Fejlesztő dokumentáció

- ***ARCH***
  - Funkciója: absztrakciós szintet biztosít egy archív fájl kezeléséhez
    - Az **ARCH** struktúra
      - **FILE*** f tartalmazza a címét az archív fájlnak
      - **char* key** tartalmazza a XOR vagy XNOR titkosítás kulcsát. Abban az esetben, ha a pointer értéke 0, akkor nincs titkosítás
      - **bool xor** mutatja, hogy az archív fájlhoz XOR vagy XNOR titkosítás alkalmazandó, amennyiben van a fájlon titkosítás beállítva
      - **node* root** az archív faszerű könyvtárrendszerének a gyökere. Ebből ágaznak ki a levelek (fájlok) és csomópontok (könyvtárak)
      - **node* sel** pointer tárolja a fenti fának azon elememét, amelyet kiválasztottak az **archfseek(ARCH* a, char* path)** függvény segítségével
      - **size_t selpos** a fent említett kiválasztott fájlban a jelenlegi pozíciót mutatja
  - Az előbbi 2 adat segítségével az int **archfread(ARCH* a, char* buff, int n)** függvény segítségével olvashatunk egy kiválasztott fájl végéig amelyet a bool **archfeof(ARCH* a)** függvénnyel ellenőrízhetünk. Az **archfeof()** függvény a fájlhoz tartozó **node** struktúrában tárolt mérete és a **size_t selpos** alapján tudja meghatározni, hogy van-e még fennmaradó olvasandó bájt. Az n a kért bájtok száma, a char* a tömb ahova írja a kiolvasott bájtokat,és az eredménye a valóban kiolvasott bájtok száma
  - Minden más függvény amely írás/olvasás műveletet végez az archívban elmenti a FILE* jelenlegi pozícióját és a függvény végén visszaállítja, azért, hogy az **archfseek() archfreed() és archfeof()** függvények viselkedése ne sérüljön
  - az **ARCH* archparse(char* path, char* key)** függvény feladata, hogy beolvassa az archív fájlt. Az első paramétere tehát elérési út az archívhoz, a második paramétere a titkosítási kulcs, amelyre csak akkor van szükség, ha a be van állítva titkosítás. A titkosítás fajtáját a fájl bitmezőjéből olvassa majd ki. Ennek módosítására vagy megadására a későbbiekben elmagyarázott függvényeknél van lehetőség
  - Az archív fájl első bájtja **bitmező**. Az LSB bit, ha 1, akkor titkosítva van az állomány. Amennyiben van titkosítás, akkor 2-es helyiértékű bit, ha 1, akkor XOR titkosítva van az állomány, különben XNOR titkosítva. Sajnos, egyelőre a többi bit kihasználatlan

- A bitmező állítható az archsetbitfield(ARCH* a, bool crypt, bool xor) procedúrával, a paraméterek alapján összeállítja és az archív elejére írja a megfelelő bájtot
- A bitmező bitszintű olvasására az AND maszkokat alkalmazó bool getbit(char field, int i) és a void setbit(char* field, int i) függvények valóak. Ezek az ***util*** modulban találhatóak
- Amennyiben a bitmező alapján titkosítva van az állomány és nullpointert kap a kulcs paraméterébe az archparse(), vagy a kulcs mérete nincs meg legalább 1 karakter, akkor null címmel visszatér
- Az ARCH* archcreate(char* char* path, char* key, bool xor) függvén feladata, hogy létrehozzon egy megadott elérési úton egy archívot, és amennyiben titkosítva kérjük, akkor a kulcsot és a titkosítás módját is paraméterként meg kell adni
- Az archfincorp(ARCH* a, char* fpath) függvény adott útvonalon levő fájlt az archív végére ír
- Az archfincorp_e(ARCH* a, char* fpath, char* tpath) függvény az adott útvonalon levő fájlt az archívon belül másik megadott útvonalra helyezi
- Az archdincorp(ARCH* a, char* path) és archdincorp_e(ARCH* a, char* fpath, char* tpath) viselkedések pedig ugyanazt vállalják mint a fenti két függvény, csak ők mappákkal működnek, és rekurzívan felderítik a kiadott mappát: a teljes megadott almapparendszer betárolásra kerül, nem csak a közvetlen gyerekek
- Az bool archrdincorp(ARCH* a, node* n, char* path, char* apath) egy rekurzív procedúra amelyet az előző két viselkedés hív. A node paraméter megfelel a path elérési úton levő mappának, illetve az apath az archívon belüli elérési útja. Ezután felderítődnek a közvetlen gyermekei, a fájlok és mappák. A fájlokat beírja az archívba, úgy, hogy az apath végére írja a fájlnevet, illetve a fájlt is a path végéhez írt fájlnévről éri el. Az almappák pedig nyilván felderítendők még. Ezért a függvény rekurzívan hívja önmagát. Amennyiben az adott almappa nem létezik még a mostani node gyermekei között, létrehozza a mappát, és hozzáírja az apath és path végére a mappa nevét, hiszen ezekkel a paraméterekkel kell rekurzívan meghívni megint a függvényt
- A bool archforceappend(ARCH* a, char* path, char* apath, uint* fsz, size_t* pos) viselkedést alkalmazza, az előző, elvégzi az archívhoz írást. A path-en megnyitja a fáljt, az fsz-be írja a méretét, a pos-ba, hogy hová kerül az archívon belül a fájl, és az apath-ot írja bele archívon belüli útvonalként. Azért kell, hogy a pos és fsz adatokat visszaadja az archrdincorp() függvénynek, mert annak szüksége van ezekre az adatokra, hogy tudja a regisztrálni a beírt fájlt a node fában
- Az archív formátuma egyszerű: az első bájt egy bitmező, és utána végig blokkok vannak. Egy blokk első 4 bájtja előjeletlen egész szám, amely a fájl

