SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. xxxx

IMPLEMENTACIJA ALGORITMA LZP3 ZA KOMPRESIJU

Lovro Grgurić Mileusnić

Zagreb, lipanj 2022

Zahvaljujem se mentoru doc. dr. sc. Stjepanu Grošu na stručnom vodstvu pri izradi ovog završnog rada.

Sadržaj

1. Uvod 1
2. Algoritam LZP 3
   - 2.1. Motivacija 3
     - 2.1.1. Algoritam LZ77 3
     - 2.1.2. Nedostatak algoritma LZ77 u odnosu na algoritma LZP 3
   - 2.2. Primjena Markovljevih modela u kompresiji teksta 4
   - 2.3. Koder 5
   - 2.4. Dekoder 7
   - 2.5. LZP varijante 8
     - 2.5.1. LZP1 i LZP2 8
     - 2.5.2. LZP3 9
     - 2.5.3. LZP4 10
3. Siemens LZP3 12
   - 3.1. Siemens programirljivi logički kontroleri 12
   - 3.2. Format .upd datoteka 12
   - 3.3. Pojednostavljenje algoritma LZP3 14
   - 3.4. Koder 14
   - 3.5. Dekoder 17
4. Implementacija Siemens LZP3 dekompresora 19
   - 4.1. Prethodni rad 19
   - 4.2. Modul `siemens_lzp` 20
     - 4.2.1. Modul `zlib` 20
     - 4.2.2. Funkcije za dekompresiju 21
     - 4.2.3. Pomoćne funkcije 24
   - 4.3. Skripta fw_decompress.py 26
     - 4.3.1. Modul util.py 27
   - 4.4. Implementacija CRC32 27
5. Zaključak 29
6. Literatura 30

Sažetak 31

Summary 32

1. Uvod

Programirljivi logički kontroleri (engl. *programmable logic controller*, skraćeno PLC) su digitalni industrijski računalni upravljački sustavi, široko primjenjivi u automatizaciji industrijskih postrojenja. PLC-ovi primaju podatke od spojenih senzora, procesiraju ih te na temelju programiranih instrukcija donose odluke i upravljaju izlaznim uređajima. U programibilnu memoriju PLC-a mogu se pohraniti upute i implementirati slijedne, logičke i druge funkcije.

Tvrtka Siemens nalazi se među najpoznatijim svjetskim proizvođačima PLC-ova i računalnih sustava za nadzor, mjerenje i upravljanje (engl. *supervisory control and dana acquisition systems*, skraćeno SCADA) industrijskim sustavima [7]. Siemensove PLC-ove pokreće odgovarajući *firmware* koji je moguće ažurirati putem mreže ili SD kartice. Neovisno o metodi, *firmware* se prenosi komprimiran te se potom dekomprimira na PLC-u i učitava u memoriju.

Kompresija odnosno sažimanje je kodiranje sa svojstvom smanjenja broja bitova potrebnih za iskazivanje podataka. Danas je kompresija prisutna u većini digitalnih medija i kanala komunikacije zbog nepraktičnosti pohranjivanja i prijenosa podataka u njihovom izvornom obliku odnosno veličini [2]. Primjerice, sažimanjem algoritmom JPEG u sekvencijalnom modu s gubitcima (engl. *lossy sequential mode*), fotografiju koja u 24-bitnom RGB formatu ima veličinu 192 kb, možemo pohraniti u 7.4 kb [3]. Primjenom takvog sažimanja postižemo omjer kompresije 1:25.9 ≈ 1:26, odnosno fotografiju tih svojstava možemo pohraniti 26 puta više ili prenijeti 26 puta brže nego u izvornom obliku. Ovisno o vrsti podataka koji se sažimaju može se koristiti kompresija bez gubitaka ili kompresija s gubitcima (engl. *lossless* i *lossy*).

Kompresija s gubitcima primarno se koristi za sažimanje podataka gdje se greške mogu tolerirati ili nisu primjetne ljudskoj percepciji: video pozivi, slike, glazba, govor. Podatke sažete kompresijom s gubitcima nemoguće je rekonstruirati u oblik potpuno jednak izvornom, već se rekonstruiraju u podatke dovoljno slične izvornom da se zadrži njihov smisao. Rezultat ljudskoj percepciji može izgledati identičan zbog nedostatne razlučivosti ljudskog oka ili uha, ali moguće je i pojavljivanje artefakata koji se mogu ili ne moraju tolerirati ovisno o primjeni [2].

S druge strane, kompresija bez gubitaka koristi se za primjene gdje je potrebno u potpunosti očuvati informaciju koju podaci nose. Primjerice, za sažimanje teksta, datoteka točno određenih formata, bankarskih zapisa i rendgenskih snimki [2]. Dekompresija teksta sažetog kompresijom s gubitcima izazvat će pojavljivanje pogrešnih ili nedefiniranih znakova u rezultatu. Kod *markdown* jezika poput HTML-a ili XML-a, gubitci kod dekompresije uzrokovat će greške prilikom parsiranja. Isti problem javlja se i za datoteke specifičnih formata poput PDF-a. Bankarski zapisi navedeni su kao primjer osjetljivih podataka gdje gubitci nisu dopustivi, dok greške u digitalnim rendgenskim snimkama mogu utjecati na kvalitetu određivanja dijagnoze pacijenta. Slično, programski kod se skoro sigurno neće ispravno izvoditi, ako su prisutni gubitci. Stoga za kompresiju *firmwarea* programirljivih logičkih kontrolera Siemens koristi kompresiju bez gubitaka.

Postoji i druga primjena algoritama za kompresiju sa svrhom ispunjavanja svojstva tajnosti kroz princip sigurnosti prikrivanjem (engl. *security-by-obscurity*). Princip sigurnosti prikrivanjem nije u skladu sa Kerckhoffovim zakonom, prema kojem sustav mora biti siguran čak i kada su poznate sve njegove pojedinosti. Ako je poznat algoritam za kompresiju, komprimirani podatci ne mogu biti zaštićeni.

Nije poznato koristi li Siemens opskurne varijante algoritme za kompresiju kako bi namjerno otežali analiziranje svojeg *firmwarea* ili zbog jednostavnosti implementacije dekompresora za sklopovlje na kojem se dekompresija odvija. Neovisno o namjeri, kompresija predstavlja dodatnu prepreku sigurnosnim istraživačima prilikom analize *firmwarea*. Cilj ovog rada je istražiti Siemensov algoritam za kompresiju *firmwarea* te izraditi javno dostupnu implementaciju njegovog dekompresora.

