ehhez a Doxygen-t alkalmaztam, amely egy népszerű, és könnyen konfigurálható dokumentáció generáló szoftver. A Doxygen kikeresi a forráskódban található kommenteket, rendszerezi, és összegyűjti az osztályokat, majd egy átlátható, diagramokkal kiegészített dokumentációt készít a kívánt kimeneti formátumban.

3.3. Alkalmazott algoritmusok

A feladat során a legtöbb tervezést a szteganográfiai motor igényelte, különböző szakterületeket érintő algoritmusokat kellett alkalmaznom. Ezen algoritmusok kiválasztásáról, és a paraméterezésük tervezéséről lesz ebben az alfejezetben szó.



9. ábra: A szteganográfiai motor rétegei (saját szerkesztés)

A szteganográfiai motort érdemes volt elkülönített rétegekre bontani az adatrejtési folyamat lépései szerint, ezzel javítva a tesztelhetőséget is. Az elképzelt felépítést a 9. ábra szemlélteti, és a továbbiakban a rétegekben használt algoritmusokat lentről felfelé haladva ismertetem. A motor pontos működése viszont még nem lesz tárgyalva.

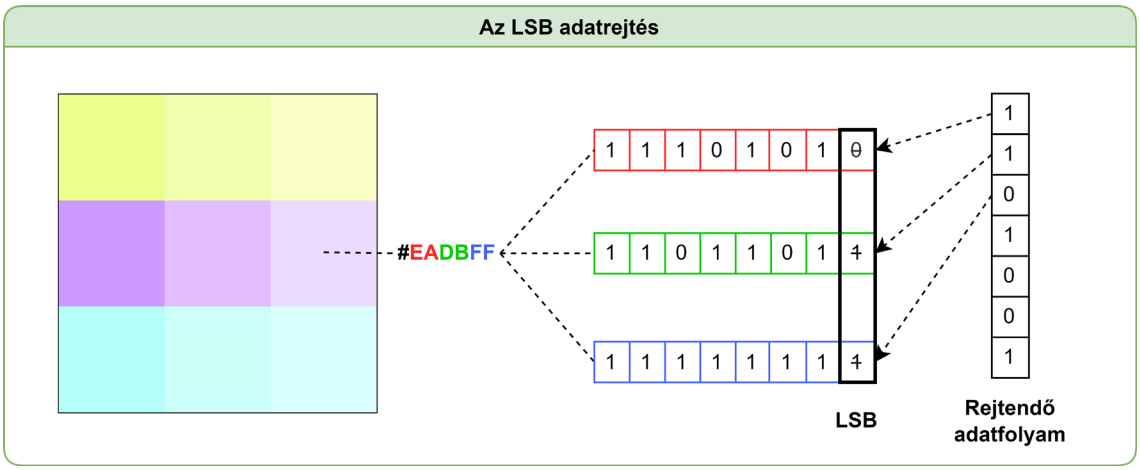
3.3.1. A Random-LSB adatrejtési algoritmus

A szoftver specifikáció alapján olyan szteganográfiai algoritmust kellett választani, amely vagy nagy adatrejtési kapacitással rendelkezik, vagy nehezen észlelhető, (általában ez a két tulajdonság kizárja egymást,) és lehetőség szerint robusztus is, azaz ellenáll a fájlban történt módosításoknak. Mivel a specifikáció magas prioritással meghatározta, hogy a szoftver szükség szerint bővíthető kell, hogy legyen új szteganográfiai algoritmusokkal, a program első verziójának elkészítéséhez elegendő volt egy egyszerűbb algoritmus is, mintha csak egy referencia implementáció lenne.

Az LSB[20] adatrejtési módszer szinte bármilyen mintavételezett, tér tartományban lévő, zajtoleráns digitális jel esetében alkalmazható. Ilyen jel lehet például egy PCM[21] formátumban enkódolt hang, vagy egy sRGB színtartományban ábrázolt kép. Mivel a PNG képformátumot céloztam meg, mint hordozófájl típust, így a módszer elvét ezen keresztül fogom ismertetni.

A PNG egy veszteségmentesen tömörített, raszteres képfájl formátum. Egy átlagos, színes PNG fájl 3 csatornán (az R, G és B, azaz a vörös, zöld és kék csatornákon) tárolja az alap színek intenzitását, amelyek additív keveréséből áll össze a pixelek színe. (Lehet még egy opcionális alfa csatorna is, amely az opacitást reprezentálja.) Ahogy a 10. ábra szemlélteti, az egyes csatornák egy 8 bites számként tárolják a csatornához tartozó szín intenzitását, azaz csatornánként 2^8, pixelenként pedig $2^{8 \cdot 3} \approx 16\ millió$ színt lehetséges megkülönböztetni. Ha van 16 millió színünk, két szín között lehet olyan kicsi a különbség, hogy egyes kijelzők nem is képesek megjeleníteni, egy ember pedig végképp nem tudja a szemével megállapítani. Ez a redundancia felhasználható adatrejtésre.

[20] Least Significant Bit
[21] Pulse-code modulation



10. ábra: Az LSB adatrejtés sRGB képek esetén (saját szerkesztés [18, Ábr. 2], [19, o. 242] alapján)

Ha az egyes csatornák utolsó két legkisebb helyiértékű bitjét (azaz utolsó két LSB-jét) (10. ábra), felülírjuk a rejteni kívánt adattal, egy sRGB kép esetén pixelenként 6 bitet tudunk tárolni. (Ez egy 1920x1080 felbontású kép esetében már 1,56 megabájtot jelent!) A tárolt adat viszont a képen zaj formájában fog megjelenni. Az asszociált maximális zajterhelés általánosan az alábbi egyenlet szerint számítható.

$$Q_z[dB] = (S - K) \cdot 20 \log 2$$

2. egyenlet: A maximális zajterhelés számítása LSB módszer esetén [19, o. 246]

ahol: Q_z – a maximális zajterhelés decibelben
S – a felülírandó bitek száma mintánként
K – a minta bitmélysége

Mivel a 8 bites csatornákból 2 bitet használunk fel adatrejtésre, a képet a legroszszabb esetben is csak $Q_z = (2 - 8) \cdot 20 \log 2 \approx -36dB$ zaj fogja terhelni, amely jóval az észlelhetőség határán belül van. [19, o. 246]

Az LSB módszer ötvözhető fejlettebb technikákkal is. Készíthető például olyan algoritmus, amely kifejezetten a kép zajosabb részeit megkeresve rejti el az adatokat, ezáltal tovább növelve az észlelhetetlenséget. Meglehet próbálni a rejtendő adatokat összeilleszteni az LSB értékekkel olyan módon, hogy minél kevesebbet kelljen megváltoztatni. A legegyszerűbb dolog, amit tehetünk az az, hogy a kép pixeleit szekvenciálisan végig iterálva írjuk be az adatokat. Viszont kifejezetten feltűnő, hogyha a kép egyik fele sokkal zajosabb, mint a másik, és ez szoftverrel is egyszerűen kimutatható. Egy alternatíva lehet,

ha pszeudo-véletlenszerűen választjuk ki a pixeleket, innen ered a Random-LSB elnevezés. Ennek a módszernek megvan az az előnye is, hogy a véletlenszám generátort inicializálhatjuk valamilyen kulcs segítségével, ezzel további biztonságot adva adataink számára, mintha egy transzpozíciós rejtjelező eljárás lenne.

A Random-LSB módszert tehát nem triviális detektálni amennyiben kevés adatot rejtünk el, viszont mégis magas adatrejtési kapacitással rendelkezik amennyiben szükséges, és nem is túl bonyolult implementálni. Az LSB módszer nagy hátránya általánosan viszont, hogy nem túl robusztus. Ha levágjuk a képet, elmossuk, rárajzolunk, elforgatjuk, szinte bármilyen módosítást végzünk, az információvesztéssel jár. További negatívum, hogy csak tömörítetlen, vagy veszteségmentesen tömörített formátumoknál alkalmazható, ráadásul a hozzáadott zaj csökkenti a tömörítés hatékonyságát, így az eredetinél nagyobb fájlokat eredményez. Ezektől a hátrányoktól eltekintve viszont megfelelő választásnak tartottam a szoftverem megvalósításához.

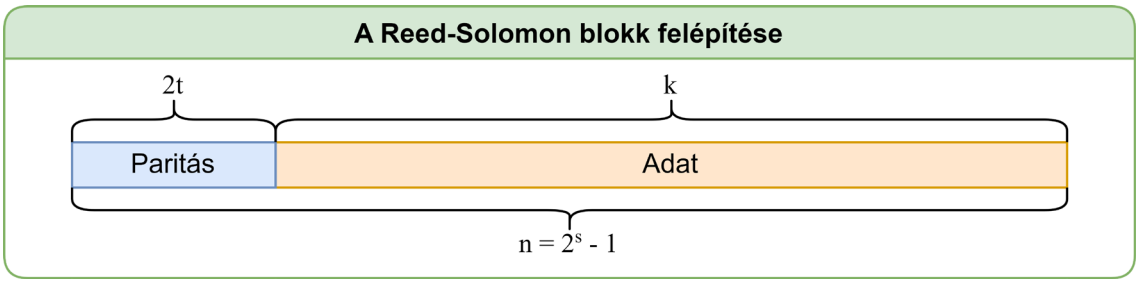
3.3.2. A Reed-Solomon hibajavító kódolás

Mivel egyetlen vizsgált szoftver sem alkalmazott hibajavító eljárásokat, itt volt egy lehetőség innoválni. A hibajavítás hasznossága legfőképp az alkalmazott szteganográfiai módszertől függ, viszont általánosságban véve megelőzheti a hosszú távú tárolásból eredő adat degradációt. A Random-LSB módszer esetében a hibajavítás például képes lehet helyrehozni a képre való rajzolásból eredő hibákat. (Ráadásul a Random-LSB módszernek az adatok véletlenszerű szétszórásából eredően van egy interleaving szerű hatása is, amely segít a burst-hibák javításában.) Összességében tehát a hibajavításra a szteganográfia kapcsán inkább csak egy extra képességként lehet tekinteni, amely kisegíthet speciális esetekben, ennek következtében nem fordítottam kifejezetten nagy figyelmet az algoritmus kiválasztására.

A program tervezett felépítéséből adódóan egy olyan hibajavító könyvtárra volt szükség, amely támogatta az adatfolyamok (`Stream`-ek) kezelését is. A .NET platformon csak egyetlen ilyen implementáció volt elérhető: egy Reed-Solomon kódolást megvalósító osztálykönyvtár [20]. Így egyébként sem volt lehetőségem válogatni a különböző kódolási algoritmusok között, csak ha implementáltam volna egy saját könyvtárat. Az említett könyvtár cél platformja viszont eredetileg a .NET Framework volt. Szerencsére a .NET Foundation szolgáltat egy ingyenes konverziós eszközt, az Upgrade Asszisztenst, amely

segítségével át lehetett emelni a könyvtár forráskódját .NET Standard alapra, ezáltal integrálhatóvá vált a saját projektembe.

A Reed-Solomon kódolás szimbólumokként kezelve, blokkokba szervezve transzformálja az adatot.



11. ábra: A Reed-Solomon blokk felépítése (saját szerkesztés [21, Ábr. 2] alapján)

A könyvtárban implementált enkóder és dekóder 3 paraméterrel konfigurálható, amelyet részben a 11. ábra szemléltet:

- $n \in \mathbb{N}$ blokkméret, $n = 2^s - 1$, ahol $s \in \mathbb{N}$ a szimbólum méret [21]
- $k \in \mathbb{N}, k < n$ adat szimbólumok száma
- $p \in \mathbb{N}$ primitív polinom bináris reprezentációban, amelynek fokszáma $\log_2(n+1)$

A paritás szimbólumok száma $2t = n - k, t \in \mathbb{N}$ amely által $2t$ korrupt szimbólum állapítható meg, és t hibás szimbólum javítható. [21]

(Kísérleti úton megállapítottam, hogy $d \in \mathbb{N}$ bájt adat $e \in \mathbb{N}$, $e = \left\lfloor \frac{d}{k} + 1 \right\rfloor \cdot n$ bájt enkódolt adattá transzformálódik, valamint, hogy e bájt enkódolt adat $d = \left\lfloor \frac{e}{n} \right\rfloor \cdot k$ bájt dekódolt adatmennyiséget eredményez. Ez fontos információ volt az enkódolás és dekódolás során a megfelelő memória mennyiség allokálásához.)

Mivel a C# az adatfolyamokat bájtonként kezeli, érdemes $s = 8$ bites szimbólum méretet választani. Ha kisebbet választunk, fel kell tördelni a bájtokat kisebb adatstruktúrákra, amely felesleges számítási-, és memóriaigénnyel jár. Nagyobb szimbólumméret természetesen alkalmazható lenne, hiszen így már reprezentálható a bájt teljes értéktartománya, viszont a könyvtár a teljesítmény javítása érdekében a szorzás műveleteket egy szorzótábla segítségével végzi el, ezáltal minél nagyobb a szimbólumméret, annál magasabb a memóriaigény [20].

Egy gyakran alkalmazott konfiguráció az $n = 255, k = 223, s = 8$, amely $t = 16$ szimbólumot képes kijavítani, azaz 6.24% véletlenszerű meghibásodást. Ezt alkalmaztam, mivel kiegyensúlyozott opció volt a dekódolási sebesség [21], és a javítható hibák mennyisége között.

Az osztálykönyvtár a $p = x^8 + x^4 + x^3 + x^2 + x^0 \equiv 100011101_2 = 285_{10}$ primitív polinomot alkalmazta a tesztesetekben, ezért erre esett a választás, mert csak így lehettem biztos abban, hogy megfelelően fog működni.

3.3.3. Titkosítási algoritmusok

A titkosítás tervezése során alapvetően két irányban indulhattam el:

- Általánosítva használom a .NET-ben elérhető titkosítási algoritmusokat, valamint blokktitkosítási módokat, és emellett manuálisan valósítom meg a metaadatok, valamint a titkosított adatok hitelesítését, ezzel elkerülve további dependenciák hozzáadását.
- Olyan titkosítási algoritmusokat választok, amelyek alapvetően támogatják az adat, és metainformáció hitelesítést. Az ilyen képességekkel rendelkező titkosítási eljárásokat általánosságban AEAD[22] eljárásoknak nevezik.

Mivel kriptográfiai rendszerek tervezése és implementálása megfelelő szakirányú tudás hiányában nagyon rossz ötlet, természetesen a második opciót választottam, hogy minimalizáljam a hibalehetőséget. A .NET Standard viszont jelenleg csak egyetlen AEAD algoritmust támogat, és ez kifejezetten limitáló lett volna a programomra tekintve. Ezáltal egy külső osztálykönyvtárra, az NSec-re esett a választás, amely 3 AEAD eljárást is implementál [22]:

- AES-256-GCM, azaz AES algoritmus 256 bites kulcsmérettel, GCM[23] blokktitkosítási módban, hardveres gyorsítással (viszont csak x64-es processzorokon, AES-NI instrukciókkal támogatott [23]).
- ChaCha20-Poly1305, azaz ChaCha20 algoritmus, Poly1305 MAC[24]-kel, amelynek egyik előnye, hogy gyorsabb, mint a hardveres gyorsítás nélküli AES.

[22] Authenticated Encryption with Associated Data
[23] Galois/Counter Mode
[24] Message Authentication Code

- XChaCha20-Poly1305, amely hasonló, mint a ChaCha20-Poly1305, csak nagyobb nonce[25] méretet használ.

Az NSec előnye, hogy multiplatform, és egy egyszerű, modern, `Span<T>` alapú API-t szolgáltat. A .NET-tel szemben a kriptográfiai osztályainak nem szükséges implementálni az `IDisposable` interfészt, mivel a népszerű libsodium kriptográfiai könyvtáron alapul, ezzel biztosítva, hogy még véletlenül sem történik információ szivárgás egy helytelenül használt objektum miatt. [24]

Egy AEAD eljárás működtetéséhez 4 paramétert kell megadni az NSec esetében:

- A titkosítási kulcs, amely 256 bit méretű.
- A nonce, amely, ahogy a neve is utal rá, egy kulcsonként egyszer használatos szám, viszont nem titkos.
- A nem titkos, de hitelesítendő metainformáció. (Pl.: az alkalmazott titkosítási algoritmus.)
- A titkosítandó, vagy rejtjelezett adat, attól függően, hogy titkosítani, vagy visszafejteni szeretnénk.