méretét tárolja. A fájl után közvetlenül a blokk végén lezáró nullával végződő, a mappa gyökeréhez mért relatív útvonal szerepel. Utána jön a következő blokk
    - Egy ilyen blokk olvasására van az **archreadblock(ARCH* a)** viselkedés
    - A **bool archsetcrypto(ARCH* a, bool on, char* key, bool xor)** feladata, hogy elbírálja, hogy a kért titkosítási kérelem végrehajtható-e. Ellenőrzi,hogyha titkosításra kap felkérést, akkor van-e már titkosítva az állomány, vagy ha dekódolásra kap felkérést, akkor nem volt-e az archív alapból dekódolt. Ha pedig nincs ilyen baj, akkor az alábbi függvényt alkalmazza:
    - Az **archcryptthrough(ARCH* a)** procedúra feladata, hogy titkosítás bekapcsolása vagy kikapcsolása esetén végrehajtsa a XOR vagy XNOR kódolást minden egyes fájlon. Ez egy szimmetrikus kódoló/dekódoló függvény, hiszen a XOR és XNOR is szimmetrikus kriptó
    - A **bool archextract(ARCH* a, char* fpath, char* tpath)** függvény feladata, hogy adott gyökérhez mért útvonalon levő fájlt kicsomagoljon a megadott útvonalra. Mivel egy könyvtárat teljes mappaszerkezete kicsomagolandó, ezért ezt a kicsomagolást is rekurzív viselkedésnek adja tovább:
    - A **void archrextract(ARCH* a, node* n, char* p)** viselkedés kicsomagolja egy adott könyvtár közvetlen gyermekeit, és rekurzívan meghívja magát a közvetlen könyvtárgyermekeit reprezentáló **node**-okon. A fájlokat és a mappákat is létrehozza, amelyek még nem léteznek
    - Az **archrelease(ARCH* a)** viselkedés a release függvények családjába tartozik, amelyek adott struktúra heap memóriájának felszabadítására vannak. Ez egy rekurzív felszabadító függvény, hiszen egy egész faszerkezetet fel kell szabadítania
    - Az **archreader(ARCH* a, char* buff, int n, size_t* pos)** és **archwriter(ARCH* a, char* buff, int n, size_t* pos)** függvények feladata, hogy adatot olvassanak az archívból. Minden olyan magasabb szintű függvény, amit eddig felsoroltam ezeken keresztül ír vagy olvas, azért, mert automatikusan végrehajtja olvasáskor/íráskor a titkosítási műveleteket amennyiben szükséges
- *crypto*
    - Tartalmazza a **xorcrypt(char* buff, uint bsz, char* key, int ksz, size_t* pos)** és **xnorcrypt(char* buff, uint bsz, char* key, int ksz, size_t* pos)** függvényeket, amelyek feladata egy adott buffer megfelelő letitkosítása a kulccsal. Az utolsó **size_t* pos** paramétere hangsúlyos. Ennek 0 a kezdőértéke és a függvény módosítja címen keresztül. Erre azért van szüksége, mert egy nagyobb fájlra a titkosítási kulcsot úgy lehet alkalmazni, hogy ismételjük azt, maradékos osztás segítségével. A pontos illesztéshez viszont muszáj tudni, hogy az adott fájl melyik pozíciójánál járunk éppen
    - Az egyetlen viselkedés amely ezeket a függvényeket alkalmazza értelemszerűen az **archreader() és archwriter()**