Rad je strukturiran u tri poglavlja. U prvom poglavlju opisan je algoritam LZP i njegove varijante, motivacija iz koje je slijedio njegov nastanak te opis teorijskog modela na kojem se algoritam temelji. U drugom poglavlju definirani su implementacijski detalji Siemens varijante algoritma LZP3 te su opisani koder, dekoder i struktura datoteka koje sadrže komprimirani *firmware* za Siemens PLC-ove. U trećem poglavlju detaljno je opisana Python implementacija modula za dekompresiju podataka komprimiranih Siemens varijantom algoritma LZP3 te popratna skripta.

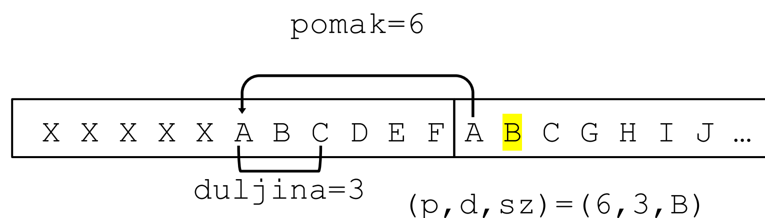
2. Algoritam LZP

Algoritam LZP je vrsta entropijskog kodiranja kojom se postiže kompresija bez gubitaka. Spada u algoritme za kompresiju koji koriste metode rječnika poput LZ77, LZ78 i LZW. Autor algoritma je Charles Bloom, koji ga je napisao kao poboljšanje algoritma LZ77 [4]. Ime algoritma proizlazi iz njegovih LZ temelja te načinu predviđanja (engl. prediction) korištenjem Markovljevih modela. Postoje 4 varijante algoritma: LZP1, LZP2, LZP3 i LZP4.

2.1. Motivacija

2.1.1. Algoritam LZ77

Algoritam LZ77 za kodiranje koristi dva posmična prozora, prozor za pretragu (engl. *search buffer*) koji obuhvaća zadnjih *n* znakova koji su već prošli kodiranje te prozor za kodiranje (engl. *look-ahead buffer*) koji gleda idućih *k* nekodiranih znakova. Svrha navedenih prozora je pronalazak ponavljanja niza znakova s početka prozora za kodiranje na bilo kojoj poziciji u prozoru za pretragu. Ukoliko je pronađeno ponavljanje, LZ77 koder na izlaz ispisuje uređenu trojku (*pomak*, *duljina*, *sljedeći znak*), gdje *pomak* predstavlja pomak od trenutne pozicije kodera odnosno dekodera unazad do prethodnog pojavljivanja niza znakova, pronađenog u prozoru za pretragu, *duljina* duljinu prethodnog pojavljivanja, a *sljedeći znak* znak kojeg treba ispisati nakon niza koji se ponovio (Slika 2.1). Posmični prozor i prozor za pretragu se potom pomiču za jedan znak i postupak se ponavlja.



Slika 2.1 Primjer kodiranja LZ77 algoritmom

2.1.2. Nedostatak algoritma LZ77 u odnosu na algoritma LZP

Nedostatak algoritma LZ77 koji algoritam LZP ispravlja je rezervacija prevelikog broja okteta prilikom kompresije za pohranjivanje pomaka do prošlog pojavljivanja nekog niza

znakova, odnosno za *pomak* [4]. Pretpostavimo da LZ77 koder koristi posmični prozor za pretragu duljine 64k okteta te da je minimalna duljina pronađenih ponovljenih nizova znakova 4 okteta, a prosječna duljina 7 okteta, odnosno znakova. Kako bi *pomak* adresirao cijeli prozor za pretragu potrebno je rezervirati 2 okteta, odnosno 16 bitova, po ponovljenom nizu znakova. Ako je prosječna duljina ponovljenog niza znakova 7 okteta, pohrana pomaka zauzima 1 oktet po 3.5 okteta izvornih podataka [4]. Prostor potreban za pohranu sljedećeg znaka i duljine ponovljenog niza znakova zanemarujemo u ovom kontekstu. LZP potpuno izbacuje *pomak* i *sljedeći znak* iz uređene trojke.

2.2. Primjena Markovljevih modela u kompresiji teksta

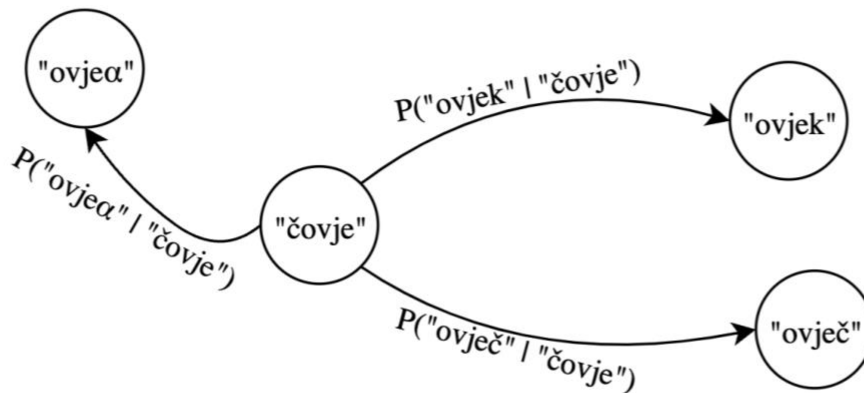
Markovljevi modeli, nazvani po ruskom matematičaru Andreju Andrejeviču Markovu, u kontekstu teorije informacije su sredstvo kojim se može prikazati postojanje ovisnosti između podataka. Niz simbola $\{x_n\}$ prati Markovljev model k-tog reda ako vrijedi:

$$P(x_n|x_{n-1},\dots,x_{n-k}) = P(x_n|x_{n-1},\dots,x_{n-k},\dots) \quad (1)$$

Odnosno, poznavanje *n* prethodnih simbola daje istu količinu informacija o idućem simbolu kao i poznavanje svih prethodnih simbola [2].

Markovljevi modeli imaju široku primjenu u kompresiji teksta, gdje na vjerojatnost pojavljivanja nekog znaka značajno utječu prethodno pojavljeni znakovi. Primjerice, jezik izvornog teksta je hrvatski te se prethodno pojavio niz znakova „čovje“. Razumno je zaključiti da je vjerojatnost pojavljivanja slova č ili k, kao u riječima „čovjek“, „čovječji“ ili „čovjekolik“, veća nego vjerojatnost pojavljivanja jednog od preostalih slova. U kontekstu kompresije teksta, Markovljevi modeli nazivaju se modelima konačnog konteksta (engl. *finite context models*). Red modela određuje broj prethodnih znakova koji se promatraju kao kontekst za predviđanje idućeg znaka ili niza znakova. Kontekst prvog reda u prethodnom primjeru je slovo e, na temelju kojeg ne dobivamo značajnu količinu informacija o idućem slovu. Povećanjem reda modela, dobivamo preciznije predviđanje.