A nonce értékének kiválasztásáról azt kell tudni, hogy katasztrofális következményekkel járhat, ha újra felhasználva, többször titkosítunk vele adatot ugyanazzal a kulccsal párosítva [25]. Mivel az alkalmazás jellegéből adódóan nincs lehetőségem arra, hogy megbizonyosodjak arról, hogy egy adott kulccsal még nem volt használva egy adott nonce, az egyetlen megoldás, hogy véletlenszerűen generálom. A nonce mérete AES-GCM és ChaCha20-Poly esetében 96 bit, XChaCha20-Poly esetében pedig 192 bit. Ezáltal csekély egy véletlenszerűen generált nonce ütközésének az esélye, de az XChaCha20 a legbiztonságosabb ilyen szempontból. (Az AES-GCM-nek is létezik egy nonce ütközés ellenálló verziója, az AES-GCM-SIV, ezt viszont az NSec sajnos nem támogatja.) A nonce véletlenszerűségének köszönhetően azonos bemeneti adatok mindig más-más rejtjelezett adatokká fognak transzformálódni, még azonos kulcs esetén is, így nem állapítható meg, hogy bármely két rejtjelezett adatfolyam ugyanazzal a kulccsal volt-e titkosítva. Természetesen a rejtjelezett adatfolyam visszafejtéséhez is szükség van a nonce-ra, ezért metaadatként ezt is tárolni kell.

[25] Number only used once

3.3.4. Jelszó alapú kulcs származtatás

A titkosítási kulcs mérete adott, így amennyiben nem véletlenszerűen generáljuk a kulcsokat, hanem valamilyen titkos forrás alapján, (esetünkben felhasználói jelszavak alapján,) szükség van egy olyan eljárásra, amely változó méretű bemenethez egy fix méretű kimenetet rendel, azaz egy hash függvényre [19, o. 10]. A biztonságos jelszó hasheléshez azonban az átlagos kriptográfiai hash függvény nem elégséges, mivel túlságosan gyors. Ahogy azt már korábban említettem, a gyors kulcsszámítás lehetségessé teheti a brute-force támadásokat a jelszó ellen. Tehát valamilyen specializált, jelszó hashelésre tervezett algoritmust kell alkalmazni.

A kriptográfiai közösségben az egyik ilyen széleskörben elfogadott jelszó hash függvény az Argon2, amely 2015-ben a PHC[26] nyertese lett [26], és az OWASP által is első sorban ajánlott [8].

Az Argon2-nek két verziója létezik az eredeti 2015-ös specifikáció szerint [27, o. 4]:

- Argon2d
- Argon2i

Az Argon2-re vonatkozó legújabb RFC kiadásban viszont már egy hibrid verziót, Argon2id-t említik elsődleges variánsként, amely a korábban említett verziók előnyeit, és hátrányait vegyíti [28].

Az Argon2 három paramétert szolgáltat az erőforrásigény konfigurálásához:

- Számítási igény: párhuzamosság foka p
- „Tér” igény: memória használat m [kB]
- Idő igény: iterációk száma t

A specifikáció továbbá paraméter javaslatokat tesz egyes felhasználási esetekre, valamint segítséget nyújt a saját paraméter választáshoz is [28, o. 11–12]. Mivel a specifikáció az Argon2id-t ajánlja, és az NSec is csak ezt az Argon2 variánst támogatja jelen pillanatban, az Argon2id-re kellett essen a választás. (A könyvtár függőségeket próbáltam minimálisan tartani.)

[26] Password Hashing Competition

Az Argon2id-hez a következő paramétereket ajánlja a specifikáció alapértelmezetten: $t = 1, p = 4, m = 2^{21}$. Ezen ajánlat viszont meglehetősen nagy memória igényt tüntet fel (2 GiB), ezáltal a második ajánlatra esett volna a választás: $t = 3, p = 4, m = 2^{16}$. Azonban az NSec könyvtár jelenleg csak $p = 1$ párhuzamossági fokot támogat, így megpróbáltam a legjobb belátásom szerint módosítani a paraméterezést. Mivel memória szempontjából limitált az alkalmazás, és a párhuzamossági fok nem változtatható, egyetlen lehetőségem volt: növeltem az időigényt, miközben a memóriaszükségletet ésszerű szinten tartottam: $t = 8, p = 1, m = 2^{18}$ (262 MiB).

Ideális esetben minden jelszóhoz hozzáadunk egy véletlenszerűen generált, úgynevezett salt értéket, ezzel különböző hash-eket generálva még egyező jelszavak esetén is. Ennek első sorban az a célja, hogy védjük a jelszavakat a rainbow táblás támadások ellen, amennyiben a tárolt hash értékük kiszivárog. Viszont az alkalmazásomnak felesleges kockázat bármilyen jelszót hash-t tárolni például hitelesítés céljából, ugyanis a titkosítás visszafejtése elbukik helytelen kulcs esetén. Így egyébként is szükségtelenné válik a salt, viszont, ha alkalmaznám további komplikációkat is okozna a program felépítéséből adódóan, amelyre a 4.1-es alfejezetben visszatérek.

3.3.5. Tömörítési algoritmusok

Minél kevesebb adatot rejtünk el, annál nehezebb detektálni a jelenlétét. Ezért természetesen érdemes tömöríteni az adatokat a rejtési folyamat végrehajtása előtt. Viszont mielőtt ezt megtehetjük, szükség van arra, hogy valamilyen módon összefűzzük a bemeneti fájlokat egy nagy adatfolyammá, valamint a hozzájuk tartozó metainformációkat (pl.: a fájl nevét, létrehozási dátumát) is tárolnunk kell.

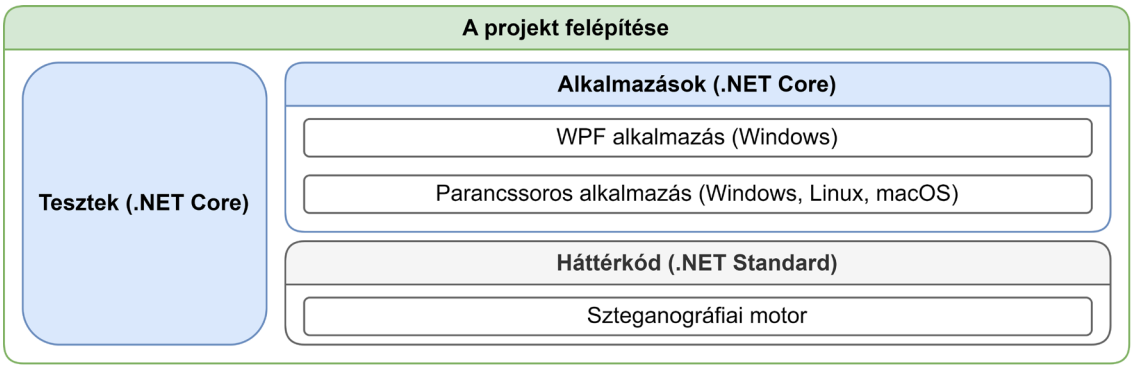
A TAR fájlformátumot pontosan olyan esetekre találták ki, amikor egy fájlrendszer nélküli „nyers” tárolóba kell fájlokat írni [29], és ráadásul egy szabad szabvány, tehát kiváló választás ezen probléma megoldásához. Viszont alapértelmezetten nem túl hatékony, ha kisebb fájlokat kell tárolni. A TAR formátum specifikációja szerint az archívumban minden adat 512 bájt méretű blokkokba szerveződik. Ezen blokkok úgynevezett rekordokba csoportosulnak. A rekordok alapértelmezetten 20 blokk, azaz 10240 bájt méretűek, tehát még ha ennél kisebb fájlokat is archiválunk, az elkészült TAR archívum mindig legalább ekkora méretű lesz. Szerencsére a rekordok mérete konfigurálható, blocking factor-nek nevezik. A blocking factor-t 1-re állítva a TAR archívum minimum

mérete 2048 bájtra csökken, és tömörítést alkalmazva természetesen kisebb is lesz, mint az eredeti fájlok összesített mérete. [30], [31]

A fájlok archiválásának megoldása után két tömörítési algoritmust választottam ki. Normál esetben tömörítési algoritmust a tömörítendő adatok jellegének megfelelően választunk, egy szteganográfiai alkalmazás esetében azonban nem igazán lehet megjósolni, milyen fájlok tárolására lesz használva. Ezért az általános tömörítési algoritmusok között válogattam. Az észlelhetetlenség növelése érdekében célszerű volt a lehető legmagasabb tömörítési aránnyal rendelkező algoritmust megkeresni. Egy, a Google által készített meggyőző összehasonlítás alapján esett a választás a Brotli-ra, a kiváló kitömörítési sebessége, és a magas tömörítési aránya miatt (11-es szinten) [32]. Tömörítési sebessége viszont kifejezetten alacsony, ezért szükség volt egy középútra is: egy gyors, elfogadható tömörítési aránnyal rendelkező algoritmusra. Erre a célra a Meta (Facebook) által fejlesztett Zstandard-et választottam. Így a felhasználónak lehetősége van választani a sebesség, és a tömörítési arány között.

4. Megvalósítás

Mivel fontos volt a multiplatform követelmény teljesítése, és úgy gondoltam az elkészült munka értékesebb lesz, ha a szteganográfiai motort különállóan tervezem meg, a projektet a következőképp szerveztem.



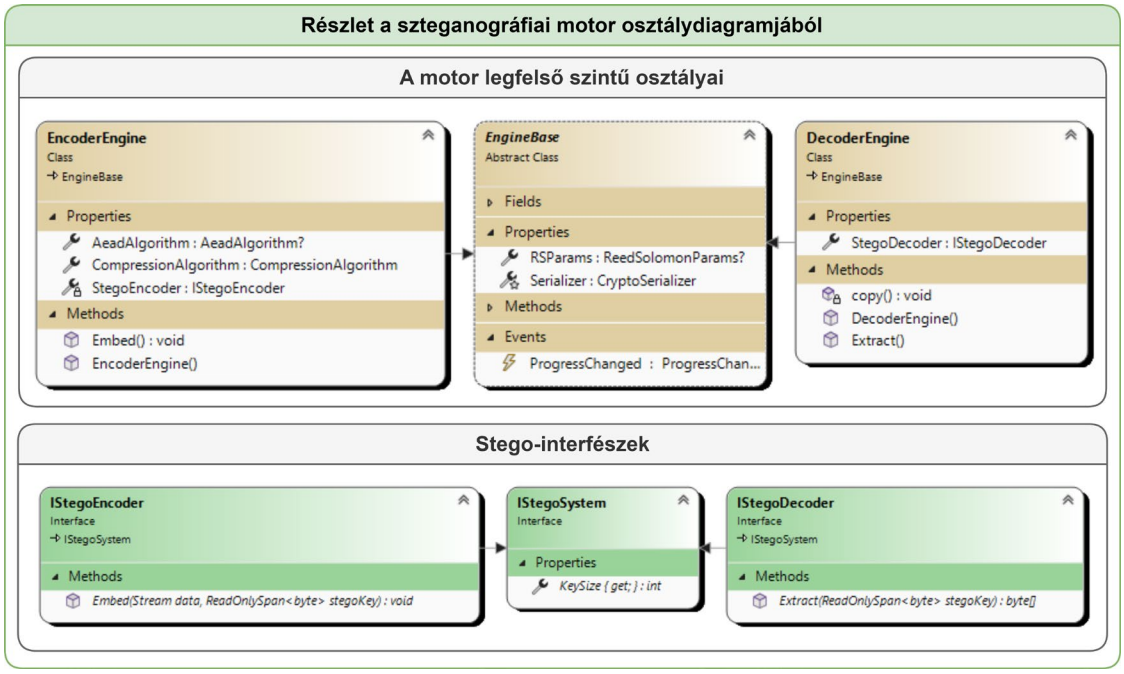
12. ábra: A projekt felépítése (saját szerkesztés)

Ahogy fent látható (12. ábra), a szteganográfiai motor .NET Standard-en, míg a projekt többi része .NET Core-on került megvalósításra. A motor esetében szem előtt tartva a platformok közötti kompatibilitást választottam meg az alkalmazott osztálykönyvtárakat, viszont egy közös, .NET Core alapú teszt projektben került tesztelésre minden funkcionalitás. Így tehát a motor bár elméletileg alkalmazható lenne például egy telefonos alkalmazásban Xamarin platformon, vagy egy asztali alkalmazásban .NET Framework-ön, és .NET Core-on bármilyen támogatott operációs rendszeren, a gyakorlatban nem lett teljesen validálva.

A motoron kívül elkészült két gyakorlatilag különálló alkalmazás: egy felhasználói felületes WPF, valamint egy multiplatform parancssoros program, amelyek, mivel megosztják a motor paraméterezését, és ugyan azt a szteganográfiai rendszer implementációt alkalmazzák, kompatibilisek egymással. Amint korábban említettem, minden háttér funkcionalitás unit tesztek segítségével lett ellenőrizve, (ehhez az NUnit programcsomagot alkalmaztam) így csak a felhasználói felület, és a parancssoros paraméterek tesztelésére volt szükség manuálisan.

A forráskód komplexitásából adódóan nem célszerű minden megvalósítási részletre kitérni, viszont a legfontosabb komponensek ismertetve lesznek.

4.1. A szteganográfiai motor



13. ábra: Részlet a szteganográfiai motor osztálydiagramjából (saját szerkesztés)

Fent látható (13. ábra), hogy maga a motor kifejezetten kevés tulajdonsággal és metódussal rendelkezik. Ez annak köszönhető, hogy a S.O.L.I.D. alapelveket követve (de nem teljesen betartva) valósult meg [33]:

1) **S**ingle responsibility principle
2) **O**pen/closed principle
3) **L**iskov substitution principle
4) **I**nterface segregation principle
5) **D**ependency inversion principle

Az első alapelv betartása érdekében például a motor egy közös absztrakt osztályból, az `EngineBase`-ből származtatva lett ketté osztva egy `EncoderEngine`-re és egy `DecoderEngine`-re. Ez azért hasznos, mert az adatrejtés és adat visszafejtés során szükséges paraméterek ezáltal nem keverednek, a közös funkcionalitás viszont megmarad. A részletes osztálydiagramon már látható (2. melléklet), hogy a motor által használt osztályok konstruktor injekcióval kerülnek átadásra, ezzel biztosítva a megfelelő inicializációt, viszont a motor használójára bízva az átadott osztályok konfigurálását. A fenti ábrát (13. ábra) (vagy a mellékletet) tovább vizsgálva az is észrevehető, hogy a motor önmagában

nem valósít meg szteganográfiai algoritmust, csak interfészt biztosít (függőség megfordítási elv).

A szteganográfiai motor tervezése során továbbá az állapot visszajelzésre is figyelmet fordítottam. Az osztály `ProgressChanged` eseményére feliratkozva követhető az adatrejtési és adat visszafejtési folyamat állapota, valamint, a log4net osztálykönyvtárnak köszönhetően részletesen naplózza a folyamat lépéseit is. Ezen felül a motor munkája nem csak megfigyelhető, hanem meg is szakítható `CancellationToken`-ek átadásával, ezzel támogatva az aszinkron programozást.

4.1.1. Az adatrejtési réteg

A motort tükrözve, egy közös funkcionalitást definiáló interfészből (`IStegoSystem`) leszármaztatva két interfészt definiáltam: egyet az adatrejtést megvalósító osztályhoz (`IStegoEncoder`), és egyet az adat visszafejtést megvalósító osztályhoz (`IStegoDecoder`). Ezáltal az implementáló nem kényszerül felesleges funkcionalitás megvalósítására, viszont semmi sem akadályozza meg abban, hogy egyetlen osztály formájában valósítsa meg mindkét interfészt, betartva az interfész szegregáció elvét. Az `IStego`-interfészeknek köszönhetően a motorhoz megvalósított szteganográfiai rendszerek szabadon konfigurálhatóak az általuk implementált fájlformátumnak, valamint algoritmusnak megfelelően, és teljesen elkülönülnek a motortól. A motor még az elkészült stego-fájl exportálásának és a kinyert adatok kimentésének a menetét sem határozza meg, ugyanis a feladatköre csak a memóriában lévő adatfolyamok kezelésére terjed ki (megint csak betartva az egy felelősség elvét). Ahogy látható (13. ábra), a motor szolgáltat továbbá az `IStego`-interfészek számára egy úgynevezett stego-kulcsot. A stego-kulcs szerepe, hogy valamilyen titok segítségével inicializálható legyen az implementált szteganográfiai rendszer, ezzel növelve a titkosított adatok biztonságát is. Ennek kapcsán viszont ki szeretnék fejteni egy problémát, amelyre a 3.3.4 alfejezetben utaltam.