- *unordered_map*
  - Az unordered_map egy hasítótábla
  - node** buckets egy pointer tömb, és mindegyik pointer egy láncolt lista elejére mutat, amelyekben azon node elemek kerülnek eltárolásra, amelyeknek ugyanaz a hash érték jutott
  - a vödrök számát az uint bsz változó tárolja
  - az uint sz a vödrökben összesen tárolt elemek számát mutatja
  - rollhash32(char* key, int b, int m) egy 32 bites hash értéket generál sztring paraméter alapján. Nagy szavakban úgy kezeli a sztringet, mint egy 256-os számrendszerbeli modulált számot, amelyben minden számjegyet (karaktert) egy adott kitevős együtthatóval szorzunk. A bázis (radix) és a modulátor paraméterként meghatározható
  - A hash értéket maradékosan osztva a vödrök számával megkapjuk azt, hogy melyik láncolt listába kerüljön elhelyezésre az adott kulcs
  - Természetesen a bázis és a modulátorok prímszámok. Olyan prímszámok lettek meghatározva, amelyek nem engedik, hogy a hash érték túlcsorduljon. Emiatt a uint hashof(char* key) függvény egy absztrakció a rollhash32() függvény felett, amely amely az előre meghatározott prímekkel hívja az előbbi függvényt
  - Az unodered_map* umcreate() függvény a hasítótáblát hozza létre, az umrelease(unordered_map* um) pedig felszabadítja. Az umreleae() függvény paramétereként egy felszabadító függvény adható meg, amely arra van, hogy a hasítótáblának a node struktúráiban tárolt void* data által hivatkozott memóriát kezelje ahogy kell
  - Az node* umput(unordered_map* um, char* key, void* dat) művelettel adott kulcsban kérhető egy generikus adat eltárolása (void* dat). A létrehozott node címét adja vissza
  - Az node* umget(unordered_map* um, char* key) művelet az umput() ellentettje, adott kulcsban levő node kikérése való. Ha nem létezik nullpointert ad vissza
  - Az void umresize(unordered_map* um) viselkedés feladata, hogy megduplázza a hashmap vödreinek a számát, és újra elossza azokat a megváltozott vödörszám és a hash értékek alapján. Ez a viselkedés akkor hívódik meg, amikor az umput() észleli, hogy a tárolt elemek száma már a vödrök számának kétszerese. Emiatt a get és put műveletek maradnak konstans probabilisztikus idejűek
  - A node** umgetbucket(unordered_map* um, char* key, uint* hash) visszaadja annak a vödör pointerének a pointerét, amelybe a paraméterként megadott kulcsnak kerülnie kell, emellett pedig kipakolja az általa generált hash értéket a hash paraméterbe

- A node* buinsert(void* dat, char* key, uint hash, node** bucket) feladata, hogy a kiszámított vödörben levő láncolt listában elhelyezze a megadott kulcsot (és a hozzá tartozó void* data elemet). A létrehozott elemet amennyiben nem létezik visszaadja, ellenkező esetben nullpointerrel tér vissza
- A node* busearch(node* bucket, char* key) feladata, hogy az umgetbucket() által meghatározott vödörben kikeresse az adott node elem címét. Amennyiben nem található, 0 címmel tér vissza
- A void umwalk(unordered_map* um, void (*iterator)(node* i)) viselkedés végigiterál az összes láncolt listán a void llwalk(node* n, void (*iterator)(node* i)) viselkedés segítségével, és mindegyikhez meghívja a paraméterül adott függvényt
- Ennél fejlettebb módja a tábla elemein való végigfutására az IT absztrakció, amelynek egy példányát adja vissza az IT* umiterator(unordered_map* um) függvény