Prethodni primjer možemo prikazati kao Markovljev lanac u kojem su stanja nizovi od *k* prethodećih znakova (Slika 2.2).



$$P(\text{"ovje}\alpha\text{"} \mid \text{"čovje"}) > P(\text{"ovje}\alpha\text{"} \mid \text{"čovje"}) \approx P(\text{"ovje}\alpha\text{"} \mid \text{"čovje"})$$

$$\alpha \in [\text{A-z}] \setminus [\text{k, č}]$$

Slika 2.2 Primjena Markovljevih modela u kompresiji teksta prikazana Markovljevim lancem

LZP koder unaprjeđuje LZ77 primjenom modela konačnog konteksta u fazi odabira nekog od prethodnog pojavljivanja niza znakova koji se trenutno nalazi u prozoru za kodiranje. LZP smanjuje skup podudarnih nizova sa skupa svih prethodno pojavljenih podudarnih nizova na samo posljednji podudarni pojavljeni niz. Drugim riječima, umjesto da koder na izlaz ispiše pomak do prethodnog pojavljivanja, LZP podrazumijeva korištenje posljednje pojave tog konteksta. Stoga je potrebno ispisati samo zastavicu podudaranja (engl. *match flag*) i njegovu duljinu, a pomak se dohvaća iz tablice raspršivanja (engl. *hash table*) koju konstruiraju koder i dekoder tijekom kodiranja odnosno dekodiranja [4].
Predviđanje modelima konačnog konteksta uz LZP koriste i drugi Lempel-Ziv algoritmi poput LZMA (engl. Lempel-Ziv Markov chain Algorithm) [5].

2.3. Koder

Poopćeni postupak kodiranja ili kompresije prikazan je pseudo kodom i vrijedi za sve varijante LZP algoritma (Kod 2.1). Za svaki oktet ulazne datoteke, LZP koder uzima prethodnih *k* okteta, gdje je *k* red modela konačnog konteksta za odabranu varijantu. Uzetih *k* okteta nazivaju se kontekst. Potom koder računa izlaz funkcije sažetka za trenutni kontekst te provjerava je li u tablici raspršivanja već upisana vrijednost pomaka za sažetak trenutnog konteksta. Ovisno o postojanju zapisa u tablici raspršivanja postupak se grana:

a. Ako ne postoji, ovo je prva pojava trenutnog konteksta. Koder na izlaz ispisuje zastavicu koja označava nepostojanje podudaranja i trenutni oktet.
b. Ako postoji, trenutni kontekst se već pojavio u procesu kodiranja. Potrebno je provjeriti postoji li podudaranje između niza okteta ulaza i niza okteta na koje pokazuje pomak u tablici raspršivanja za sažetak trenutnog konteksta. Ako se nizovi podudaraju, koder na izlaz ispisuje zastavicu i duljinu podudaranja. U suprotnom koder ispisuje zastavicu nepostojanja podudaranja i trenutni oktet.

Koder osvježava tablicu raspršivanja, odnosno upisuje indeks trenutnog okteta u tablicu raspršivanja za sažetak trenutnog konteksta. Tablicu je potrebno osvježiti kako bi u idućim prolazima koder mogao dohvatiti indeks posljednjeg pojavljivanja za neki kontekst. Postupak se ponavlja dok koder ne obradi sve oktete ulaza. Dodatno, koder na početku postupka kodiranja, prije glavne *dok* petlje, ispisuje prvih *k* okteta ulaza koji čine početni kontekst za (*k*+1)-i oktet, odnosno prvi oktet koji se obrađuje u *dok* petlji.

Ovisno o LZP varijanti, dodatno se na znakove i zastavice primjenjuje određena vrsta kompresije poput Huffmanovog ili aritmetičkog kodiranja [4], [6].

```
ulaz = procitaj_ulaznu_datoteku();
ispiši_pocetni_kontekst(ulaz);
i = duljina_pocetnog_konteksta;

dok(ulaz[i]):
       kontekst = dohvati_kontekst(ulaz, i);
       h = sažetak(kontekst);
       ako (j = htablica[h]):
              duljina = usporedi_nizove(i, j, ulaz);
              ako (duljina > 0):
                     ispiši_zastavicu_podudaranja();
                     ispiši(duljina);

              inače:
                     ispiši_zastavicu_nepodudaranja();
                     ispiši(ulaz[i]);

       inače:
              ispiši_zastavicu_nepodudaranja();
              ispiši(ulaz[i]);

       htablica[h] = i; //osvjezi tablicu rasprisvanja
       i++
```

Kod 2.1 Pseudokod poopćenog postupka kompresije

2.4. Dekoder

Analogno postupku kodiranja, postupak dekodiranja započinje ispisivanjem prvih k okteta komprimiranog ulaza koji će se koristiti kao početni kontekst. Svaki oktet podataka komprimiranog ulaza opisan je zastavicom koja mu prethodi. Ovisno o pročitanoj zastavici postupak dekodiranja se grana:

a. Ako je pročitana zastavica podudaranja, oktet kojeg opisuje je duljina podudaranja. Dekoder računa vrijednost funkcije sažetka za trenutni kontekst, odnosno za zadnjih k ispisanih znakova i dohvaća pomak iz tablice raspršivanja. Potom ispisuje niz znakova pročitane duljine koji kreće od pomaka dohvaćenog iz tablice raspršivanja.
b. Ako je pročitana zastavica znaka, dekoder ispisuje znak na izlaz.

Dekoder potom osvježava tablicu raspršivanja, upisujući novi pomak za sažetak trenutnog konteksta, kako bi u svim trenutcima imao istu tablicu kao i koder. Postupak dekodiranja prikazan je pseudokodom (Kod 2.2).

```
izlaz[];
ulaz; //ulazni tok komprimiranih podataka
ispiši_početni_kontekst(ulaz, izlaz);

dok postoje okteti na ulazu:
        z = dohvati_zastavicu(ulaz);
        oktet = dohvati_oktet(ulaz);

        ako je z == zastavica podudaranja:
                duljina = oktet;
                kontekst = dohvati_kontekst(izlaz);
                h = sažetak(kontekst);
                pomak = htablica[h];
                ispisi_niz_znakova(izlaz, duljina, pomak);

        ako je z == zastavica znaka:
                ispisi(oktet);

htablica[h]= duljina(izlaz) – 1; //osvjezi tablicu
```

Kod 2.2 Pseudokod poopćenog postupka dekompresije

2.5. LZP varijante

Algoritam LZP ima četiri varijante koje se razlikuju u 3 parametra:

- red modela konačnog konteksta
- funkcija sažetka
- zastavice podudaranja

Red modela konačnog konteksta, u daljnjem tekstu red konteksta, određuje broj prethodećih znakova, odnosno okteta, koji se uzimaju kao trenutni kontekst na koji se primjenjuje funkcija sažetka. Funkcija sažetka koristi se za konstrukciju tablice raspršivanja, u kojoj se pohranjuju pomaci do podudarnih nizova znakova, prema sažetku njihovog konteksta. Zastavicama podudaranja koder naznačuje dekoderu da je potrebno ispisati ponovljeni niz znakova, odnosno niz znakova koji je već dekodirao.