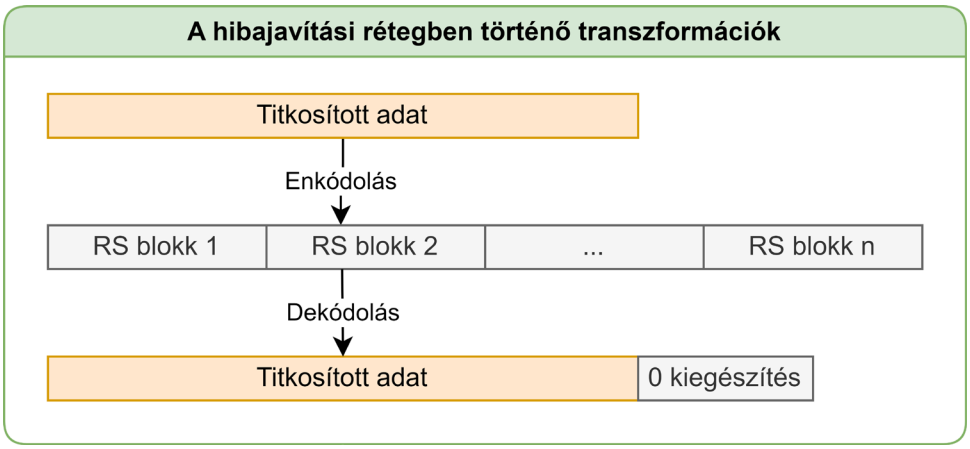
Amennyiben vissza szeretnénk fejteni adatainkat, az első lépés a stego-kulcs kiszámítása. Tegyük fel, hogy véletlenszerűen generált salt-ot használunk a jelszó hash kiszámításához (amelyből a stego-kulcs is számítva van). Ebben az esetben a stego-kulcs előállításához az adat visszafejtési folyamat legelején, az adatrejtési rétegben kell ismernünk a salt értékét, ez azt jelenti, hogy ott is kell tárolnunk. A probléma ezzel az, hogy az adatrejtési réteg a hibajavítási réteg alatt helyezkedik el, azaz a salt értéke nem kapna semmilyen védelmet az esetleges korrupcióktól, miközben kritikus információ az adatok

visszanyeréséhez. Ezáltal dönteni kellet a stego-kulcs, (amely fontos biztonsági funkcionalitás,) és a salt alkalmazása között. Amint korábban említettem, a salt használata egyébként is felesleges a program esetében, viszont amikor ebbe a problémába ütköztem, teljesen elvetettem a salt koncepcióját. (Egyéb, réteg szegregációs problémákat is okozna a salt használata, amelyre nem fogok kitérni.)

Az adatrejtési réteg az interfészek mellett tartalmaz egy `ByteStream` nevezetű absztrakt segédosztályt is, amely, ahogy a neve is utal rá, olyan adatfolyamok implementálásához nyújt alapot, amelyek bájtok szintjén manipulálják az adatokat.

4.1.2. A hibajavítási réteg

A motorban az egyetlen komponens, amely nem cserélhető, a hibajavítási réteg. Ez abból az egyszerű tényből következik, hogy ismernünk kell az alkalmazott hibajavító kódolást, valamint a paraméterezését ahhoz, hogy el tudjuk végezni a hibajavítást. Amennyiben általánosítjuk ezt a réteget, el kell tárolnunk a stego-fájlban azt a metaadatot, hogy milyen kódolás volt alkalmazva. Természetesen ezen a metaadaton nem alkalmazhatjuk azt a hibajavító kódolást, amelyet leír, hiszen nem lehetne visszafordítani. Alternatívaként a metaadathoz lehet alkalmazni egy fix hibajavító kódolást, miközben a valódi adathoz a metaadatban meghatározott hibajavító kódolást használjuk. Viszont ez egyrészt feleslegesen bonyolult, másrészt a metaadatot magas redundanciával kellene tárolnunk, amely értékes tárhelyet pazarol, harmadrészt a 3.3.2 alfejezetben már említettem, hogy a hibajavítás valójában csak speciális esetekben hasznos. Tehát az optimális megoldás az volt, hogy a hibajavítási rétegben alkalmazott kódolás fix, és a paraméterezése a motoron keresztül (`RSParams` tulajdonság), a programozó által határozható meg az adott implementációhoz. (Az `RSParams`-ot `null` értéken hagyva teljesen kikapcsolható a hibajavító kódolás.)



14. ábra: A hibajavítási rétegben történő transzformációk (saját szerkesztés)

A hibajavítási rétegben az átadott titkosított adatfolyam egy egymást követő Reed-Solomon blokkokból álló adatfolyammá alakul, a fent látható módon (14. ábra). Mivel a titkosított adat mérete nem mindig egészszerese a Reed-Solomon blokk üzenet szimbólum méretének, a legutolsó blokk esetenként kiegészítést kell alkalmazzon. Ennek következtében a dekódolás során megjelenik egy probléma: el kell távolítani a kiegészítést. Mivel a kiegészítést az enkóder automatikusan elvégzi, nem lehet speciális, minden esetben felismerhető kiegészítést alkalmazni. Így tehát nincs más lehetőség, mint eltárolni az adat méretét, és ez alapján egyszerűen lecsonkítani az adatfolyam végét. Az adatfolyam méretének tárolását viszont a titkosítási rétegre bíztam, és ennek hosszas indoklása van.

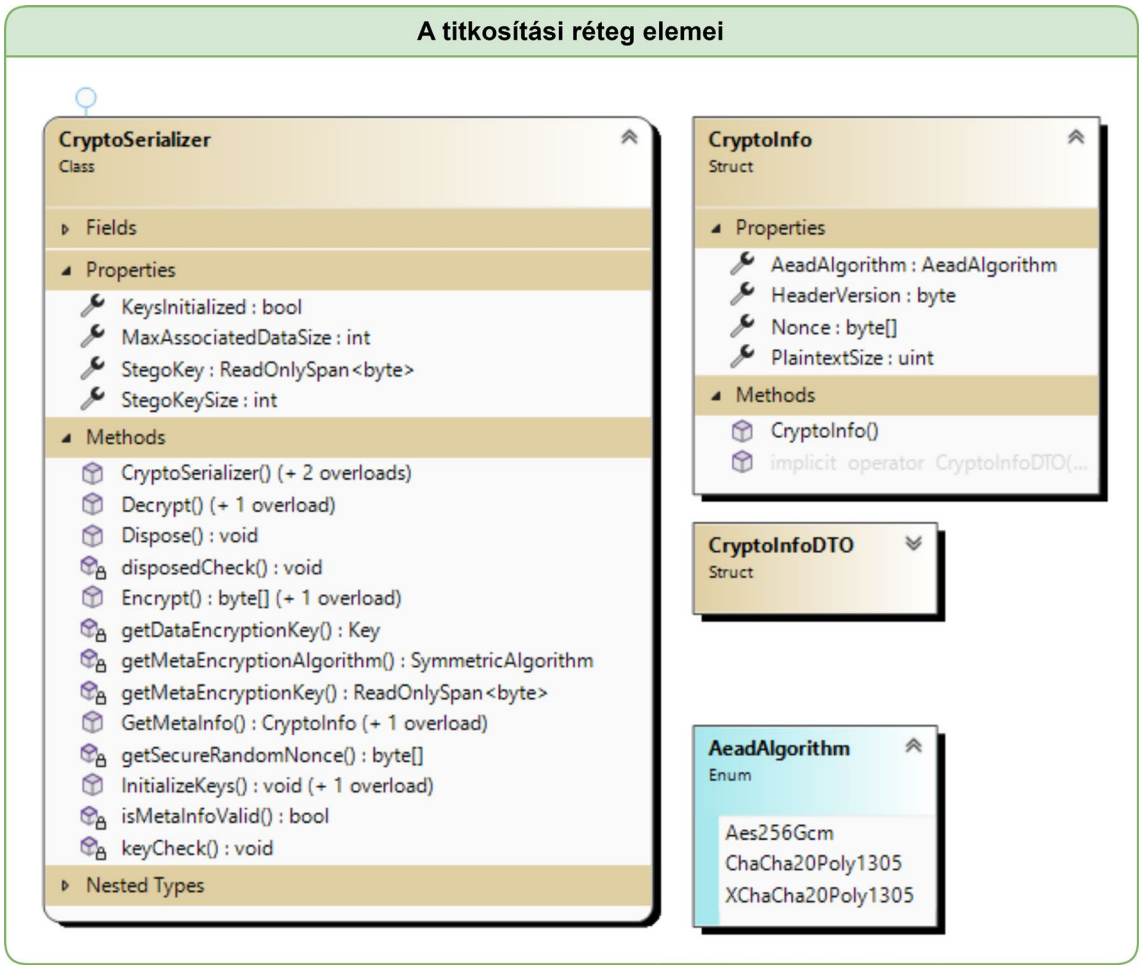
4.1.3. A rétegek közötti adatfolyamok csonkítása

Legyen a továbbiakban egy adott rétegben az eltárolt adat mérete L. A hibajavítási réteg létezésének következtében nem érdemes L-t az adatrejtési rétegben tárolni, hiszen így egy esetleges korrupció teljes adatvesztéssel járhat. (Technikailag ebben az esetben próbálgatással nyilván meg lehetne találni a valódi értéket, de nem praktikus.) Ha nem az adatrejtési rétegben tárolódik L értéke, a tervezés során feltételezni kell, hogy az adatrejtési réteg által kinyert adatfolyam nem csak információt, hanem felesleges adatot is fog tartalmazni az információ után. Ezt a felsőbb rétegekben el kell távolítani. Mivel a hibajavítási réteg opcionális, L értékének tárolása és az adatfolyam levágása így a titkosítási rétegben kell, hogy sorra kerüljön.

Ebből a megoldásból kifolyólag viszont megjelennek egyes teljesítménybeli problémák. Az első, hogy amennyiben egy adott szteganográfiai rendszer implementációja valóban felesleges plusz adatot ad vissza, a program eleinte a szükségesnél több memóriát

fog használni. Viszont hangsúlyoznám, hogy ez a legtöbb esetben maximum néhány megabájtot fog jelenteni, tehát elfogadható hátrány az *L* érték védelmének érdekében. A második, amely sokkal jelentősebb, hogy így alapvetően a hibajavítási rétegnek sokkal több adatot kellene dekódolnia, mint amennyi szükséges. Hogy ezt elkerüljük, a hibajavítás megkezdése előtt dekódolható az adatfolyam legelején elhelyezkedő kriptográfiai fejléc, amely tartalmazza *L* értékét. Ennek ismeretében a felesleges adat dekódolása (a 3.3.2 alfejezetben ismertetett egyenletek segítségével) már elkerülhető. Így amennyiben használva van a hibajavítási réteg, a felesleges adat ott lesz levágva, amennyiben nem, a titkosítási réteg egyébként is levágná, mivel *L* értékének megfelelően végzi a titkosítás visszafejtését.

4.1.4. A titkosítási réteg



15. ábra: A titkosítási réteg elemei (egyszerűsített) (saját szerkesztés)

A titkosítási réteg a `CryptoSerializer` osztály formájában valósult meg. Ahogy a 15. ábra (és a 2. melléklet is) szemlélteti, ezen osztály megvalósítása az egyik legkomplexebb, a használata viszont valójában kifejezetten egyszerű. Mielőtt bármit is teszünk az osztállyal, először inicializálni kell a titkosítási kulcsokat. Ez kétféleképp tehető meg: vagy példányosítás során a konstruktor paraméterezésével, vagy utólag, az `InitializeKeys` metódus meghívásával. Az osztály készenlétét a `KeysInitialized` tulajdonság jelzi.

Látszólag a kulcsok külön kezelése feleslegesnek tűnhet, hiszen az `Encrypt` és `Decrypt` metódusok meghívása során is kiszámíthatóak lennének. Viszont tekintve, hogy a kulcsok kiszámítása egy hosszas művelet, nem lett volna előnyös az osztály újrahasználhatóságának szempontjából: ha több adatfolyamot szeretnénk titkosítani ugyan azzal a jelszóval, miközben nincs elkülönítve a kulcsszámítás, feleslegesen számoljuk ki újra és újra ugyan azt kulcsot.

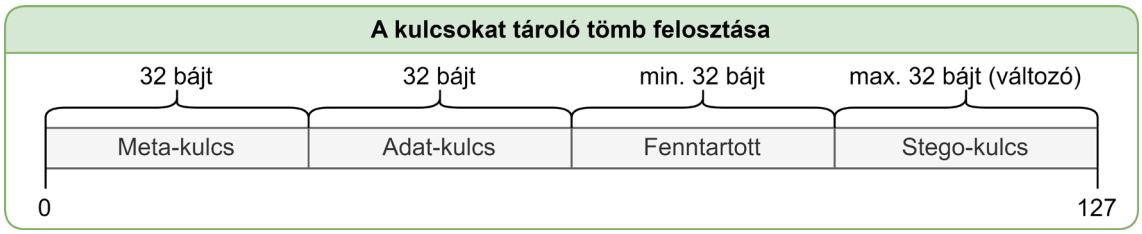
Fontos gondolni a kulcsok memóriában lévő kezelésére is. A program életciklusa során a GC[27] gyakran mozgatja a menedzselt memóriában lévő adatokat, hogy optimalizálja a teljesítményt. Ez viszont a kulcsokra nézve biztonsági kockázatot jelent. A GC ugyanis az objektumok mozgatása során csak másolatot készít azokról, miközben az eredetileg foglalt memóriában megmaradnak az értékek.

```
_pbkdfBytes = new byte[_pbkdfBytesSize];
_pbkdfBytesHandle = GCHandle.Alloc(_pbkdfBytes,
GCHandleType.Pinned);
pbkdf.DeriveBytes(password, salt, _pbkdfBytes);
```

1. kódrészlet: A kulcsok memóriában való mozgatásának megakadályozása (saját kód)

Hogy kiküszöböljem ezt a problémát, a kulcsokat tároló tömböt lerögzítettem a memóriában, közvetlenül a kulcsok kiszámítása előtt (1. kódrészlet). (A kódrészletben látható salt érték egy egyszerű nullákkal inicializált tömb.) Az osztály életciklusának végén természetesen a kulcsokat megfelelően meg kell semmisíteni, ehhez kézenfekvő megoldás volt az `IDisposable` interfész implementálása. A `Dispose` metódus meghívása során egyszerűen csak felül kell írni a kulcsokat tartalmazó tömböt valamilyen értékekkel, például nullákkal, és fel kell szabadítani a GC számára.

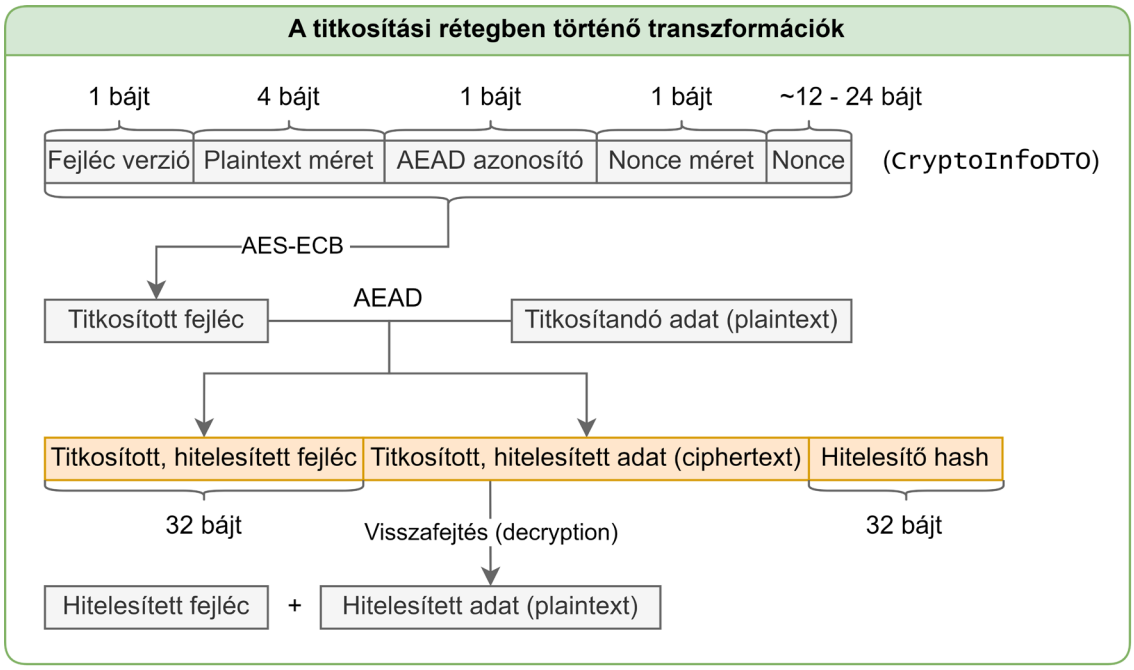
[27] Garbage Collector



16. ábra: A kulcsokat tároló tömb felosztása (saját szerkesztés)

Mivel általánosságban ésszerű egy kriptográfiai titkot minél kevesebbszer felhasználni, (hogy a titok szivárgása esetén csökkentsük a járulékos károkat,) minden egyes titkosítási feladathoz külön kulcsot alkalmaztam. Elsőre furcsának tűnhet, hogy minden kulcs egyetlen tömbben tárolódik, viszont ez egyszerűsíti a korábban említett memória kezelési problémát: a `ReadOnlySpan<T>` típus használatának köszönhetően lehetséges volt a kulcsokat tároló tömb szegmenseire hivatkozva hozzáférni az egyes kulcsokhoz, ezzel továbbra is elkerülve a kulcsok másolását. Az említett szegmensek elhelyezkedését és méretét a 16. ábra szemlélteti. Természetesen mivel a jelszó feldolgozását a `CryptoSerializer` végzi, értelemszerű, hogy a stego-kulcsot is ő szolgáltatja. (A stego-kulcs méretét az osztály `StegoKeySize` tulajdonsága határozza meg.)