- *IT*
  - Az absztrakció feladata, hogy egyszerűen végig lehessen iterálni egy hasítótábla elemein
  - Az IT* itcreate(node** buckets, int bsz) függvényben megadandóak a vödrök és azoknak a száma, ez elegendő a bejáráshoz
  - A node* itnext(IT* i) függvény visszaadja a hasítótábla következő node elemének a címét, ha meg nincs következő, akkor 0-át ad vissza folyamatosan
- *node*
  - Általános láncolt lista elem
  - Ugyanakkor elkerülhetetlen volt, hogy összemelegedjen szorosabban a hashmap-pel is, így nem csak következő elemet (struct node* next) és generikus adatot tárol (void* data), hanem hasítótábla kulcsot: char* key, illetve uint hash értéket is
- *Info*
  - Az info struktúra összefog mindent egy csomagban, amit egyetlen node elemnek tárolnia kell. Ez az a struktúra tehát, amelyek példányaira mutatnak a node void* data pointerei
  - bool file mutatja, hogy az adott elem fájl vagy mappa
  - Amennyiben mappa, akkor az unordered_map* um pointer tárolja a közvetlen gyerekeit
  - Amennyiben fájl, akkor a size_t addr tárolja a fájl pozícióját az archívban. És a size_t fsz tárolja a fájl méretét
- *Util*
  - Az util modul nagyon fontos segédfüggvényeket tartalmaz
  - Ide tartozik a fentebb már említett bool getbit(char field, int i) és void setbit(char* field, int i) viselkedés, amely a bitmező olvasására való

- Az archívban tárolt fájlok elérési útjai alapján a program fájlok és mappák faszerű hierarchiáját képezi le. Ennek a magja az egységnyi elem, a node, amelynek a meghatározó tulajdonságait a void* data által mutatott info objektum tartalmazza. Amennyiben ez egy mappa, akkor újabb node elemeket tartalmazó hasítótáblához jutunk. Ezen fában való közlekedés, és a fa könnyed szerkesztésére ide helyeztem el a segédfüggvényeket. A size_t nfsz(), bool nfile(node* n), size_t naddr(node* n), unordered_map* num(node* n) függvények arra valók, hogy gyorsan le lehessen kérni a void* data által mutatott info objektumból a valójában node-ot jellemző tulajdonságokat
- Ezeken kívül számos függvény ezen node fa menedzsmentjét intézik:
- node* nsub(node* n) függvény például lekéri a paraméterül kapott node ból a megadott kulccsal rendelkező gyermek node objektum címét a hozzá tartozó hasítótáblából, már amennyiben természetesen ez egy mappa. Ha nem létezik a gyermek, akkor 0 címmel tér vissza
- az ncdsub(node* n, char* key) és ncfsub(node* n, size_t addr, size_t fsz) függvények arra valók, hogy ha a paraméterül kapott node mappa, akkor ha nem létezik a paraméterül kapott kulcsú gyermeke, akkor azt létrehozza. Az ncdsub() almappát, az ncfsub() alfájlt szúr be. A sikeresen beszúrt elem címét visszaadja
- az nmakepath(node* n, char* path) függvény feladata, hogy a megadott node tól kezdve létrehozza a megadott elérési utat végig. A legutolsó útvonalkomponenshez létrehozott node címét adja eredményül
- az node* nrmakepath(node* n, char** mat, uint i, uint sz) függvény feladata, hogy mindig a teljes létrehozandó útvonal következő komponensét létrehozza a megfelelő node közvetlen gyermekeként. Az előbbi függvény ezt használja, rekurzív függvény. char** mat a feldarabolt char* path paraméter, uint i a feldolgozandó darab indexe, uint sz az összes komponens száma
- A node* nfind(node*n, uint i, uint sz, char** mat) függvény feladata, hogy kikeresse a megadott útvonalon levő node címét, a paraméterül adott node-tól kezdve. A char** mat, uint i, uint sz a feldarabolt útvonalkomponensek tömbje, a jelenlegi index a tömbben, és a tömb teljes mérete, respektíven
- uint fgetsz(FILE* f) függvény visszaadja a paraméterül adott fájl méretét
- az int mkd(const char* dir) függvény létrehozza a megadott útvonalon a mappát, amennyiben csak egy létező mappa közvetlen gyermekeként kell csak létrehozni. Amennyiben a fordítás POSIX rendszer, akkor az unistd.h fejléc mkdir() függvényét hívja, amennyiben Windows, akkor a direct.h fejlécben levő _mkdir() függvényt hívja
- az char* _strcat(char* to, char* str) függvény heap-en hozza létre az összeagglutinált sztringet, de végül sajnos semmi sem használja
- a char* combine(char* path, char* with) feladata, hogyelérési út szeparátorral összekösse a két útvonalkomponenst