2.5.1. LZP1 i LZP2

LZP1 i LZP2 su najjednostavnije i najbrže varijante algoritma, s najnižom razinom kompresije [4]. Funkcija sažetka ovih varijanti kontekstu trećeg reda C pridružuje 12-bitni sažetak H:

$$H = ((C \gg 11) \,^\wedge\, C) \ \& \ 0xFFF$$

Duljina sažetka od 12 bitova može adresirati tablicu raspršivanja od 2^{12} zapisa.

LZP1 i LZP2 varijante algoritma koriste tri različite zastavice duljine 1 ili 2 bita:

- 00 – idući oktet je duljina podudaranja,
- 01 – idući oktet je znak, a oktet nakon njega duljina podudaranja,
- 1 – idući oktet je znak.

Primjerice, za niz znakova i duljina a b c 7 12 d, kontrolne zastavice bile bi 1|1|01|00|1, a izlaz iz kodera:

$$1\ a \mid 1\ b \mid 0\ 1\ c\ 7 \mid 00\ 12 \mid 1\ d$$

Potrebno je još kodirati znakove i duljine.

Za kodiranje znakova u LZP1 koristi se 8-bitni ASCII kod, a duljine su kodirane prema ekspandirajućem cjelobrojnom kodu promjenjive duljine (engl. *expanding variable-length-integer code*) prikazanom u tablici (Tablica 2.1)

Kod je varijabilne duljine, koja je početno širine 2 bita, a proširuje se prvo s 3 dodatna bita, nakon toga s još 5 bitova te nakon toga s 8 bitova za svako iduće proširenje. Time dobivamo kodne riječi duljine 2, 5 ili 10 + 8k, gdje je $k \in \mathbb{N}$. Najveće vrijednost u svakom proširenju, $11_{(2)}$, $111_{(2)}$, $11111_{(2)}$ i $1111\ 1111_{(2)}$, naznačuju proširenje koda, odnosno imaju značenje „slijedi još bitova“ (engl. *more bits follow*)[4]. Nema smisla kodirati duljinu 0, zato što se podudaranje duljine 0 se ne smatra podudaranjem. Stoga, kod kreće od duljine 1, kodirano $00_{(2)}$. Nakon duljine 3, kodirano $10_{(2)}$, duljinu 4 kodiramo proširivanjem koda s 3 dodatna bita, tako da je dobiveni kod za duljinu 4 $11|000_{(2)}$. Analogno proširujemo s dodatnih 5 bitova te nakon toga svaki idući put s 8 bitova.

duljina	kodna riječ
1	00
2	01
3	10
4	11|000
5	11|001
6	11|010
…	
10	11|110
11	11|111|00000
12	11|111|00001
…	
41	11|111|11110
42	11|111|11111|00000000
43	11|111|11111|00000001

Tablica 2.1 Kod za kodiranje duljine u LZP1 i LZP2

Za razliku od LZP1, u LZP2 se za kompresiju znakova koristi dvoprolazno Huffmanovo kodiranje, gdje se u prvom prolazu računa učestalost pojavljivanja znakova, nakon kojeg se konstruira tablica kodova. U drugom prolazu se vrši kodiranje konstruiranom tablicom [6].

2.5.2. LZP3

LZP3 je poboljšanje u odnosu na LZP2, koristi kontekst duljine 4 okteta i potvrđivanje konteksta (engl. *context confirmation*). Osim znakova, Huffmanovim kodom komprimiraju

se i duljine podudaranja. Funkcija raspršivanja korištena u LZP3 varijanti kontekstu četvrtog reda C pridružuje 16-bitni sažetak H:

$$H = ((C \gg 15) \ \hat{}\ C\ )\ \&\ 0xFFFF$$

Duljina sažetka od 16 bitova može adresirati tablicu raspršivanja od 2^{16} zapisa. Ovisno o implementaciji, tablica raspršivanja u jednom zapisu uz pomak može sadržavati i posljednji kontekst za koji je izračunata vrijednost sažetka tog zapisa. Ako ne sadrži posljednji kontekst, on se tijekom kompresije dohvaća iz učitanih ulaznih podataka pomoću dohvaćenog pomaka.

Posljednji kontekst je potreban za metodu potvrđivanja konteksta. Tijekom kompresije, koder će za sažetak trenutnog konteksta dohvatiti posljednji kontekst koji je imao istu vrijednost sažetka. Ako su trenutni i posljednji kontekst isti, kontekst je potvrđen i proces kompresije se nastavlja. U suprotnom, koder računa sažetak za kontekst trećeg reda, duljine 3 okteta i ponavlja postupak potvrđivanja. Ako ne uspije, koder ponavlja postupak za kontekst drugog reda te ako ponovno potvrđivanje ne uspije ispisuje znak na izlaz. Ako je potvrđivanje konteksta uspješno za kontekst drugog ili trećeg reda, proces kompresije se nastavlja provjeravanjem duljine podudaranja te ispisom duljine ili u gorem slučaju, znaka na izlaz. Za svaki neuspjeli pokušaj potvrđivanja zapis pomaka za taj sažetak se briše iz tablice raspršivanja [4]. Za kompresiju zastavica i znakova koristi se Huffmanovo kodiranje.