Visszatérve az osztály használatára, a titkosítási kulcsok inicializálása után már meghívhatóak az `Encrypt` és `Decrypt` metódusok.



17. ábra: A titkosítási rétegben történő transzformációk (saját szerkesztés)

A titkosítási folyamat első lépéseként amennyiben nem lett specifikálva az `AeadAlgorithm` paraméter értéke (`null`), egy CSRNG[28] használatával véletlenszerűen kiválasztásra kerül egy titkosítási algoritmus. Ezt követően elkészül a kriptográfiai fejléc: a (szintén egy CSRNG által) véletlenszerűen generált nonce, valamint a metainformációk egy `CryptoInfoDTO`[29] struktúrába szerveződnek (17. ábra), (amely a `CryptoInfo` struktúra primitív típusokból álló változata,) és a StructPacker programcsomag segítségével bináris adatfolyammá alakulnak. Normál esetben a metainformációkat tartalmazó fejléc nem lenne titkosítva, viszont annak érdekében, hogy a program ne rendelkezzen felismerhető szignatúrával, (a fejléc struktúrája ismert, és kereshető,) egy egyszerű AES-ECB[30] titkosítási eljárást alkalmazok, a meta-kulcs felhasználásával. Hangsúlyozom, hogy ennek nem célja a kriptográfiai biztonság szolgáltatása, (bár a rejtett adat mérete titkos,) csak potenciálisan megnehezíti a program támadását egy rosszul implementált szteganográfiai rendszer esetén. Továbbá kiemelném, hogy a fejléc felépítése szándékosan olyan módon lett megtervezve, hogy a nonce értéke átfedje a két 16-16 bájt méretű AES blokkot. Ezáltal a fejléc alapján sem állapítható meg, hogy ugyanazzal a jelszóval volt-e titkosítva bármely két adatfolyam. Mivel a fejléc tartalmazza az alkalmazott AEAD algoritmus azonosítóját, a jövőben hozzáadhatóak új kriptográfiai eljárások is, megtartva a régiek támogatását. Ezzel megvalósul a kriptográfiai agilitás követelménye, viszont nyilván a fejléc titkosítási eljárása nem módosítható. A fejlécben látható verziószám azt a célt szolgálja, hogy visszafelé is kompatibilis maradhasson a szoftver egy esetleges fejléc módosítás után. A titkosítási folyamat utolsó lépéseként végbe megy az AEAD titkosítás az adat-kulcs felhasználásával, itt asszociált adatként természetesen a titkosított fejléc van megadva, ezáltal hitelesíthető a visszafejtés során.

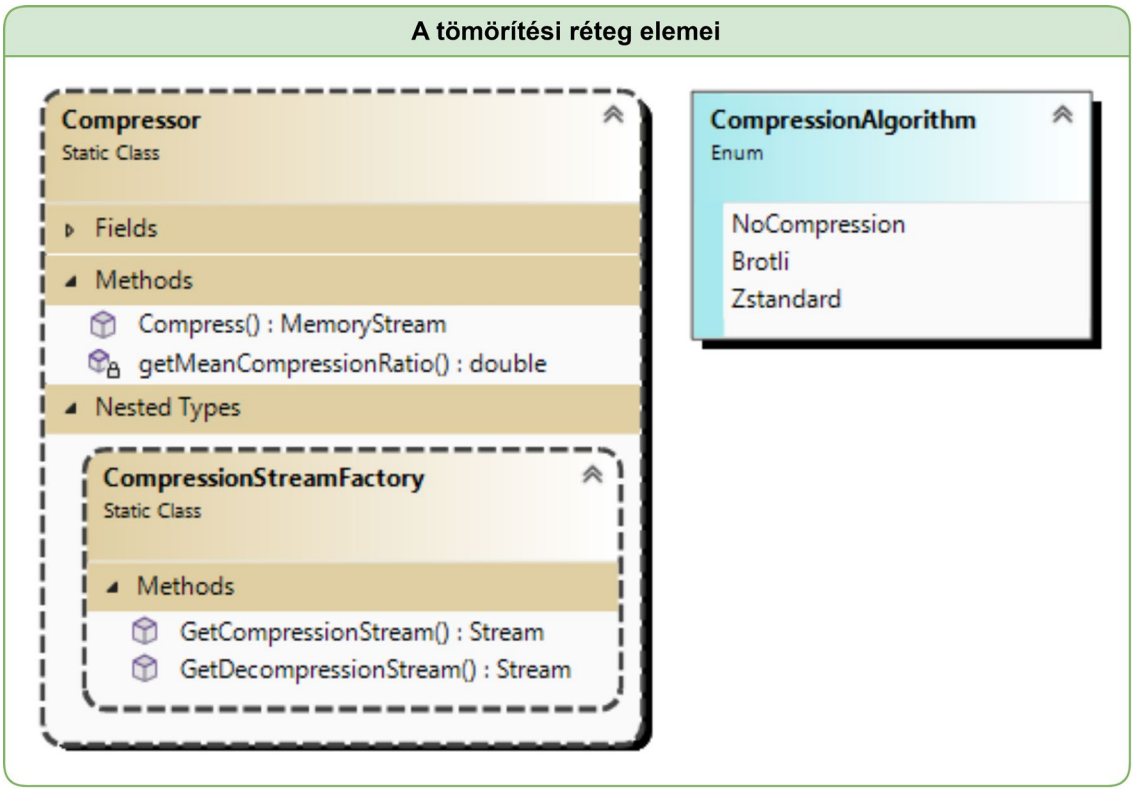
A titkosítás visszafejtése a fejléc feloldásával kezdődik. Sajnos az NSec könyvtár egy limitációja, hogy az asszociált adat (tehát a fejléc) csak az AEAD titkosítás visszafejtése során hitelesíthető, így potenciálisan korrupt fejléccel kell dolgozni ideiglenesen. Ez biztonsági kockázatot a gyakorlatban nem igazán jelent, a legrosszabb dolog, ami történhet, hogy megpróbál túl sok memóriát lefoglalni a program az adatfolyam csonkítása során. Miután elhatárolódott a valódi információ a felesleges adattól, megkezdhető az AEAD titkosítás visszafejtése a fejlécben szereplő információk szerint. Ha sikerült eljutni

[28] Cryptographically Secure Random Number Generator
[29] Data Transfer Object
[30] Electronic Codebook

a programnak eddig a pontig egy korrupt fejléccel, az AEAD hitelesítési folyamata természetesen el fog bukni. (A hitelesítés az adatokra is vonatkozik, tehát semmiképpen sem fog korrupt, vagy szándékosan módosított információt kinyerni a program.) Egyéb esetben ezen lépés után véget ér az adatok feloldásának folyamata, és a `Decrypt` metódus visszatér a hitelesített fejléccel, valamint az adatokkal.

4.1.5. A tömörítési réteg

A S.O.L.I.D. elveket követve, a tömörítést eredetileg egy `ICompressor` interfészt megvalósító injektált osztály hajtotta volna végre egy `Compress` és egy `Decompress` metóduson keresztül. Ezzel a megközelítéssel viszont problémákba ütköztem. Egyrészt, mivel a tömörítés állapot nélküli, felesleges egy osztálypéldányt létrehozni: a tömörítés végrehajtásához csak a tömörítendő adatfolyam szükséges és elvégezhető egy egyszerű statikus metódussal is. Viszont statikus metódusok jelenleg nem definiálhatóak egy interfészen. Másrészt, a visszafejtési folyamat során a motor használójának újra meg kellett volna adnia a megfelelő osztálypéldányt a kitömörítés elvégzéséhez, ennek következtében a motor használójának is kellett volna megjegyeznie, hogy milyen tömörítési algoritmus volt alkalmazva egy adott rejtési folyamat során. Ez a gondolatmenet azt sugallja, hogy alapvetően felesleges a tömörítést a szteganográfiai motorra bízni, érdemes teljesen elkülöníteni. Azonban, mivel törekedtem arra, hogy a motort minél egyszerűbb legyen alkalmazni, mindenképp kellett találnom egy megoldást.



18. ábra: A tömörítési réteg elemei (saját szerkesztés)

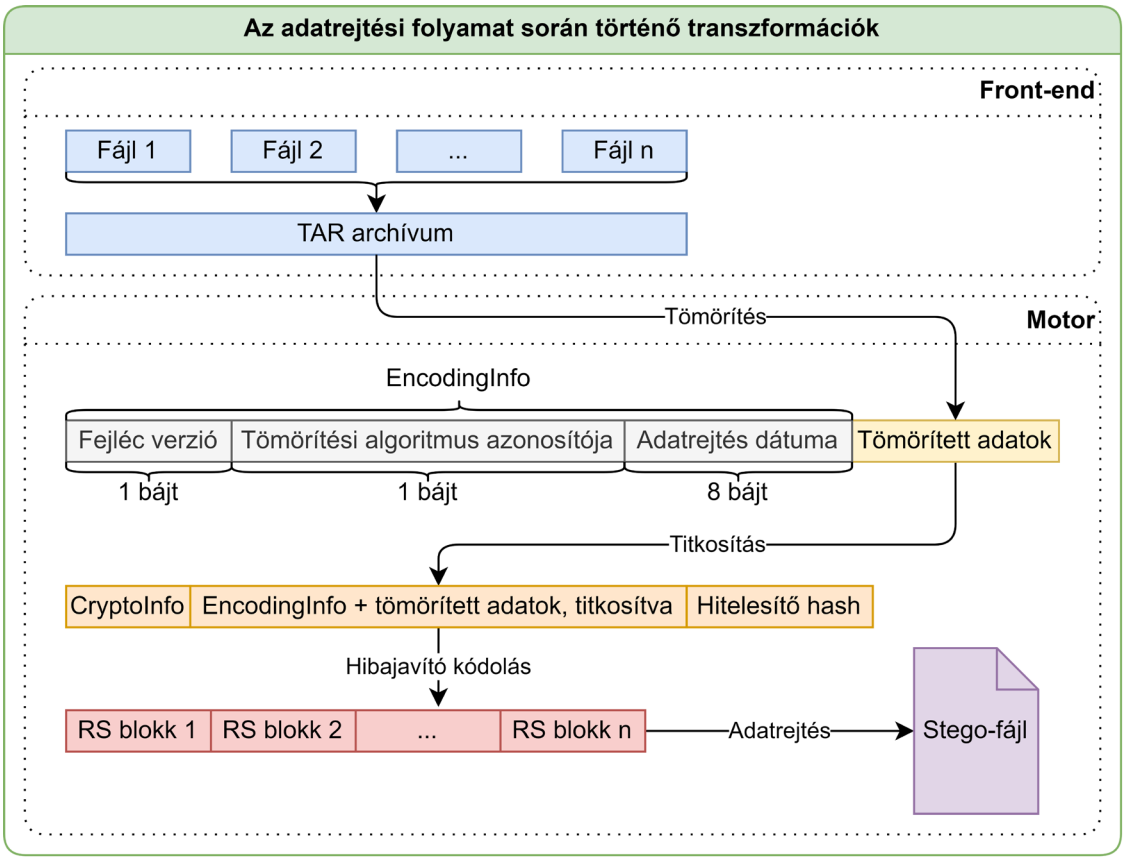
Ahogy a 18. ábra szemlélteti, a tömörítési réteg végül egy statikus osztály formájában valósult meg. Mivel a tömörítés és kitömörítés eljárások között sok az átfedés, egyetlen `Compress` függvény van, amely számára egy (beépített) `CompressionMode` enumerátor segítségével specifikálható, hogy milyen műveletet végezzen.

Magának a tömörítési algoritmusnak a kiválasztása a `CompressionAlgorithm` enumerátor megadásával történik, ez alapján példányosul a tömörítő `Stream` a `CompressionStreamFactory` segítségével. Technikailag szebb megoldás lett volna, ha a `NoCompression` opció helyett `nullable` típusként definiálom az enumerátort, viszont a `nullable` típusoknak szerializáció során plusz egy bájtot kell foglalni, és ez tárhely pazarlás lenne. (Tárolni kell az alkalmazott tömörítési algoritmus azonosítóját, hogy az adat visszafejtés során automatikusan kitömöríthető legyen az adat.)

Összességében ennek a megoldásnak köszönhetően triviális további tömörítési algoritmusok hozzáadása, és amennyiben a motor használója saját tömörítési algoritmust kíván alkalmazni, egyszerűen csak specifikálnia kell a `NoCompression` értéket.

4.1.6. A rétegek együttműködése

A rétegek működésének ismeretében tekintsük át röviden, hogyan alakulnak át a rejtendő adatok az adatrejtési folyamat során.



19. ábra: Az adatrejtési folyamat során történő transzformációk (saját szerkesztés)

Ahogy a 3.3-as alfejezetben utaltam rá, a motor rétegei jól megfeleltethetőek az adatrejtés lépéseinek. Fent látható (19. ábra), hogy az adatrejtés első lépéseként a motor tömöríti az adatokat, majd hozzájuk csatol egy metainformációkat tartalmazó fejlécet. Ezen fejléc tartalma, hasonlóan a titkosítási réteg fejlécéhez, egy struktúrába van szervezve (`EncodingInfo`), majd pedig szerializálva. Miután a motor előkészítette az adatokat, már ismerős lépések következnek: a tömörített adatok titkosítva lesznek, enkódolva, majd a stego-kulccsal együtt átadásra kerülnek a szteganográfiai rendszert (adatrejtő réteget) implementáló osztálynak.

Az adatok visszafejtéséhez fordított irányban kell elvégezni a lépéseket, azzal a különbséggel, hogy a folyamat során meg kell szabadulni az esetleges felesleges adatoktól. Ennek a logikáját már korábban ismertettem a 4.1.3-as alfejezetben.

4.2. A parancssoros alkalmazás

Mivel a feladat nehezét a szteganográfiai motor végzi, az alkalmazások megvalósításához csak az I/O[31] kezelését, valamint a kiválasztott szteganográfiai algoritmust kellett implementálni. A parancssoros program alapvetően (egy speciális esetet kivéve[32]) nem interaktív módon lett megtervezve, az induláskor megadott paraméterek alapján elvégzi a feladatát, majd kilép. Ebből kifolyólag a főprogram felépítését egyszerűbb volt procedurális stílusban, egymást követő metódushívásokra lebontva megvalósítani (3. melléklet).

Az alkalmazás multiplatform, a digitális mellékletek között elérhetőek bináris állományok Windows-ra, és Linux-ra, amelyek függőségek nélkül indíthatóak a megfelelő rendszeren. (Linux esetében csak az Ubuntu volt tesztelve.) A program segédlete a `help`, `-h` vagy `--help` paraméterekkel érhető el.

Az adatrejtést megvalósító osztályokat az `IStegoAlgorithm` interfész általánosítja, és a `StegoSystemFactory` példányosítja (3. melléklet). Ennek a megoldásnak köszönhetően a program különösebb módosítás nélkül bővíthető további adatrejtő algoritmusokkal. Kiemelném, hogy a szoftverspecifikációban meghatározott használati esetektől eltérő módon (3.1.2-es alfejezet), az alkalmazott algoritmust specifikálni kell, és nem a program választja a hordozó fájlformátuma alapján. (Jelenleg viszont van alapértelmezés, mivel csak egy algoritmus lett implementálva.) Így a program sokkal rugalmasabb lehet az algoritmusokra tekintettel, nincsen fájlformátumokhoz kötve. (Az egyszerű használhatóság érdekében a WPF alkalmazás esetében nem tértem el ilyen módon a specifikációtól.)

