Architektura komputerów

Laboratorium 4 w terminie 10 maja 2021

Jakub Superczyński 241381

29 maja 2021

1 Cel laboratorium

Celem laboratorium było napisanie w języku assemblera w połączeniu z C w architekturze 32-bitowej na platformę Linux programu do steganografii. Miał on działać w dwóch trybach: w trybie kodowania miał ukrywać w pliku bitmapy wiadomość tekstową podaną przez użytkownika, w trybie dekodowania miał wydobywać ukrytą wiadomość. Program miał korzystać z algorytmu najmniej znaczącego bitu.

2 Opis implementacji

2.1 Wczytywanie i zapisywanie do pliku

Ważną częścią, realizowaną w C, jest wczytywanie bitów pikseli z pliku oraz zapisywanie do pliku. Początkowo chciałem wykorzystać do tego zewnętrzną bibliotekę, okazało się jednak że ten sposób rodzi pewne problemy gdy chciałem połączyć wczytywanie z algorytmem w assemblerze. Dlatego ostatecznie ręcznie działam na plikach, korzystając tylko ze standardu C. Istotne jest, że otwieram plik bitmapy jako plik binarny, dzięki czemu program nie traktuje wczytywanych danych jako znaki ASCII, ale jako liczby (reprezentowane w pliku właśnie binarnie). Pliki .bmp mają metadane, dlatego pierwszym krokiem jest znalezienie interesującej mnie części, czyli kolorów pikseli. Jest to dosyć proste - pliki bitmap zawsze w tym samym miejscu w pliku mają zapisany adres (offset) początku bitów pikseli. Wystarczy więc użyć funkcji *fseek* aby ustawić wskaźnik w pliku na odpowiednie miejsce i odczytać 4 bajty. Znając już offset pikseli, potrzebuję jeszcze jednej informacji: ilości bajtów jaką muszę wczytać. Tę informację można obliczyć z drugiego nagłówka. Nagłówek ten może różnić się w zależności od bitmapy, jednak potrzebne mi informacje znajdują się na jego początku, który jest zawsze taki sam. Informacje te to: wysokość i szerokość obrazu w pikselach oraz liczba bitów na których zakodowany jest piksel. Mnożę więc szerokość przez liczbę bitów a następnie dzielę przez 8, by otrzymać liczbę bajtów w jednym rzędzie. Następnie mnożę tę liczbę przez wysokość by otrzymać rozmiar samego rysunku w bajtach.

Mając już te dane można łatwo wczytać bajty pikseli. Deklaruję więc tablicę danych typu char (ponieważ char ma rozmiar jednego bajta) o rozmiarze równym obliczonej liczbie bajtów, ustawiam wskaźnik w pliku na offset pikseli a następnie czytam plik korzystając z zadeklarowanej tablicy jako buforu. Dzięki temu mam zapisany ciąg bitów kodujących piksele, a adres początku tego ciągu przekazuję do funkcji assemblerowej. Zapisywanie do pliku odbywa się analogicznie. Program assemblerowy nie tworzy nowego ciągu, ale modyfikuje ten który został mu przekazany. Dlatego aby zapisać do pliku znowu ustawiam wskaźnik na offset pikseli i po prostu zapisuję zawartość tablicy. Ponieważ jest to ta sama tablica (tylko ze zmodyfikowaną zawartością), mam pewność że nadpiszę odpowiednią ilość bajtów.

2.2 Kodowanie

Program koduje tekst na dwóch ostatnich bitach każdego kolejnego bajtu. Składa się z dwóch pętli: zewnętrzna odpowiada za pobieranie kolejnych znaków do zakodowania, kończy się w momencie gdy znak jest równy znakowi nowej linii (oznacza to koniec tekstu). Pętla wewnętrzna odpowiada za pobranie bajtu pikseli oraz przeprowadzenie samej operacji kodowania, wykonuje się 4 razy (ponieważ każdy znak ma 8 bitów, a w jednej iteracji pętli wewnętrznej kodowane są 2). Operacja odbywa się następująco: za pomocą przesunięć zeruję ostatnie dwa bity bajta pikseli oraz "wyciągam" odpowiednie dwa bity znaku, ustawiając je na najmniej znaczących pozycjach i zerując pozostałe (robię to przesuwając bity o odpowiednią liczbę, zależną od numeru iteracji, w lewo, a następnie przesuwając o 6 w prawo). Po przesunięciach nadpisuję wcześniejszą wartość piksela otrzymaną liczbą. Gdy znak do zakodowania będzie równy znakowi nowej linii, pętle się kończą, ale należy zakodować jeszcze znak null. Robię to analogicznie do wszystkich poprzednich znaków.

2.3 Dekodowanie

Dekodowanie również składa się z dwóch pętli: wewnętrzna pętla przesuwa adres, do którego zapisuję dany bajt wynikowego tekstu i wykonuje się tak długo aż napotkany znak do zapisania nie będzie równy null; zeruje również zmienną pomocniczą. Pętla wewnętrzna wykonuje się 4 razy. Pobiera ona kolejny bajt piksela, zeruje wszystkie bity oprócz dwóch ostatnich. Następnie dodaje te dwa bity do zmiennej pomocniczej (która w pierwszej iteracji równa jest 0) i przesuwa jej bity w lewo, o ile nie była to ostatnia iteracja. Po 4 iteracjach mam już 8 bitów, czyli cały jeden znak, mogę więc zapisać go do pamięci.

3 Problemy podczas implementacji

Choć zadanie z początku wydawało się dosyć trudne, już po realizacji okazało się nie być takie problematyczne. Sam algorytm kodowania i dekodowania

okazał się najmniejszym problemem, choć oczywiście wymagał nieco czasu i poprawiania błędów wynikających z nieobycia w pisaniu w assemblerze. O dziwo większy problem stanowiła część odpowiadająca za operacje na plikach .bmp. Pierwszym moim pomysłem było wykorzystanie którejś z gotowych, zewnętrznych bibliotek. Niestety wszystkie które znalazłem miały poważne mankamenty: albo mocno utrudniały współpracę z assemblerem, przechowując wartości pikseli jako podwójny wskaźnik, albo działały na bardzo nielicznych formatach bitmap (a raczej jedna biblioteka miała obydwa te problemy), albo nie wczytywały żadnej bitmapy znalezionej w Internecie, twierdząc że są niepoprawne. Dopiero po wielu próbach stwierdziłem że bitmapy mają na tyle regularną strukturę, że do moich zastosowań dość łatwo można operować na nich bez pomocy jakiejkolwiek biblioteki niestandardowej. Gdy za pomocą dokumentacji udało mi się samodzielnie zaimplementować wczytywanie wartości pikseli, praca dośc mocno ruszyła do przodu i raczej nie stanowiła więcej problemów.