2.5.3. LZP4

LZP4 varijanta algoritma u odnosu na prethodne varijante koristi značajno kompleksniji proces traženja konteksta petog reda. Iako je kontekst duljine 5 okteta, funkcija sažetka se primjenjuje samo na najmanje značajna 4 okteta konteksta C, kojem pridružuje 16-bitni sažetak H :

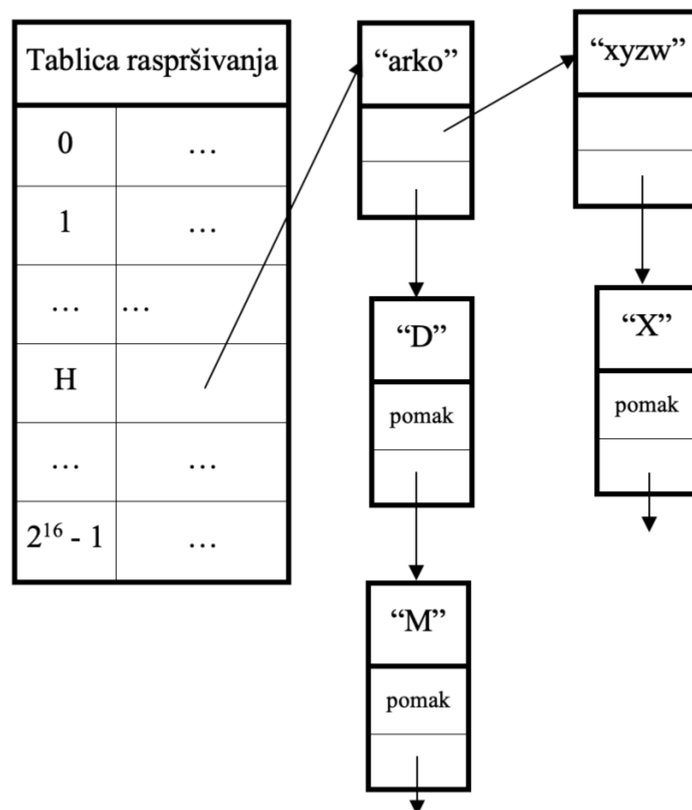
$$H = ((C[1:5] \gg 15) \ \hat{}\ C[1:5]\ )\ \&\ 0xFFFF$$

Iz duljine sažetka proizlazi da tablice raspršivanja može sadržavati 2^{16} zapisa. Svaki zapis tablice je pokazivač na ulančanu listu čvorova koji sadrže kontekste četvrtog reda s istom vrijednosti funkcije sažetka. Svaki od čvorova sadrži pokazivač na listu čvorova, od kojih

svaki sadrži najznačajniji oktet konteksta petog reda i pokazivač na posljednje pojavljivanje cjelokupnog konteksta u ulaznom spremniku.

Primjerice, konteksti petog reda „Marko“, „Darko“ i „vxyzw“, gledajući samo 4 najmanje značajna okteta, odnosno kontekste četvrtog reda, svodimo na dva niza „arko“ i „xyzw“. Ako pretpostavimo da „arko“ i „xyzw“ imaju istu vrijednost sažetka, bili bi pohranjeni u ulančanoj listi na koju pokazuje zapis u tablici raspršivanja za njihov sažetak. Najznačajniji okteti „M“ i „D“ pohranjeni su u listi na koju pokazuje čvor „arko“, a oktet „v“ u listi na koju pokazuje čvor „xyzw“ kako je prikazano (Slika 2.3).

Koder je u postupku kompresije izgradio opisanu tablicu raspršivanja te je došao u stanje u kojem je prethodni kontekst petog reda „Marko“. Kako bi dohvatio pomak za posljednje pojavljivanje konteksta „Marko“, koder prvo računa sažetak od 4 najmanje značajna okteta konteksta, „arko“. Iz tablice raspršivanja dohvaća pokazivač na ulančanu listu svih konteksta četvrtog reda koji imaju isti sažetak kao „arko“ te ju pretražuje za čvor s vrijednošću „arko“. Dohvaća pokazivač na iduću listu iz čvora „arko“ te ju pretražuje dok ne nađe čvor s vrijednosti „M“. Pronađeni čvor sadrži pokazivač na prošlo pojavljivanje cijelog konteksta „Marko“.



Slika 2.3 Primjer LZP4 tablice raspršivanja

3. Siemens LZP3

3.1. Siemens programirljivi logički kontroleri

Programirljivi logički kontroleri su digitalni industrijski računalni upravljački sustavi za automatizaciju industrijskih postrojenja. Kao predvodeća tvrtka u automatizaciji, Siemens je 2017. godine imala najveći udio na svjetskom tržištu programirljivih logičkih kontrolera [7]. Širini primjenjivosti PLC-ova, svjedoči činjenica da se koriste u mnogim industrijama, od automobilske do prometne i tekstilne [16]. Zbog kompleksnosti sustava koji upravljaju postrojenjima u navedenim industrijama mogući su mnogi vektori napada te se stvara potreba za razvojem sigurnosnih rješenja za navedene sustave. Drastičnost utjecaja napada na industrijska postrojenja vidljiva je na primjeru crva Stuxnet, uzroku zaustavljanja mnogih iranskih nuklearnih postrojenja preopterećivanjem centrifuga prisutnih u tim postrojenjima, 2010. godine [17]. Stuxnet je koristio niz ranjivosti u operacijskom sustavu Windows kako bi se proširio po mreži postrojenja, a potom ranjivosti u Siemens Step7 programskoj potpori da bi dobio pristup PLC-ovima koji upravljaju postrojenjem.

Firmware Siemensovih PLC-ova se isporučuje unutar .upd datoteka koje su podijeljene u nekoliko sekcija. U jednoj od sekcija nalazi se komprimirani *firmware*, a dekomprimiranje te sekcije odvija se nakon prijenosa na PLC. Za potrebe analize algoritma kompresije odabran je Siemens S7-1200 PLC, odnosno više inačica odgovarajućih *firmware* datoteka preuzetih sa Siemensovih službenih web stranica.

3.2. Format .upd datoteka

Datoteke u kojima se nalazi komprimirani *firmware* za S7-1200 imaju .upd ekstenziju te su veličine ≈ 13MB. Za opisivanje formata datoteke, kao primjer uzeta je inačica *firmwarea* 04.02.00, ali sve opisano vrijedi i za ostale inačice.

Datoteka je raspodijeljena u nekoliko sekcija označenim zaglavljima BG_ABL, A00000, B00000 i FW_SIG [9]. Zaglavlja se ponavljaju 2 puta, a na samom početku .upd datoteke nalazi se preambula koja sadrži informacije o inačici *firmwarea*. U heksadekadskom ispisu početka *firmwarea* vidljiva je preambula koja se proteže kroz prvih 44 okteta (Ispis 3.1).

```
OFFSET       frmwr_v04.02.00
--------------------------------------------------------------------------------
0x00000000  04 00 00 00 00 00 00 00 00 00 00 00 56 04 02 00 |............V...|
0x00000010  36 45 53 37 20 32 31 32 2D 31 41 45 34 30 2D 30 |6ES7.212-1AE40-0|
0x00000020  58 42 30 20 00 00 00 00 00 00 00 00 20 00 00 00 |XB0.............|
0x00000030  88 DC FD FF 42 47 5F 41 42 4C 84 B1 C8 00 F0 F7 |....BG_ABL......|
0x00000040  B3 CF 41 30 30 30 30 30 02 00 00 00 FF FF FF FF |..A00000........|
0x00000050  42 30 30 30 30 30 48 00 00 00 95 FE F0 FF 46 57 |B00000H.......FW|
0x00000060  5F 53 49 47 42 47 5F 41 42 4C 01 00 EF 00 00 00 |_SIGBG_ABL......|
0x00000070  00 04 36 45 53 37 20 32 31 32 2D 31 41 45 34 30 |..6ES7.212-1AE40|
0x00000080  2D 30 58 42 30 20 56 04 00 00 41 30 30 30 30 30 |-0XB0.V...A00000|
0x00000090  AE A8 00 00 00 03 5D 1B 41 53 00 00 04 00 40 00 |......].AS....@.|
```