4.2.1. A paraméterek feldolgozása

A bemeneti paraméterek feldolgozását a Mono.Options parancs értelmező csomag segítségével (amely számos paraméterezési konvenciót támogat, ezáltal kiváló választás volt), egy külön osztályban implementáltam (`CommandParser`). Ez az osztály a program életciklusa során egyszer használatos, ezért (egyéb okok mellett) singleton-ként valósult meg. Mivel a program két irányba indulhat el az alapján, hogy adatot szeretnénk rejteni, vagy adatot visszafejteni, két fő parancs áll rendelkezésre: az `embed` és az `extract`.

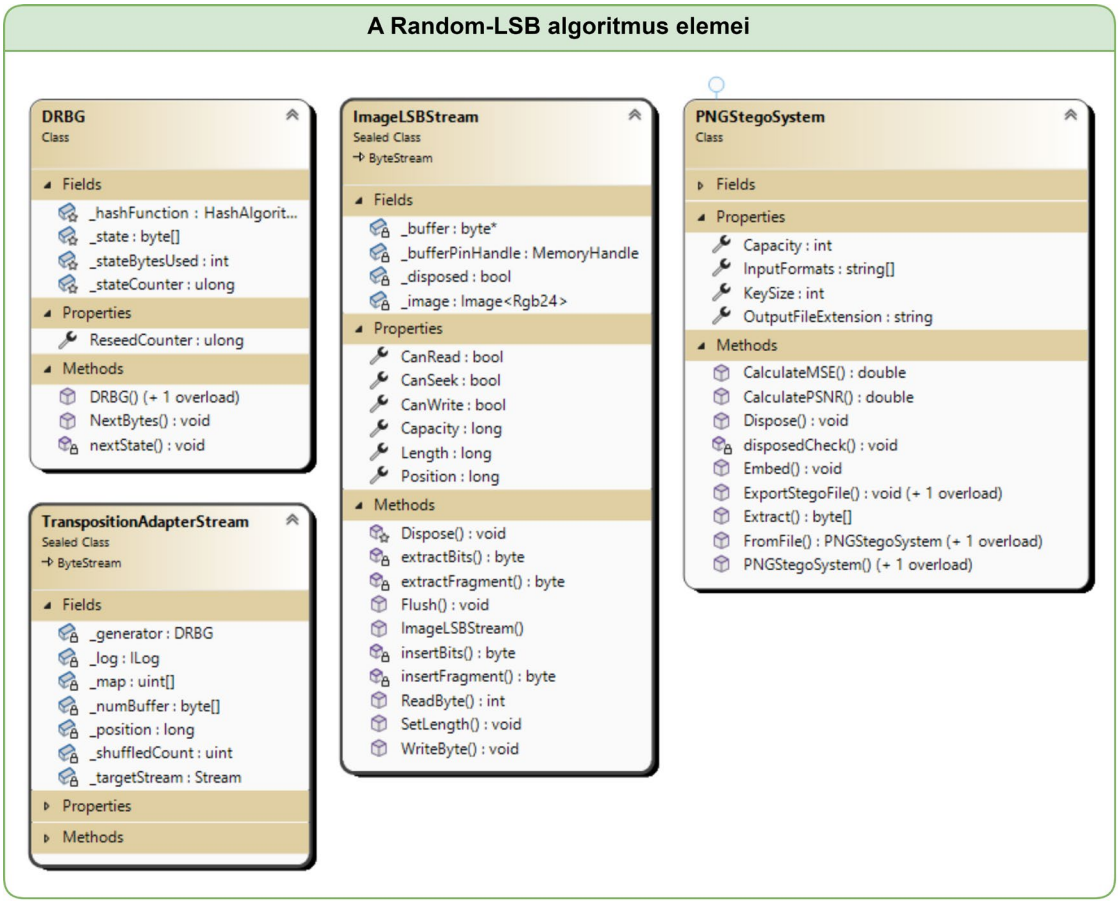
[31] Input/Output

[32] Amennyiben nem adunk meg jelszó paramétert, a program interaktív módon kéri be, és biztonságosabb módon, egy `SecureString` objektumban rögzíti.

Ezen parancsok paraméterezése között (a motort tükrözve) van valamennyi átfedés, de alapvetően érdemes volt őket elkülöníteni. Erre szolgálnak az `EmbedParams` és az `ExtractParams` struktúrák. Miközben a program indulásakor a `CommandParser` sorban dolgozza fel a paramétereket, egyúttal fel is tölti a választott parancsnak megfelelő struktúrát. (A Mono.Options működéséből adódóan az említett struktúrákat osztályszintű tulajdonságként kellett kezelnem.) Egy ideiglenes paramétervalidációt követően (pl.: léteznek-e a fájlok, megvan-e adva minden szükséges paraméter) a főprogram már használhatja is a megadott paramétereket.

Megvalósult a standard bemenet, és a standard kimenet átirányításának kezelése is, viszont a használt parancssor karakterkódolásának függvényében eltérő sikerrel működik. A program az elkészült stego-fájlt a standard kimenetre fogja kiírni, amennyiben át lett irányítva. Ilyenkor, ha például PowerShell-t használunk, az UTF-16 karakterkódolás következtében a kimentett fájl korrupt lesz. A standard bemenet átirányítása során az ott érkező adatok kerülnek a stego-fájlba, és nem történik TAR archiválás, mivel nem ismertek az adatfolyammal kapcsolatos metainformációk. Ennek következtében a visszafejtett adatokról nem feltételezhető, hogy mindig TAR formátumúak lesznek: amennyiben sikertelen a kiolvasott adatok TAR kicsomagolása, a program egyszerűen kimenti a nyers adatfolyamot.

4.2.2. A Random-LSB módszer megvalósítása



20. ábra: A Random-LSB algoritmus elemei (saját szerkesztés)

Ahogy a 20. ábra szemlélteti, a Random-LSB algoritmus újrahasznosítható módon, több osztályra bontva valósult meg. (Ez a felépítés a tesztelést is elősegítette.) A képfájlok dekódolása, és enkódolása a SixLabors.ImageSharp könyvtár segítségével történik. Ennek köszönhetően a program képes számos bemeneti képformátumot elfogadni, (TGA, TIFF, WEBP, PBM, JPEG, BMP, PNG, és GIF) miközben a memóriában általánosan, sRGB színtérben manipulálhatja azokat.

Maga az adatrejtés átlátszó módon, az `ImageLSBStream` osztályban történik, amely, ahogy a neve is utal rá, egy adatfolyam formájában teszi elérhetővé a betöltött kép LSB értékeit, mintha egy tárolóhely lenne. Fontos teljesítmény szempontjából a memóriában lévő pixelek elérésének, és módosításának a sebessége. Mivel az adatfolyamot bájtonként kellett feldolgozni, ráadásul a kép bármely pontját véletlenszerűen elérve, így három lehetőségem volt:

- Az ImageSharp indexelő operátorának a használata
- A pixel pufferhez való közvetlen hozzáférés
- A pixel puffer másolása menedzselt memóriába

Egy rövid tesztelés során mindhárom lehetőséget kipróbáltam. Annak ellenére, hogy az ImageSharp indexelő operátora a dokumentáció szerint egy nagyságrenddel gyorsabb, mint a `System.Drawing` névtérben elérhető függvények [34], mégis elfogadhatatlan teljesítményt nyújtott. A legjobb sebességet a pixel puffer másolása biztosította volna, viszont a hatalmas memóriapazarlás nem indokolta a használatát az egyébként is elfogadható sebességgel bíró közvetlen hozzáférés felett, így az utóbbi volt a legoptimálisabb megoldás, még ha nem is problémamentes. Ugyanis így `unsafe` kontextusban, pointerek segítségével kell kezelni a pixel puffert, tehát kritikus a pontos memóriacím számítás. Emellett ahhoz, hogy az ImageSharp képes legyen szolgáltatni a pixel pufferre mutató pointert, specifikálni kell a kép betöltése során, hogy egyetlen blokk memóriába töltse azt be. Ebből kifolyólag a memória kezelés hatékonyságának csökkenése mellett kellően nagy felbontású képek memória allokációja során elméletileg `OutOfMemoryException`-t dobhat a program, ha nem talál kellő hosszúságú folytonos memóriát.

Az LSB értékek felülírása adattal, valamint az adatok kiolvasása triviálisan megvalósítható néhány bitszintű művelettel, ezért a lépéseit nem részletezem, viszont az `ImageLSBStream` forráskódjában jól követhető.

Az adatok véletlenszerű szétszórását a `TranspositionAdapterStream` végzi, amely folyamatosan változtatja az alatta lévő adatfolyam (`ImageLSBStream`) pozícióját minden műveletvégzés előtt. Ahhoz, hogy a véletlen-pozíció ismétlődéseket elkerüljem több lehetőségem is volt, a legoptimálisabbnak viszont azt a megoldást tartottam, ha egy előre legenerált index táblázatot összekeverek. Mivel a legközelebbi primitív típus, a `ushort` túl kicsi volt, ez egy `uint` típusú számokból álló tömböt jelent, amely egységenként 4 bájt méretű. Ennek következtében elméletben maximum 4 GiB adat indexelhető, viszont az átlagos képfájl kapacitása sokkal kisebb ennél, nem is beszélve a memóriaszükségletről. Keverő algoritmusként a Fisher-Yates-re esett a választás, mivel egyszerű implementálni, és hatékony. (A saját implementációm esetében annyiban módosul az algoritmus, hogy csak szükség szerint történik keverés, azaz nincsen előre megkeverve a teljes index tömb.)

A Fisher-Yates-nek viszont szüksége van egy véletlen-forrásra, és a beépített véletlenszám generátorok seed mérete nem kellően nagy, hogy megfelelő biztonságot nyújtsanak. Szerencsére kriptográfiai hash függvényekkel viszonylag egyszerű determinisztikus véletlenszám generátorokat készíteni (`DRBG`[33] osztály), így nem volt szükség további osztály könyvtárak keresésére. Egy, a NIST[34] által definiált konstrukcióhoz hasonló megoldás lett implementálva, amely a belső állapotot egy számlálóval kiegészítve mindig önmagával hash-eli [35]. Hash függvényként a publikációban is meghatározott SHA-256-ot alkalmaztam, így elméletileg 2^{48} állapotváltás vihető végre biztonságosan, mielőtt új seed-et kellene generálni [36, o. 38]. (Ez bőven azon a tartományon belül van, amit a `ulong` típusú számláló reprezentálni képes.) Hangsúlyozom, hogy a `DRBG` osztály implementációja több, mint valószínű, hogy tartalmaz valamilyen sebezhetőséget. Viszont nem állapítható meg vele kapcsolatban olyan egyértelmű biztonsági probléma, amely indokolná más megoldások keresését.

Az algoritmus használatához szükséges interfészeket a `PNGStegoSystem` osztály valósítja meg, amely példányosítja és használja a korábban említett osztályokat, valamint általános információkat szolgáltat. Az elkészült PNG fájl exportálása a lehető legmagasabb tömörítési opcióval történik, az LSB adatrejtésből származó tömörítési hatékonyság csökkenés korrigálása érdekében. Ennek következtében az adatrejtési folyamat leglassabb lépése általában az elkészült fájl kiexportálása.

4.3. A WPF alkalmazás

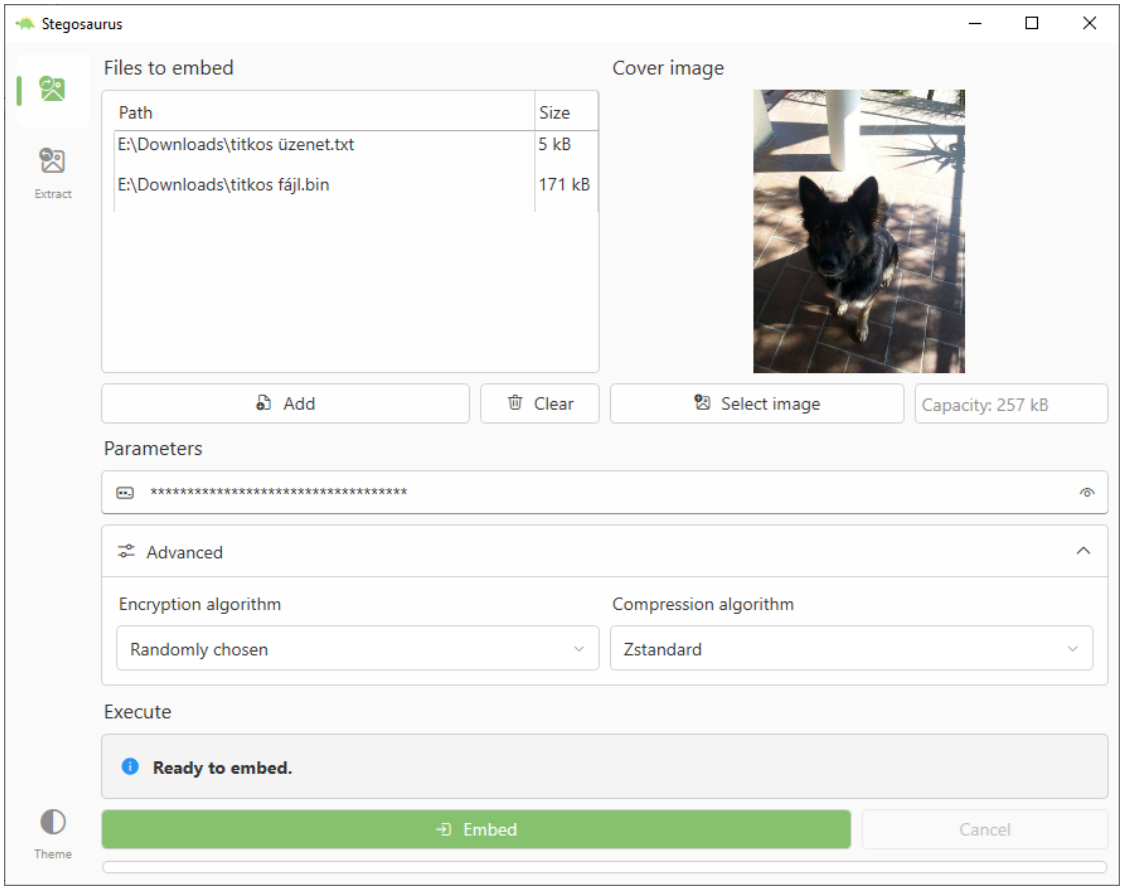
A felhasználói felületes alkalmazás a WPF technológiának megfelelően az MVVM[35] architektúrát követve valósult meg (4. melléklet). Mivel az adatrejtési algoritmus már elkészült a parancssoros alkalmazásban, és a szteganográfiai motor elkülönül, így valójában csak a felhasználói felület (View), és az interakciós logika (ViewModel) megvalósítása volt szükséges. Az alkalmazás felhasználói segédlete az 5. mellékletben található, és a futtatható állományok a digitális mellékletben érhetőek el.

A felhasználói felület és a háttérkód adatkötések segítségével, az `INotifyPropertyChanged` interfészen keresztül kommunikál. A PropertyChanged.Fody csomagnak köszönhetően viszont egy `ViewModelBase` nevezetű absztrakt

[33] Deterministic Random Bit Generator
[34] National Institute of Standards and Technology
[35] Model-View-ViewModel

osztály implementálásával automatikusan megvalósult az interfész, ezzel sok felesleges kódot megtakarítva. A gombnyomások kezelésével szintén egy segédosztály, a `RelayCommand` lett megbízva, amely, mivel az `ICommand` interfészt implementálja, hozzákötheto egy adott gomb `Command` tulajdonságához, és gombnyomás esetén meghívja a gombhoz tartozó metódust.

4.3.1. A felhasználói felület



21. ábra: A felhasználói felület adatrejtési nézete (saját képernyőkép)

Bár a WPF alapértelmezett kinézete már kifejezetten idejétmúlt, továbbra is igaz, hogy kiválóan testre szabható. Természetesen saját kontrollok tervezése, és implementálása egy hosszas folyamat, ebből kifolyólag sokkal érdemesebb volt egy stílus csomagot alkalmazni: a WPF-UI a Windows 11 tervezési irányzatait megközelítő felhasználói felület csomag, és egyszerű használatának köszönhetően kiváló választás volt egy egységes kinézet kialakításához.

Ahogy fent látható (21. ábra), a program nézetei a baloldali navigációs sávon válthatóak, valamint a navigációs sáv láblécén található egy téma váltó gomb is. (A program

alapértelmezetten a rendszernek megfelelő témával indul, az elsődleges szín viszont szándékosan zöld marad.)