- a **int* kmptbl(chr* pattern)** függvény létrehozza kmp mintaillesztő algoritmus által használt prefixtáblázatot, egy adott mintához.
- A **char* kmpfi(int* tbl, char* str, char* pat)** függvény paraméterül kap egy **kmptbl()** függvény által generált prefixtáblázatot, és visszaadja a minta legelső előfordulásának pointerét egy szövegen belül. Tulajdonképpen **strtok()-**ot helyettesíti, amely nem módosítja az eredeti sztringet. A függvény fájlok keresésére lett volna kitalálva, csak mivel a specifikációban nem szerepelt és idő hiányában voltam, végül ez a függvény sem került használatra
- **char** ppsplit(char* path, uint* splitsz)** függvény feladata, hogy feldaraboljon egy útvonalat komponenseire. Az **ssrelease(char** mat, int sz)** a megfelelő procedúra a felszabadításra
- A **char* ppnext(char* from) a ppsplit()** által használt függvény, mely visszaadja egy sztring pozíciótól a legelső útvonalszeparátor előfordulásának pointerét, különben ha nem létezik akkor 0 címmel tér vissza
- A **char* lastcomp(char* path)** függvény feladata, hogy visszaadjon egy pointert arra a karakterre az útvonalon, amely a legutolsó útvonalszeparátor után található. Amennyiben ez nem létezik, akkor ugyanazt a pointert adja vissza.
- A **char* cwd(char* buff, size_t max)** függvény feladata, hogy visszaadja a program futásának mappáját. Ez megint paltformdependsens függvény, POSIX rendszeren az **unistd.h**-ban levő **getcwd()** függvényt hívja, különben Windows-on az ennek megfelelő **direct.h** fejlécben levő **_getcwd()** függvényt hívja

- *NIT*
  - A **NIT** absztrakciót biztosít az **IT** struktúra felett, a feladata egy mappának minősülő **node** struktúra közvetlen gyerekeinek az enumerációja
  - A **NIT** is biztosítja ugyanazt a **node* nitnext(NIT* n)** operációt, amellyel a soron következő **node** gyermek címe kivehető, illetve ha nincs több gyermek akkor 0-t ad vissza
  - A **NIT** mindössze egy egyszerű wrapper, amely az **unordered_map* num(node* n)** függvénnyel kinyeri a node-hoz tartozó hasítótáblát, és az ahhoz visszaadott **IT** objektum által adott eredményeket közvetíti
- *dirmapper*
  - A dirmapper tartalmaz egy **SUBS** struktúrát amely lényegében egy vektort (dinamikus tömböt) tartalmaz fájlokhoz és mappákhoz, amelyek közvetlen gyermekei egy adott mappának. Ezt adja vissza a **SUBS* subsof(char* path)** függvény adott útvonalon levő mappához
  - **subsdirs(SUBS* s) és subsfiles(SUBS* s)** függvények mindössze getterei a **SUBS** struktúrának
  - A **subsof()** függvény platformdependsens. A POSIX implementáció viszonylag egyszerűbb, a **dirent.h** fejlécben levő struktúrák és függvények segítségével működik, míg a Win32 API-s implementáció picit rondább struktúrákkal csinálja meg teljesen ugyanazt. Valójában az egész projektben ez az egyetlen

forrás amit nem én írtam meg, hanem egy C++-os forrást alakítottam át úgy, hogy használható legyen C nyelven. Ebben nyilvánvalóan semmi nehézség nem volt, hiszen a Win32 API alapból C API, illetve az eredeti forrásban nem volt szükség sztring-nél összetettebb osztályok alkalmazására

- Egyéb megjegyzések: ahol egyébként nincs specifikálva, a bool visszatérési érték jelzi a függvények sikerességét, ez egy nagyon gyakran használt séma a projektben. A xor és xnor titkosítás szép így, de mivel a kulcs hossza nagy eséllyel nem ér fel a bemeneti fájlok méretével ezért ismételni kell, ez pedig így könnyen feltörhetővé teszi. XOR-nál például a 0 bitek folyamatosan reflektálják a kulcsnak a bitjeit, vagyis látszik a fájlban az ismétlődő kulcs. One Time Pad illetve AES titkosítást egyelőre nem támogat a program sajnos