Ispis 3.1 Preambula .upd datoteke v04.02.00 za S7-1200

Nakon preambule nalazi se prvi „opisni“ skup zaglavlja sekcija: BG_ABL, A00000, B00000 i FW_SIG. Svakom zaglavlju prethodi 8 okteta, a sadrže veličinu sekcije u oktetima i cikličnu provjeru zalihosti (engl. *cyclic redundancy check*, skraćeno CRC) vrijednost za detekciju grešaka u sadržaju sekcije. Veličina i CRC su zapisani kao dva 32-bitna cijela broja bez predznaka (engl. *unsigned integer*), tako da prva 4 okteta zauzima veličina, a idućih 4 CRC vrijednost (Ispis 3.2).

```
OFFSET       frmwr_v04.02.00
--------------------------------------------------------------------------------
0x00000000  04 00 00 00 00 00 00 00 00 00 00 00 56 04 02 00 |............V...|
0x00000010  36 45 53 37 20 32 31 32 2D 31 41 45 34 30 2D 30 |6ES7.212-1AE40-0|
0x00000020  58 42 30 20 00 00 00 00 00 00 00 00 20 00 00 00 |XB0.............|
0x00000030  88 DC FD FF 42 47 5F 41 42 4C 84 B1 C8 00 F0 F7 |....BG_ABL......|
0x00000040  B3 CF 41 30 30 30 30 30 02 00 00 00 FF FF FF FF |..A00000........|
0x00000050  42 30 30 30 30 30 48 00 00 00 95 FE F0 FF 46 57 |B00000H.......FW|
0x00000060  5F 53 49 47 42 47 5F 41 42 4C 01 00 EF 00 00 00 |_SIGBG_ABL......|
0x00000070  00 04 36 45 53 37 20 32 31 32 2D 31 41 45 34 30 |..6ES7.212-1AE40|
0x00000080  2D 30 58 42 30 20 56 04 00 00 41 30 30 30 30 30 |-0XB0.V...A00000|
0x00000090  AE A8 00 00 00 03 5D 1B 41 53 00 00 04 00 40 00 |......].AS....@.|
```

Ispis 3.2 Veličina i CRC za zaglavlje A00000

Drugo pojavljivanje svakog zaglavlja označava početak te sekcije unutar datoteke. BG_ABL sekcija je duljine nekoliko okteta te sadrži opisnik *firmwarea*. Sekcija B00000 sadrži metapodatke *firmwarea*, a sekcija FW_SIG digitalni potpis [9]. U sekciji A00000 nalazi se

komprimirani *firmware*. Za komprimiranje *firmwarea* tvrtka Siemens je razvila svoju implementaciju algoritma LZP3.

3.3. Pojednostavljenje algoritma LZP3

Siemens implementacija LZP3-a sadrži osnovne dijelove izvornog algoritma LZP3. Koristi tablicu raspršivanja i LZP3 funkciju sažetka te funkcionira na isti način kao poopćeni algoritam LZP u pogledu „predviđanja“ ponavljanja znakova na temelju modela konačnog konteksta. Međutim, Siemensova implementacija izostavlja i pojednostavljuje neke dijelove izvornog algoritam LZP3. Razumno je pretpostaviti da je bilo potrebno razviti jednostavniju implementaciju algoritma za ARM Cortex-R4 procesor koji se nalazi u S7-1200 PLC-u [9]. Dok izvorni LZP3 koristi Huffmanov kod za kodiranje i kompresiju duljina podudaranja i znakova, iz Siemens implementacije izostavljena je bilo kakva vrsta dodatne kompresije. Uz navedeno, Siemensov LZP3 grupira zastavice u oktete na način da jedan kontrolni oktet zastavica opisuje idućih 8 okteta podataka, tako da jedan bit kontrolnog okteta opisuje jedan oktet komprimiranih podataka. Algoritam koristi dvije vrste zastavica, zastavicu znaka i zastavicu podudaranja. Konkretnije, zastavica znaka je bit 0, a zastavica podudaranja bit 1. Zastavica znaka označava oktet kao nekomprimirani znak, a zastavica podudaranja oktetu kojeg označava daje značenje duljine podudaranja. Posljednje pojednostavljenje izvornog LZP3-a je ne korištenje „padajućeg“ potvrđivanja konteksta, već se tijekom kodiranja i dekodiranja potvrđuje samo kontekst četvrtog reda, bez spuštanja na kontekst trećeg i drugog reda u slučaju neuspjeha potvrđivanja.

3.4. Koder

Kao i u izvornom algoritmu LZP3, proces kodiranja započinje ispisom prvotnog konteksta četvrtog reda, odnosno duljine 4 okteta na izlaz. Siemensov LZP3 koder inicijalizira izlazni FIFO međuspremnik (engl. *buffer*) duljine 8 okteta te jedan oktet za grupiranje zastavica prije ispisa. Za svaki oktet ulazne datoteke, koder uzima prethodnih 4 okteta konteksta. Zatim koder računa vrijednost funkcije sažetka za trenutni kontekst te provjerava je li u tablici raspršivanja već postoji zapis s pomakom za sažetak trenutnog konteksta. Ovisno o postojanju zapisa u tablici raspršivanja postupak se grana:

a. Ako ne postoji, ovo je prva pojava trenutnog konteksta četvrtog reda. Koder postavlja zastavicu znaka u kontrolnom oktetu i trenutni znak upisuje u međuspremnik.

b. Ako postoji, trenutni kontekst se već pojavio u procesu kodiranja. Potrebno je provjeriti postoji li podudaranje između niza okteta ulaza i niza okteta na koje pokazuje pomak u tablici raspršivanja za sažetak trenutnog konteksta. Ako se nizovi podudaraju, koder postavlja odgovarajuću zastavicu podudaranja i u međuspremnik upisuje duljinu podudaranja. U suprotnom koder postavlja zastavicu znaka i u međuspremnik upisuje trenutni znak.