A Random-LSB adatrejtési algoritmusból kifolyólag a felület kifejezetten képfájlokba rejtő algoritmusokhoz lett tervezve, viszont nem igényel kimondottan sok változtatást amennyiben ezt meg szeretnénk változtatni. Egyszerűen csak le kell cserélni a hordozó-kép (cover-image) előnézetét egy általánosabb megoldásra, és át kell venni a parancssoros alkalmazás által definiált interfészeket. Ügyelve a memóriahasználatra, az előnézeti képet a program alacsonyabb felbontásban tölti be.

Amint a szoftverspecifikáció használati esetei meghatározták (3.1.2-es alfejezet), az adatrejtéshez a felhasználónak csak néhány paramétert kell megadnia: a rejtendő fájlokat és a hordozófájlt, amelyeket fogd-és-vidd módszerrel is átadhatja, opcionálisan a jelszót, és esetlegesen a haladó beállításokat, amelyek alapértelmezetten rejtettek. Miután a felhasználó megadta a szükséges paramétereket, az `Embed` gomb elérhetővé válik. A felhasználó számára a visszajelzések az ablak alján látható információs kártya, és a folyamatjelző sáv segítségével valósulnak meg, továbbá az adatrejtési kapacitást a hordozófájl alatt látható üzenet tünteti fel.

Az adat visszafejtési felület elrendezése hasonló az adatrejtő felülethez, ott viszont természetesen már csak a stego-fájlt, és a hozzá tartozó jelszót kell megadnia a felhasználónak, a szoftverspecifikációban leírtak szerint.

A 4. mellékletben látható, hogy a jelszó mező értéke is adatkötéssel kerül átadásra. Eredetileg a jelszavak kezelése a `SecureString` beépített osztállyal történt volna, így elkerülve a jelszó többszöri másolását, viszont a WPF-UI jelszó mezője sajnos nem támogatta ezt a lehetőséget. (A Microsoft Learn dokumentációja szerint a `SecureString` már egyébként sem ajánlott, viszont továbbra is nagyobb biztonságot szolgáltat, mintha egyszerű `string`-et használnánk [37].)

Annak érdekében, hogy a program felülete reszponzív maradjon az adatrejtés és adat visszafejtés folyamata alatt, a feladatok aszinkron módon, megszakíthatóan hajtódnak végre a háttérben. Ezt a szteganográfiai motor aszinkron támogatása is elősegíti. Hasonlóan a parancssoros alkalmazáshoz, a rejtendő fájlok egy TAR archívum formájában kerülnek átadásra a motornak, és ahogy korábban utaltam rá, a motor ugyan azokkal a paraméterekkel működik, mint a parancssoros alkalmazás esetében, így a két alkalmazás kompatibilis kimeneteket eredményez.

4.3.2. A stego-fájl minőségbeli változásainak visszajelzése

A vizsgált szoftverek tekintetében nem volt fellelhető olyan funkcionalitás, amely visszajelezte volna a felhasználó számára, hogy az elkészült fájlban mennyire detektálhatóak a változások, azaz mennyire van biztonságban az adat. Erre tettem próbát az elkészült kép PSNR[36] értékének kiszámításával. Bár a PSNR nem a legkiválóbb mérték a látható változások meghatározására [37], gyakran alkalmazott szteganográfiai algoritmust ismertető publikációk körében az algoritmus hatékonyságának értékeléséhez [38]–[43], és triviális implementálni. A PSNR értékének számítása az alábbi egyenlet szerint valósult meg.

$$PSNR\ [dB] = 10 \log_{10} \left(\frac{MAX}{MSE} \right)$$

3. egyenlet: A PSNR kiszámítása [44]

ahol: $PSNR$ – a csúcs jel-zaj arány decibelben
MAX – a maximum lehetséges négyzetes hiba
MSE – az átlagos négyzetes hiba

Az MSE meghatározása az alkalmazási területtől függ. Esetünkben a legegyszerűbb, ha az sRGB színtartományra, mint euklideszi térre tekintünk, és a színek közötti távolság alapján definiáljuk [45, o. 63], [46], [44]:

$$MSE = \frac{\sum_{w,h}[(R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2]}{w \cdot h}$$

4. egyenlet: Az MSE kiszámítása

ahol: w – a kép szélessége
h – a kép magassága
R, G, B – az egyes szín csatornák értékei

Ennek következtében $MAX = 255^2 + 255^2 + 255^2$ a színtartomány 8 bites bitmélységből adódóan. Miután definiáltuk a minőségbeli változás mértékét, a következő lépés az értékek jelentésének behatárolása (mivel a felhasználó számára semmitmondó, ha csak kiírjuk a PSNR értékét). A modern titkosítási eljárásoknak megvan az a kívánatos

[36] Peak Signal-to-Noise Ratio

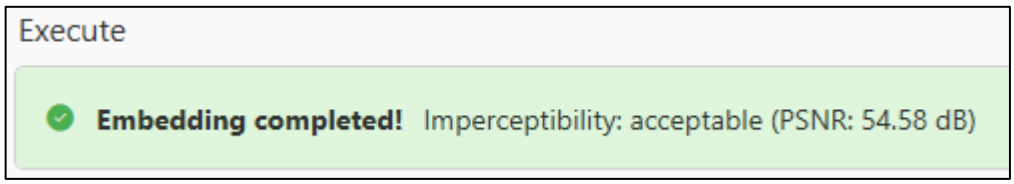
tulajdonsága, hogy a kimentüket nem lehet megkülönböztetni a véletlenszerű zajtól. Ebből kifolyólag titkosított adat rejtése esetében átlagosan 50% az esélye, hogy a kép egy adott LSB értékét módosítanunk kell. Ha az utolsó két LSB értéket írjuk felül, valamint az adott kép teljes kapacitását felhasználjuk, úgy tehát $PSNR_{átl.} = 10\log_{10}(\frac{MAX}{1.5^2+1.5^2+1.5^2}) \approx 44.61\ dB$ várható. (Két biten a maximálisan reprezentálható távolság 3, és mivel 50% a bitek változásának az esélye, így 1.5 lesz az átlagos különbség csatornánként.) Természetesen a legtöbb kép kapacitása nem lesz teljesen kihasználva, ennek megfelelően a $45\ dB$ az értékelési tartomány abszolút minimumának tekinthető.

Ebből a gondolatmenetből kiindulva a következő minőségi tartományokat határoztam meg, $6\ dB$ lépésekkel:

1. táblázat: A PSNR minőségi tartományok

A PSNR minőségi tartományok	
PSNR	**Minőség**
> 57 dB	Kiváló
> 51 dB	Elfogadható
≤ 51 dB	Gyenge

Így a program az alább látható módon jelzi a felhasználó számára az adat észlelhetőségi szintjét az adatrejtés végeztével.



22. ábra: Az adatrejtés minőségének visszajelzése (saját képernyőkép)

4.4. A projekt értékelése

A projekt során több ezer sor kód, 130 teszteset, és két alkalmazás készült el, több, mint 600 Git commit eredményeként. Tekintsük át tehát a végeredményt összesítve. Az egyszerűség érdekében a két program egyként lesz értékelve a továbbiakban, tekintve, hogy alap funkcionalitást tekintve azonosak.

4.4.1. A követelmények validálása, és összevetés a konkurenciával

2. táblázat: A projekt értékelése

Értékelési szempont	Súly	A Stegosaurus projekt pontja
Hordozható	1	1
Titkosítás	1	3
Sztego-algoritmusok	1	1
Adatintegritás ellenőrzése	1	1
Releváns kimeneti fájl formátum	1	1
Fájl rejtés	1	1
Multiplatform	0.75	1
Bővíthetőség	0.5	1
Konzolos interfész	0.5	1
Tömörítés	0.5	1
Hibajavítás	0.25	1
Láncolás	0.25	0
UX	0.25	3
Összegzés		75%

Ahogy fent látható (2. táblázat), ha a 2.2-es alfejezetben ismertetett szempontok szerint egy gyors áttekintést követően értékeljük a projektet, kiderül, hogy 75%-os eredményt kapna (a több algoritmus támogatásának, valamint kisebb extra funkciók hiányában), ezzel második helyezést érve el az OpenPuff után. Természetesen ez az értékelési rendszer nem mutat teljes képet, és nem is ez a célja. Ennek tükrében vizsgáljuk meg részletesebben is az elkészült szoftver képességeit.

Az alkalmazott adatrejtési algoritmus egy kezdetleges megoldás, és egyben a konkurenciával szemben a program legnagyobb gyengesége. Ennek ellenére viszont eleget tesz minden adatrejtésre vonatkozó alap kritériumnak: az algoritmus nagy kapacitással rendelkezik (az értékelési rendszerben (2.2-es alfejezet) ez egy pontot jelent a háromból), és releváns kimeneti fájlformátummal dolgozik. Ezen felül a program a felépítéséből adódóan kifejezetten egyszerűen bővíthető fejlettebb algoritmusokkal, (bár ez legfőképp a

parancssoros alkalmazás esetében igaz,) még ha nem is valósult meg több adatrejtési algoritmus az első verzióban.

Információbiztonsági szempontból a program minden konkurens szoftvert egyértelműen felülmúl. Modern kriptográfiai eljárásokat alkalmaz, kriptográfiailag agilis, korszerűen lassítja a jelszó hash kiszámítását, valamint adat visszafejtés során nem csak az adat integritását, hanem annak hitelességét is ellenőrzi, ezzel lehetetlenné téve az aktív támadásokat a tárolt információval szemben. Továbbá titkosítja a metainformációkat tartalmazó fejléceket is, ezáltal a stego-kulcs feltörése esetén sem szivároghat ki semmilyen információ. Korábban nem látott funkcionalitás, hogy hibajavító kódolást is alkalmaz, ezzel hosszabb távon biztosítva a tárolt információk épségét.

A felhasználói felület az aktuális tervezési konvencióknak megfelel, letisztult, és egyszerűen kezelhető. Feladatvégzés során reszponzív marad, a folyamat állapotát és a problémákat egyértelműen kommunikálja a felhasználó felé. Ezek természetesen alap elvárások minden felhasználói felületes programtól, így ilyen szempontból a legtöbb esetben nem származik valódi előnye a vizsgált szoftverekkel szemben. Ettől eltekintve egy egyedisége, hogy becslést ad a rejtett adatok észlelhetőségére vonatkozóan. Továbbá, a szoftverspecifikációnak megfelelően jelzi a felhasználó számára a várható adatrejtési kapacitást is.

Az egyéb követelmények közül is többet teljesít: korszerű tömörítési eljárásokkal tömöríti az adatokat, multiplatform, valamint kezeli a standard bemenet és standard kimenet átirányítását parancssoros használat esetén. Bár a hordozhatóság követelménye is többnyire teljesül, hiszen nem szükséges telepíteni a programot, valamint függőségek nélkül indítható, a bináris állományok nagy mérete problémát okozhat a program terjesztése során, és ez egy jelentős hátrány a konkurenciával szemben.

Mindent összevetve a program sikeresen teljesítette a szoftverspecifikációban meghatározott legfontosabb elvárásokat, viszont megvannak a saját hátrányai is, amelyek nagyrésze kiküszöbölhető továbbfejlesztéssel.

4.4.2. Továbbfejlesztési lehetőségek

A projekt elsődleges továbbfejlesztési iránya több fájlformátum támogatása, azaz több szteganográfiai algoritmus implementálása. Ugyanis ez növelné az elkészült programok hasznosságát a legnagyobb mértékben. Emellett természetesen érdemes hozzáadni

azokat a kisebb funkcionalitásokat, amelyek nem valósultak meg a szoftverspecifikációból. Így például több nyelv támogatása, és az elrejtett fájlok végleges törlése. Ezutóbbi viszont fájlrendszer specifikus, ezért kifejezetten komplikált a megvalósítása. Egy hasonlóan problémás funkcionalitás, hogy a program képes legyen csak metainformációkat kiolvasni egy adott stego-fájlból. Ez a szteganográfiai motor jelentős áttervezését igényelné, és bár hasznos, egy áttervezést sokkal érdemesebb a stego-fájlok láncolásának megvalósítására fordítani. Ne felejtsük el, hogy a szteganográfiai motor újrahasznosítható további alkalmazások fejlesztéséhez. Ezáltal a projektbe potenciálisan bevonhatóak akár mobil platformok is, mint például a Xamarin, de egy viszonylag új technológia a .NET MAUI is, amely amennyiben kiforrja magát, érdemes lehet megfontolni. Szteganográfia mellett egyes szoftverek ajánlottak vízjelezési funkciókat is, ez is egy potenciális lehetőség a projekt jövőjét tekintve. Érdemes lehet továbbá lehetővé tenni egyszerre több stego-fájl párhuzamos visszafejtését, ezzel jelentősen felgyorsítva az adatok visszanyerésének sebességét.

Az említett új funkcionalitások hozzáadása mellett a jelenlegiek is továbbfejleszthetőek. A 4.3.2-es fejezetben utaltam rá, hogy a PSNR mellett léteznek pontosabb módszerek is az észlelhetőség mérésére, ilyen például az SSIM[37]. De már egy percepció alapú színtartományban (pl.: HSI[38] [45, o. 58]) való PSNR mérés is értékesebb eredményeket hozhat. Egy már korábban említett probléma (4.1.4-es alfejezet), hogy egy technikai limitációból adódóan a titkosítás visszafejtése során a program ideiglenesen hitelesítetlen fejléccel kell dolgozzon. Bár ez jelenleg nem jelent biztonsági kockázatot, a jövőre tekintve érdemes egy megoldást találni a titkosítási fejléc hitelesítésére annak használata előtt.

[37] Structural Similarity Index Measure
[38] Hue, Saturation, Intensity

5. Összefoglalás

Ezen szakdolgozatom célkitűzése egy felhasználóbarát, modern adatrejtő alkalmazás fejlesztése volt. A szoftverprojektet megelőzően felkutattam a konkurens alkalmazásokat, felmértem az általánosan elérhető funkciókat és hiányosságokat egy értékelési rendszeren keresztül, valamint a kiemelkedő szoftvereket közelebbről is megvizsgáltam. Az elemzés során kiderült, hogy a legfejlettebb programok is rendelkeztek olyan hiányosságokkal és problémákkal, amelyek valóban indokolták egy új alkalmazás fejlesztését. Tapasztalataim alapján meghatároztam az elkészítendő szoftver projekttel kapcsolatos elvárásokat és főbb funkcionalitásokat, majd nekiláttam a tervezésnek.

Az szoftver előzetes tervezése jelentős kutatómunkával járt, legfőképp a kriptográfia, és természetesen a szteganográfia területén. Felkutatva számos módszert és eljárást úgy gondolom sikeresen kiválasztottam a leg testhezállóbbakat, és az elérhető információk alapján a modern elvárásoknak megfelelően integráltam azokat a saját projektembe.

A gondosan kiválasztott fejlesztési módszereknek köszönhetően a fejlesztés rugalmasan ment, és a szoftverprojekt a specifikációban meghatározottak szerint készülhetett el. A fejlesztés során a legnagyobb kihívást kifejezetten a titkosítás implementálása jelentette, ugyanis az általános elvárások mellett a biztonságos működésre is kimagaslóan figyelni kellett.

A dolgozat utolsó felében az elkészült munkát a legfontosabb részletek kiemelésével mutattam be, végül értékeltem, és megvizsgáltam a továbbfejlesztésre vonatkozó lehetőségeit. Összességében az elkészült projekt egy jó alapot nyújt, és rendelkezik valódi potenciállal a konkurens szoftverekkel szemben, viszont további fejlesztést igényel, hogy kiteljesüljön.

Irodalomjegyzék

[1] H. Macit, A. Koyun, és O. Güngör, „A review and comparison of steganography techniques”, nov. 2018.

[2] „OpenStego”. https://www.openstego.com/ (elérés 2022. szeptember 27.).

[3] „openstego/RandomLSBInputStream.java at master · syvaidya/openstego”. Elérés: 2022. október 31. [Online]. Elérhető: https://github.com/syvaidya/openstego/blob/master/src/main/java/com/openstego/desktop/plugin/randlsb/RandomLSBInputStream.java#L86

[4] „openstego/StringUtil.java at master · syvaidya/openstego”. Elérés: 2022. október 31. [Online]. Elérhető: https://github.com/syvaidya/openstego/blob/master/src/main/java/com/openstego/desktop/util/StringUtil.java

[5] „openstego/OpenStegoCrypto.java at master · syvaidya/openstego”. Elérés: 2022. október 30. [Online]. Elérhető: https://github.com/syvaidya/openstego/blob/master/src/main/java/com/openstego/desktop/OpenStegoCrypto.java

[6] „Java Platform, Standard Edition Java API Reference”. https://docs.oracle.com/en/java/javase/13/docs/specs/security/standard-names.html (elérés 2022. november 4.).

[7] K. Moriarty, B. Kaliski, és A. Rusch, „PKCS #5: Password-Based Cryptography Specification Version 2.1”, Internet Engineering Task Force, Request for Comments RFC 8018, 2017. doi: 10.17487/RFC8018.