Ako je međuspremnik pun, ispisuje prvo oktet zastavica pa međuspremnik na izlaz. Koder osvježava tablicu raspršivanja, odnosno upisuje indeks trenutnog okteta u tablicu raspršivanja za sažetak trenutnog konteksta. Tablicu je potrebno osvježiti kako bi u idućim prolazima koder mogao dohvatiti indeks posljednjeg pojavljivanja za neki kontekst. Postupak se ponavlja dok koder ne obradi sve oktete ulaza. Ako je na kraju ostalo okteta u izlaznom međuspremniku, koder ispisuje pripadajući kontrolni oktet i oktete iz međuspremnika.

Siemensova LZP3 implementacija komprimira datoteke po blokovima (engl. *chunks*) veličine 64kB u nekomprimiranom obliku. Potrebno je dekoderu naznačiti veličinu svakog komprimiranog bloka. Stoga koder prije ispisivanja komprimiranog bloka ispisuje njegovu veličinu u obliku 32-bitnog cijelog broja bez predznaka. Vanjska petlja koja čita datoteku po blokovima i poziva funkciju za komprimiranje bloka prikazana je pseudokodom (Kod 3.1), a funkcija za komprimiranje jednog bloka prikazana je pseudokodom (Kod 3.2).

```
dok(istina):
        ulaz = procitaj_blok(ulazna_datoteka)
        ako je duljina(ulaz) == 0:
                prekini
        k_blok = komprimiraj_blok(ulaz)
        ispiši(duljina(k_blok))
        ispiši(k_blok)
```

Kod 3.1 Pseudokod odsječka za kompresiju datoteke po blokovima

```
ispiši_pocetni_kontekst(ulaz)
pomak = 0
izlaz[]
međuspremnik[8]
zastavice = 0x00
zindeks = 0
dok(ulaz[pomak]):
	kontekst = dohvati_kontekst(ulaz, pomak)
	h = sažetak(kontekst)
	ako (stari_pomak = htablica[h]):
		duljina=usporedi_nizove(pomak, stari_pomak, ulaz)
		ako (duljina > 0):
			upiši_zastavicu_podudaranja(zastavice, zindeks)
			međuspremnik[zindeks]=duljina
		inače:
			upiši_zastavicu_nepodudaranja(zastavice,zindeks)
			međuspremnik[zindeks]=ulaz[pomak]
	inače:
		upiši_zastavicu_nepodudaranja(zastavice,zindeks)
		međuspremnik[zindeks]=ulaz[pomak]

	htablica[h] = pomak //osvjezi tablicu rasprisvanja
	pomak++
	zindeks++
	ako nije zindeks < 8:
		ispisi(zastavice, izlaz)
		ispisi(međuspremnik, izlaz)
		zindeks = 0
ako zindeks > 0:
	ispisi(zastavice, izlaz)
	ispisi(međuspremnik, izlaz)
vrati izlaz;
```

Kod 3.2 Pseudokod funkcije za kompresiju jednog bloka

3.5. Dekoder

Postupak dekodiranja ili dekompresije odvija se po blokovima stvorenim prilikom kompresije. Komprimirani blokovi su prefiksirani svojom veličinom, zapisanom kao 32-bitni cijeli broj bez predznaka. Stoga dekoder mora prvo pročitati veličinu bloka, a zatim sam blok te ga dekomprimirati. Postupak čitanja blokova iz komprimirane datoteke te pozivanja funkcije za dekompresiju bloka prikazan je presudokodom (Kod 3.3), a funkcija za dekompresiju jednog bloka pseudokodom (Kod 3.4). Blokovi nakon dekompresije mogu biti maksimalno veličine 64KB, što je određeno procesom komprimiranja.

Postupak dekomprimiranja bloka odvija se u koracima od 9 okteta. Prvi oktet je upravljački, u kojem je redom svaki bit zastavica za svaki od preostalih 8. Dekoder prolazi po bitovima zastavicama te ovisno o vrijednosti zastavice postupak se grana:

a. Ako je vrijednost i-te zastavice 0, i-ti oktet je znak i potrebno ga je ispisati.
b. Ako je vrijednost i-te zastavice 1, i-ti oktet je duljina podudaranja. Dekoder računa vrijednost sažetka za trenutni kontekst, odnosno za zadnja 4 ispisana znaka i prema izračunatom sažetku dohvaća pomak iz tablice raspršivanja. Potom ispisuje niz znakova pročitane duljine koji kreće od pomaka dohvaćenog iz tablice raspršivanja.

Dekoder potom osvježava tablicu raspršivanja, upisujući novi pomak za sažetak trenutnog konteksta, kako bi u svim trenutcima imao istu tablicu raspršivanja kao i koder. Postupak se ponavlja dok izlaz nije veličine 64kB ili dok nisu obrađeni svi okteti komprimiranog bloka.

```
dok ima okteta na ulazu:
      velicina = procitaj_velicinu(ulazna_datoteka)
      ulaz = procitaj_blok(ulazna_datoteka, velicina)
      ako je duljina(ulaz) != velicina:
            prekini
      dek_blok = dekomprimiraj_blok(ulaz)
      ispiši(dek_blok)
```

Kod 3.3 Pseudo kod odsječka za dekompresiju datoteke po blokovima

```
izlaz[];
ispiši_početni_kontekst(ulaz, izlaz);
dok postoje okteti na ulazu ili duljina(izlaz) < 64kB:
       zastavice = dohvati_upravljacki_oktet(ulaz);
       za svaki bit z u zastavice:
              oktet = dohvati_oktet(ulaz);
              ako je z == 1:
                     duljina = oktet;
                     kontekst = dohvati_kontekst(izlaz);
                     h = sažetak(kontekst);
                     pomak = htablica[h];
                     ispisi_niz_znakova(duljina, pomak, izlaz);

              ako je z == 0:
                     ispisi(oktet, izlaz);
       htablica[h]= duljina(izlaz) – 1; //osvjezi tablicu
vrati izlaz;
```

Kod 3.4 Pseudo kod funkcije za dekompresiju jednog bloka

4. Implementacija Siemens LZP3 dekompresora

Kako bi se olakšala buduća analiza različitih inačica *firmwarea* Siemens programirljivih logičkih kontrolera, implementiran je dekompresor za nestandardnu varijantu LZP3 algoritma kojom je *firmware* komprimiran. Za implementaciju dekompresora odabrana je inačica 3.10.2. programskog jezika Python. Implementacija je javno dostupna na repozitoriju [15].