[8] „Password Storage - OWASP Cheat Sheet Series”. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (elérés 2022. szeptember 15.).

[9] S. Hetzl, „steghide/src/MCryptPP.cc at master · StefanoDeVuono/steghide”. Elérés: 2022. október 31. [Online]. Elérhető: https://github.com/StefanoDeVuono/steghide/blob/master/src/MCryptPP.cc

[10] „Steghide - manual”. https://steghide.sourceforge.net/documentation/manpage.php (elérés 2022. október 31.).

[11] S. Hetzl, „steghide/src/Selector.cc at master · StefanoDeVuono/steghide”. 2022. október 24. Elérés: 2022. október 31. [Online]. Elérhető: https://github.com/StefanoDeVuono/steghide/blob/master/src/Selector.cc

[12] „OpenPuff - Steganography & Watermarking”. https://embeddedsw.net/OpenPuff_Steganography_Home.html (elérés 2022. október 31.).

[13] „libObfuscate - Cryptography & Obfuscation”. https://embeddedsw.net/libObfuscate_Cryptography_Home.html (elérés 2022. október 31.).

[14] J. Katona, „Szoftverfejlesztési technológiák - 4. előadás”, 2021.

[15] JamesNK, „Cross-platform targeting for .NET libraries”. https://learn.microsoft.com/en-us/dotnet/standard/library-guidance/cross-platform-targeting (elérés 2022. november 3.).

[16] „Conventional Commits”, Conventional Commits. https://www.conventionalcommits.org/en/v1.0.0/ (elérés 2022. november 3.).

[17] Atlassian, „Gitflow Workflow | Atlassian Git Tutorial”, Atlassian. https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow (elérés 2022. november 3.).

[18] D. Jain, „LSB Image Steganography Using Python”, The Startup, 2021. április 8. https://medium.com/swlh/lsb-image-steganography-using-python-2bbbee2c69a2 (elérés 2022. november 4.).

[19] T. Virasztó, Titkosítás és adatrejtés: Biztonságos kommunikáció és algoritmikus adatvédelem. NetAcademia Kft, 2004.

[20] K. Thompson, „antiduh/ErrorCorrection”. 2022. április 29. Elérés: 2022. november 5. [Online]. Elérhető: https://github.com/antiduh/ErrorCorrection

[21] M. Riley és I. Richardson, „Reed-Solomon codes”. https://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html (elérés 2022. november 5.).

[22] „NSec.Cryptography Namespace | NSec”. https://nsec.rocks/docs/api/nsec.cryptography (elérés 2022. november 6.).

[23] „AeadAlgorithm Class | NSec”. https://nsec.rocks/docs/api/nsec.cryptography.aeadalgorithm#aes256gcm (elérés 2022. november 6.).

[24] „NSec – Modern cryptography for .NET Core”. https://nsec.rocks/ (elérés 2022. november 6.).

[25] C. Lundkvist, „Nonce reuse in encryption - what’s the worst that can happen?”, 2022. október 14. https://github.com/christianlundkvist/blog/blob/6a33c65138d2dd5df3aea68971fab5fc61e0e7bd/2021_01_25_nonce_reuse_in_encryption/nonce_reuse_in_encryption.md (elérés 2022. november 6.).

[26] „Password Hashing Competition”. https://www.password-hashing.net/ (elérés 2022. október 22.).

[27] A. Biryukov, D. Dinu, és D. Khovratovich, „Argon2: the memory-hard function for password hashing and other applications”, o. 18.

[28] A. Biryukov, D. Dinu, D. Khovratovich, és S. Josefsson, „Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications”, Internet Engineering Task Force, Request for Comments RFC 9106, 2021. doi: 10.17487/RFC9106.

[29] „tar (computing)”, Wikipedia. 2022. október 11. Elérés: 2022. november 7. [Online]. Elérhető: https://en.wikipedia.org/w/index.php?title=Tar_(computing)&oldid=1115489549

[30] „GNU tar 1.34: 9.4.2 The Blocking Factor of an Archive”. https://www.gnu.org/software/tar/manual/html_node/Blocking-Factor.html (elérés 2022. október 21.).

[31] S. Kitt, „Answer to »Why is tar archive so much bigger than text file, 10240 bytes?«”, Unix & Linux Stack Exchange, 2022. január 10. https://unix.stackexchange.com/a/685767 (elérés 2022. november 7.).

[32] J. Alakuijala, E. Kliuchnikov, Z. Szabadka, és L. Vandevenne, „Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms”, o. 6.

[33] J. Katona, „Szoftverfejlesztési technológiák - 8. előadás”, 2021.

[34] „Working with Pixel Buffers”. https://docs.sixlabors.com/articles/imagesharp/pixelbuffers.html (elérés 2022. november 12.).

[35] Gilles „SO-stop being evil”, „Answer to »Can we use a Cryptographic hash function to generate infinite random numbers?«”, Cryptography Stack Exchange, 2019. december 12. https://crypto.stackexchange.com/a/76389 (elérés 2022. október 16.).

[36] E. B. Barker és J. M. Kelsey, „Recommendation for Random Number Generation Using Deterministic Random Bit Generators”, National Institute of Standards and Technology, NIST SP 800-90Ar1, jún. 2015. doi: 10.6028/NIST.SP.800-90Ar1.

[37] Z. Wang, A. C. Bovik, H. R. Sheikh, és E. P. Simoncelli, „Image Quality Assessment: From Error Visibility to Structural Similarity”, IEEE Trans. on Image Process., köt. 13, sz. 4, o. 600–612, ápr. 2004, doi: 10.1109/TIP.2003.819861.

[38] A. U. Islam és mtsai., „An improved image steganography technique based on MSB using bit differencing”, in 2016 Sixth International Conference on Innovative Computing Technology (INTECH), 2016. doi: 10.1109/INTECH.2016.7845020.

[39] K. Muhammad, M. Sajjad, I. Mehmood, S. Rho, és S. W. Baik, „A novel magic LSB substitution method (M-LSB-SM) using multi-level encryption and achromatic component of an image”, Multimed Tools Appl, köt. 75, sz. 22, o. 14867–14893, nov. 2016, doi: 10.1007/s11042-015-2671-9.

[40] A. Tauhid, M. Tasnim, S. A. Noor, N. Faruqui, és M. A. Yousuf, „A Secure Image Steganography Using Advanced Encryption Standard and Discrete Cosine Transform”, Journal of Information Security, köt. 10, sz. 3, Art. sz. 3, jún. 2019, doi: 10.4236/jis.2019.103007.

[41] O. Khalind és B. Aziz, „LSB Steganography with Improved Embedding Efficiency and Undetectability”, Computer Science & Information Technology, köt. 5, jan. 2015, doi: 10.5121/csit.2015.50110.

[42] U. A. M. E. Ali, E. Ali, M. Sohrawordi, és N. Sultan, „A LSB Based Image Steganography Using Random Pixel and Bit Selection for High Payload”, International Journal of Mathematical Sciences and Computing, köt. 7, o. 24–31, aug. 2021, doi: 10.5815/ijmsc.2021.03.03.

[43] S. Prasad és A. K. Pal, „An RGB colour image steganography scheme using overlapping block-based pixel-value differencing”, Royal Society Open Science, köt. 4, sz. 4, doi: 10.1098/rsos.161066.

[44] „Compute peak signal-to-noise ratio (PSNR) between images - Simulink”. https://www.mathworks.com/help/vision/ref/psnr.html (elérés 2022. november 13.).

[45] A. Koschan és M. A. Abidi, Digital Color Image Processing, 1. kiad. Wiley, 2008. doi: 10.1002/9780470230367.

[46] V.V.T, „Answer to »Defining the SNR or PSNR for color images (3 channel RGB files)«”, Signal Processing Stack Exchange, 2020. december 7. https://dsp.stackexchange.com/a/71852 (elérés 2022. november 13.).

Ábrajegyzék

Táblázatjegyzék

Mellékletek jegyzéke

Nyomtatott mellékletek

1. melléklet: A szteganográfiai szoftverértékelés eredménye (saját szerkesztés)

2. melléklet: A szteganográfiai motor osztálydiagramja (saját szerkesztés)

3. melléklet: A parancssoros alkalmazás osztálydiagramja (saját szerkesztés)

4. melléklet: A WPF alkalmazás osztálydiagramja (saját szerkesztés)

5. melléklet: Felhasználói segédlet a felhasználói felületes alkalmazáshoz (saját szerkesztés)

Digitális mellékletek

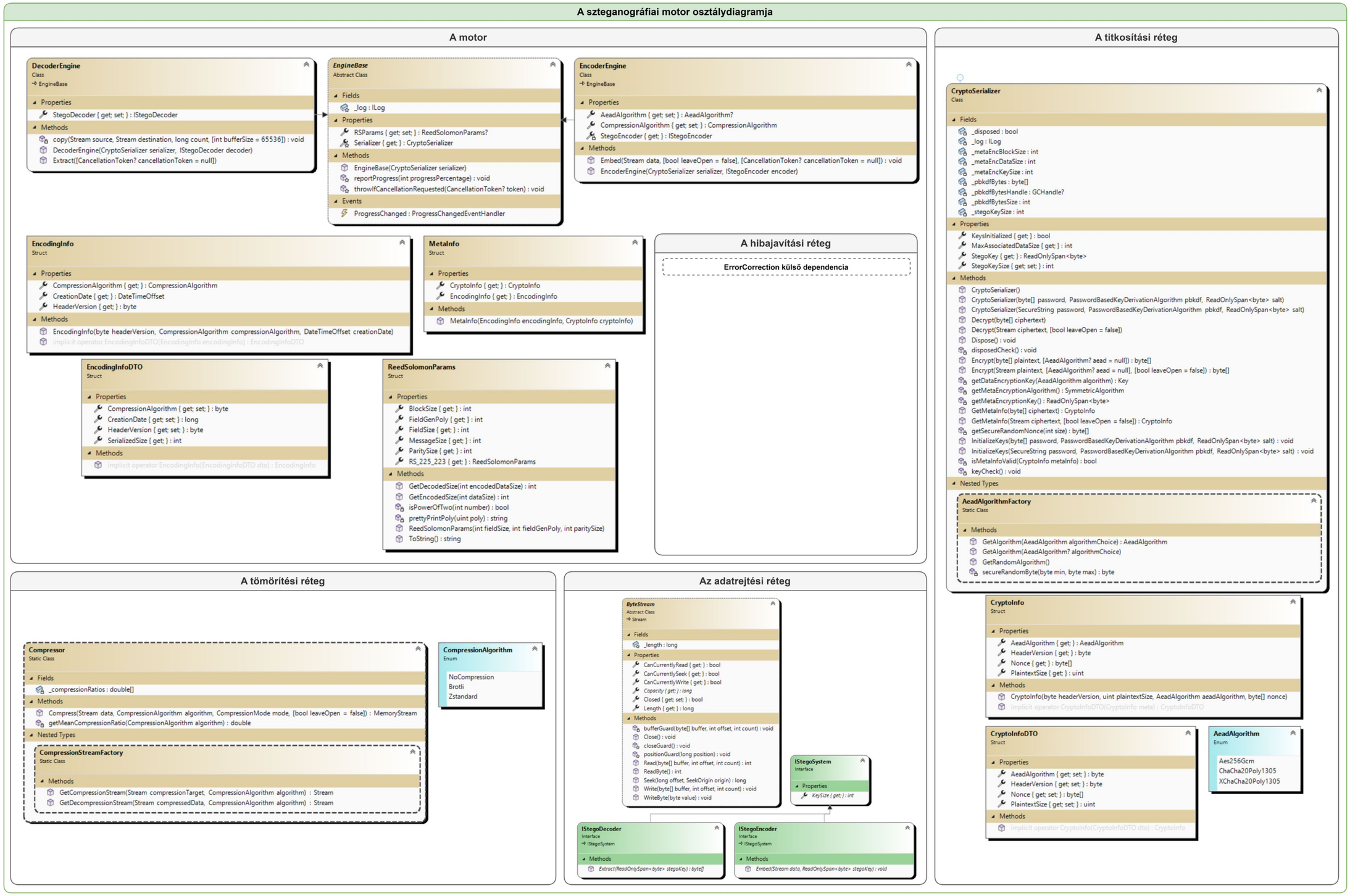
- A forráskód
- A forráskódból fordított futtatható állományok
- A forráskód Git történelme
- Az automatikusan generált fejlesztői dokumentáció
- A szteganográfiai szoftverértékelés Excel állománya

Max. pont	-	-	1	3	3	1	2	1	1	2	1	1	1	1	3
Súly	-	-	1	1	1	1	1	1	0.75	0.5	0.5	0.5	0.25	0.25	0.25
Súlyozott max. pont	15														

Suoftver neve	**Eredmény (%)**	**Eredmény**	**Hordozható**	**Titkosítás**	**Sztego-algoritmusok**	**Adatintegritás ellenőrzése**	**Releváns kimeneti fájl formátum**	**Fájl rejtés**	**Multiplatform**	**Bővíthetőség**	**Konzolos interfész**	**Tömörítés**	**Hibajavítás**	**Láncolás**	**UX**
OpenPuff	80%	12	1	3	2	1	2	1	1	1	0	0	0	1	2
Steghide	70%	10.5	1	3	1	1	1	1	1	1	1	1	0	0	1
OpenStego	70%	10.5	1	3	1	0	1	1	1	2	1	1	0	0	3
Hide'N'Send	65%	9.75	1	3	2	1	1	1	0	0	0	0	0	0	3
SilentEye	63%	9.5	0	3	1	0	1	1	1	2	1	1	0	0	3
wbStego4open	60%	9	1	3	1	0	1	1	1	1	0	0	0	0	3
Image Steganography (asztali)	58%	8.75	1	3	2	0	1	1	0	0	0	0	0	0	3
Steg	57%	8.5	1	3	1	0	1	1	1	0	0	0	0	0	3
JPHS	53%	8	1	2	1	0	1	1	1	1	1	0	0	0	1
Red JPEG XT	52%	7.75	0	2	1	1	1	1	1	0	0	1	0	0	2
GifShuffle	52%	7.75	1	2	1	0	1	1	0	1	1	1	0	0	1
Hide&Reveal	50%	7.5	1	1	1	0	1	1	1	2	0	0	0	0	3
StegoShare	48%	7.25	1	1	1	0	1	1	1	1	0	0	0	1	3
SteganoG	48%	7.25	1	3	1	0	0	1	0	0	0	1	0	0	3
Anubis	45%	6.75	1	1	0	0	2	1	1	1	0	0	0	0	2
DeepSound	43%	6.5	0	3	1	0	1	1	0	0	0	0	0	0	2
MP3Stegz	42%	6.25	1	1	1	0	1	1	0	0	0	1	0	0	3
BMPSecrets	35%	5.25	1	1	1	0	0	1	0	0	0	1	0	0	3
Hallucinate	35%	5.25	1	0	1	0	1	1	1	0	0	0	0	0	2
Jhide	33%	5	1	1	0	0	0	1	1	1	0	0	0	0	3
Xiao steganography	32%	4.75	0	2	0	1	0	1	0	0	0	0	0	0	3
SteganPEG	32%	4.75	0	1	0	0	1	1	0	1	0	1	0	0	3
rSteg	30%	4.5	1	1	0	0	1	0	1	0	0	0	0	0	3
S-Tools	30%	4.5	1	2	0	0	0	1	0	0	0	0	0	0	2
Crypture	25%	3.75	1	1	0	0	0	1	0	0	1	0	0	0	1
Trojan	25%	3.75	0	1	0	0	1	1	0	0	0	0	0	0	3
DeEgger Embedder	25%	3.75	0	0	0	0	2	1	0	0	0	0	0	1	2
SteganographX Plus	18%	2.75	1	1	0	0	0	0	0	0	0	0	0	0	3
HexaStego-BMP	17%	2.5	1	0	0	0	0	1	0	0	0	0	0	0	2
ImageHide	17%	2.5	1	0	0	0	1	0	0	0	0	0	0	0	2
SSuite Picsel	13%	2	1	0	0	0	1	0	0	0	0	0	0	0	0
QuickStego	3%	0.5	0	0	0	0	0	0	0	0	0	0	0	0	2
StegaMail	3%	0.5	0	0	0	0	0	0	0	0	0	0	0	0	2

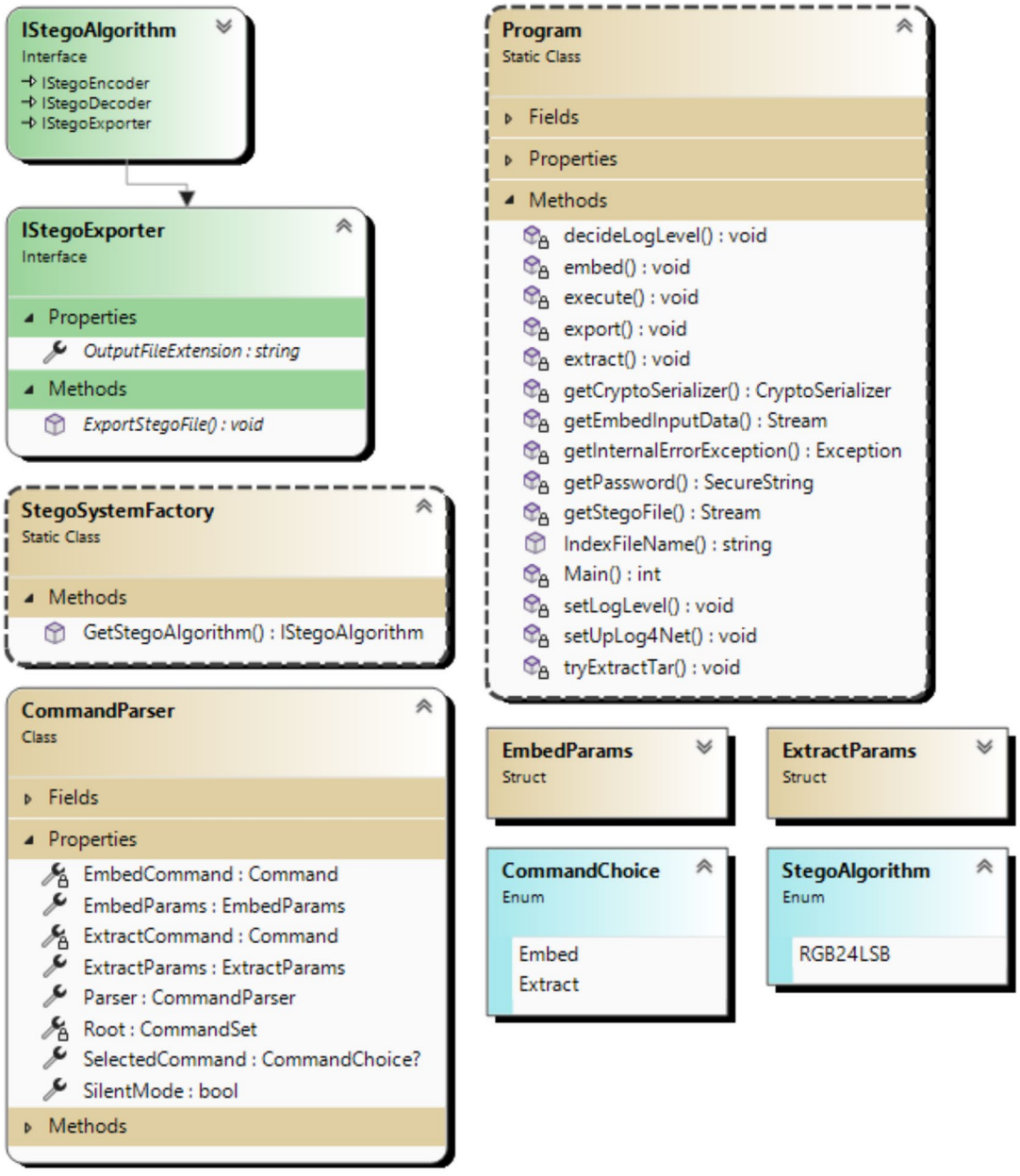
Átlag	40%
Szórás	19%

1. melléklet: A szteganográfiai szoftverértékelés eredménye (saját szerkesztés)

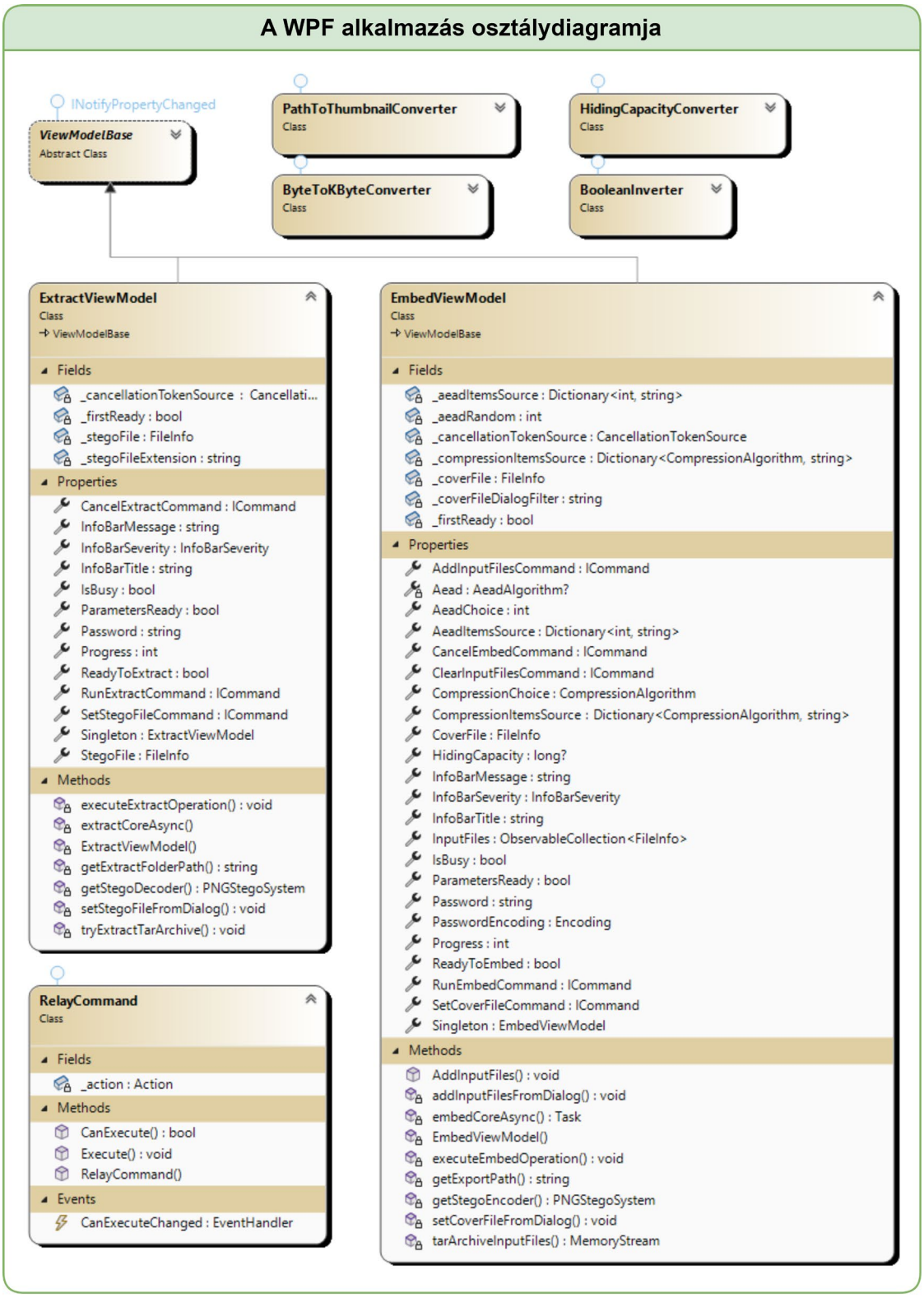


2. melléklet: A szteganográfiai motor osztálydiagramja (saját szerkesztés)

A parancssoros alkalmazás osztálydiagramja (részlet)



3. melléklet: A parancssoros alkalmazás osztálydiagramja (saját szerkesztés)

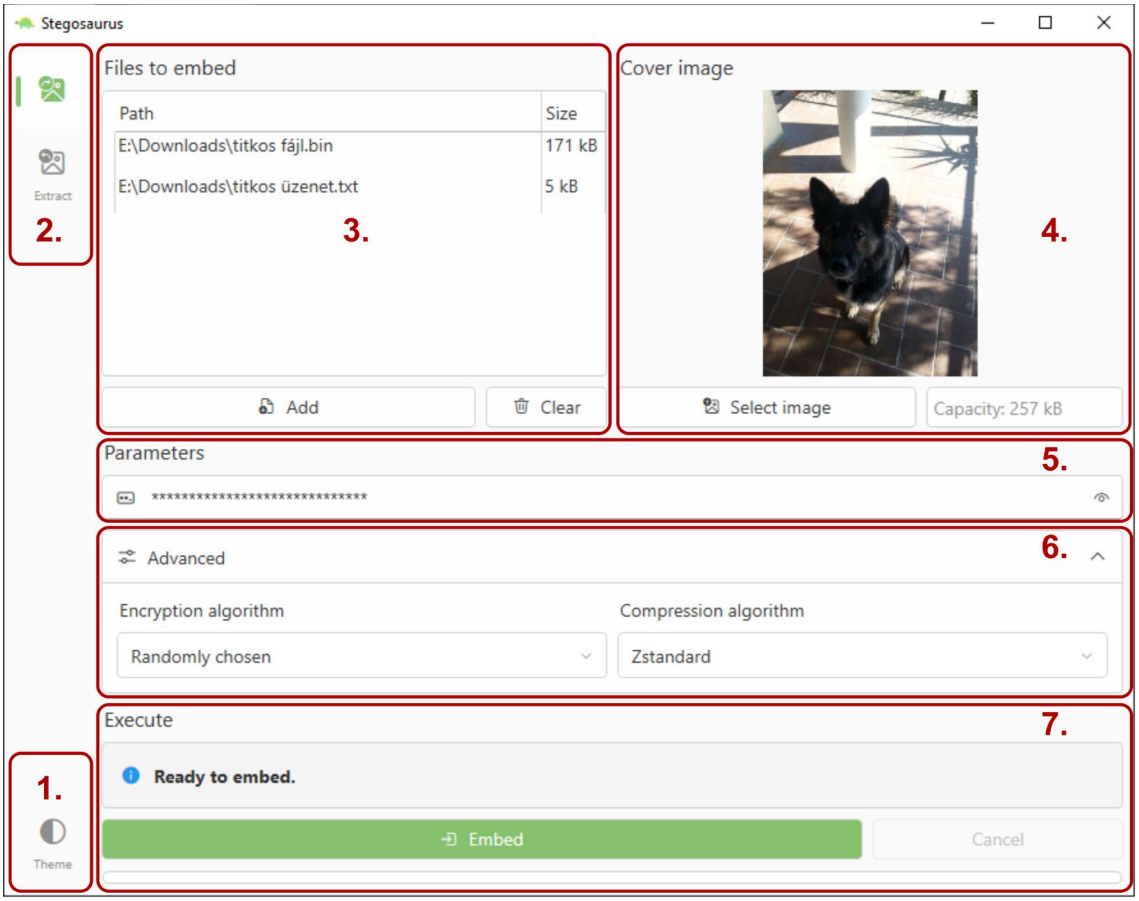


4. melléklet: A WPF alkalmazás osztálydiagramja (saját szerkesztés)

5. melléklet: Felhasználói segédlet a felhasználói felületes alkalmazáshoz (saját szerkesztés)

A Stegosaurus alkalmazás Windows 7-ig visszamenően minden Windows verzióval kompatibilis, és telepítés nélkül indítható a `Stegosaurus.WPF.exe` megnyitásával.

A program használata adatrejtéshez

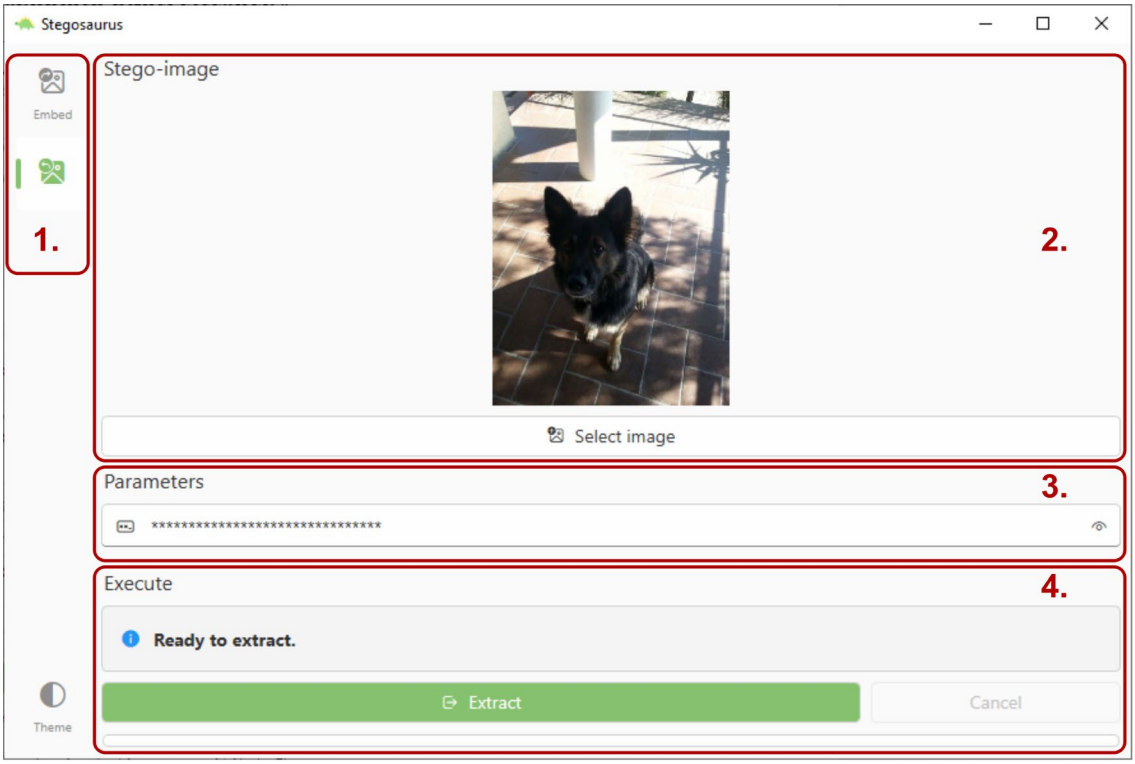


Az adatrejtés lépéseit a fenti ábra szemlélteti:

1. (Opcionális) Válasszuk ki a bal oldali navigációs sáv alján található gomb segítségével a tetszésünknek megfelelő témát.
2. Válasszuk ki a bal oldali navigációs sáv tetején az adatrejtési nézetet.
3. Tallózzuk be a rejteni kívánt fájlokat az `Add` gomb megnyomásával, vagy húzzuk őket a listába fogd és vidd módszerrel. Amennyiben hibáztunk, a `Clear` gomb megnyomásával törölhetőek a lista elemei.
4. Tallózzuk be a célfájlt, amelybe rejteni kívánjuk az adatokat. Akárcsak a 3. lépésben, itt is alkalmazható a fogd és vidd módszer.
5. (Opcionális) Adjuk meg a jelszót. A program nem határoz meg speciális feltételeket a jelszóval szemben, azonban érdemes minden esetben egyedi, és legalább 14 karakter hosszúságú jelszót alkalmazni a legnagyobb biztonság érdekében. (A jelszó maximális hossza 256 karakter.)

6. (Opcionális) Határozzuk meg az adatrejtési paramétereket. Amennyiben ugyan azt a jelszót többször tervezzük felhasználni, titkosítási algoritmusként ajánlott az `XChaCha20-Poly1305` opciót választani. A maximális tömörítési arány elérése érdekében a `Brotli`-t választhatjuk, mint tömörítési algoritmust.
7. A szükséges paraméterek megadása után az `Embed` gombra kattintva indítható az adatrejtés folyamata. A folyamat végeztével csak meg kell adnunk a felugró ablakban az exportálási útvonalat, és a `Mentés` gombra kattintva a program kimenti az elkészült fájlt.

A program használata adat visszafejtéshez



Az adat visszafejtés lépéseit a fenti ábra szemlélteti:

1. Válasszuk ki a baloldali navigációs sáv tetején az adat visszafejtési nézetet.
2. Tallózzuk be a rejtett adatokat tartalmazó fájlt a `Select image` gomb megnyomásával, vagy húzzuk a fájlt az üres felületre.
3. Adjuk meg az adatrejtés során használt jelszavunkat (amennyiben használtunk jelszót).
4. A paraméterek megadását követően az `Extract` gombra kattintva indítható az adat visszafejtés folyamata. A visszafejtés végeztével meg kell adnunk a felugró ablakban az exportálási útvonalat, és a `Mappaválasztás` gombra kattintva a program kimenti a kinyert fájlokat.