4.1. Prethodni rad

Prethodno razvoju implementacije, analizirane su .upd datoteke za Siemens Step7 programirljive logičke kontrolere modela S7-1200, S7-1500 i S7-400. Analizom *firmwarea* alatima *hexdump* i *binwalk*, utvrđeno je da su .upd datoteke komprimirane. Pronađena su zaglavlja karakteristična za kompresiju bibliotekom zlib, konkretnije kompresiju zlib implementacijom algoritma DEFLATE. Pokušaji dekomprimiranja datoteke razdvojene po pronađenim zaglavljima bili su neuspješni. Za dekomprimiranje korišten je zlib Python modul. Naposlijetku je otkriveno da datoteka nije komprimirana u potpunosti, već samo sekcija datoteke u kojoj se nalazi sam *firmware*, a korišteni algoritam za kompresiju je varijacija LZP3 [8], [9].

Nakon što je utvrđen algoritam kompresije, prvo je pokušano dekomprimiranje korištenjem node.js paketa *compressjs* [10]. Međutim, zbog implementacijskih detalja Siemensovog algoritma LZP3, nije postignuta uspješna dekompresija. Dekompresor se pokreće iz naredbenog retka sa zastavicama `-t` kojom se specificira algoritam te `-d` kojom se traži dekompresija datoteke:

```
$ ./compressjs -t lzp3 -d v04.02.00_cut.upd
```

Potom je pronađena pojednostavljena implementacija LZP3 dekompresora u programskom jeziku C, koju je implementirao autor algoritma Charles Bloom. Navedena implementacija nije komprimirala znakove i duljine podudaranja, što odgovara Siemens implementaciji, ali su se implementacije razlikovale u načinu grupiranja i obliku zastavica.

S obzirom na to da su autori rada [8] tvrdili da su uspješno dekomprimirali .upd datoteke, kontaktirani su s namjerom otkrivanja dodatnih detalja o korištenoj implementaciji. Dobiven je odgovor i implementacija algoritma autora J. B. Bédrunea [11]. Korištenjem Bédruneove implementacije uspješno je dekomprimirana *firmware* sekcija .upd datoteke:

```
$ ./unpack ../firmware/v04.05.02/frmwr unpacked
Firmware successfully unpacked
```

Na temelju Bédruneove implementacije napisan je pseudokod iz kojeg je razvijena Python implementacija Siemensove varijante LZP3 dekompresora.

4.2. Modul `siemens_lzp`

Modul `siemens_lzp` sadrži funkcije za parsiranje metapodataka zaglavlja *firmwarea*, izračun sažetka i dekompresiju podataka komprimiranim Siemensovom varijantom algoritma LZP3. Sučelje modula `siemens_lzp` strukturirano je po uzoru na modul zlib iz standardne Python biblioteke [12].

4.2.1. Modul `zlib`

Python modul `zlib` sučelje je prema istoimenoj C biblioteci, u kojoj su funkcije za kompresiju i dekompresiju algoritmom DEFLATE te funkcije za provjeru integriteta dekomprimiranih podataka [13]. Modul podržava kompresiju i dekompresiju podataka funkcijama `compress` i `decompress`. Funkcija za kompresiju:

```
compress(data: bytes, level: int)
```

Podaci se funkciji predaju kao parametar `data` u obliku Python `bytes` objekta, a opcionalno je moguće specificirati razinu kompresije putem parametra `level`. Postoji 10 razina kompresije, gdje 0 označava razinu bez kompresije, 1 označava najbržu opciju i najnižu razinu kompresije, a 9 najsporiju i najvišu razinu kompresije. Ukoliko razina nije specificirana, podrazumijevana razina je 6. Analogna funkcija za dekompresiju:

```
decompress(data: bytes, wbits: int, buffer: int)
```

Kao i u slučaju funkcije za kompresiju, komprimirani podaci se predaju funkciji kao parametar `data` u obliku `bytes` objekta. Parametar `wbits`, tipa `int`, specificira veličinu prozora za pretragu. Funkcija će automatski odrediti veličinu prozora, ako se kao `wbits` parametar preda 0. Parametar `buffer` određuje početnu veličinu međuspremnika za pohranjivanje dekomprimiranih podataka tijekom dekompresije. Ukoliko je veličina međuspremnika nedovoljana, po potrebi će se realocirati dovoljno memorije.

U slučaju da je količina podataka koju je potrebno obraditi prevelika za radnu memoriju računala na kojem se kompresija ili dekompresija odvija, dostupne su funkcije `compressobj` i `decompressobj` za stvaranje `Decompress` i `Compress` objekata

koji omogućavaju obradu podataka po dijelovima. Nakon stvaranja objekta Compress, podaci se mogu komprimirati dio po dio, predajući svaki dio metodi compress. Metoda će komprimirati oktete dok je to moguće, komprimirane oktete vratiti kao bytes objekt, a preostale oktete pohraniti u interni međuspremnik objekta. Idući poziv funkcije će spojiti oktete preostale iz prethodnog poziva i nove oktete te nastaviti kompresiju. Pozivanjem metode flush kompresija završava, metoda komprimira oktete preostale u međuspremniku te ih vraća kao bytes objekt. Korisnikova je odgovornost sve vraćene oktete spojiti. Objekt Decompress koristi se na sličan način. Uzastopnim pozivanjem metode decompress podatke može se dekomprimirati dio po dio, ali je u svakom idućem pozivu potrebno predati i podatke iz članske varijable unconsumed_tail koja sadrži neobrađene oktete iz prethodnog poziva. Metodom flush dekomprimiraju se podaci preostali u internom međuspremniku objekta te se dekompresija završava. Ukoliko su nakon kraja predanih komprimiranih podataka bili prisutni nekomprimirani okteti, oni će se po završetku dekompresije nalaziti u unused_data članskoj varijabli Decompress objekta.

4.2.2. Funkcije za dekompresiju

U modulu siemens_lzp nalaze se dvije funkcije za dekompresiju, funkcije decompress i decompress_chunk. Funkcija decompress dekomprimira podatke proizvoljne veličine, dok funkcija decompress_chunk dekomprimira blok podataka te ograničava rezultat na veličinu 64kB. Veličina rezultatnog bloka odgovara detaljima Siemensove LZP3 implementacije.

Funkcija decompress_chunk prima podatke preko parametara chunk tipa bytes te ih vraća dekomprimirane.

```
decompress_chunk(chunk: bytes)->bytes
```

Funkcija započinje inicijalizacijom izlaznog međuspremnika output tipa bytes te tablice raspršivanja hash_table kao što je prikazano u kodu (Kod 4.1). Tablica raspršivanja je ugrađenog tipa dict, u kojem su ključevi sažetci, a vrijednosti indeksi prvog znaka u izlaznom međuspremniku nakon konteksta kojem sažetak pripada. S obzirom na to da je indeks idućeg znaka ekvivalentan duljini izlaznog spremnika, dohvaća se primjenom funkcije len na izlazni spremnik. Potom se čita početni kontekst, ispisuje u izlazni međuspremnik te osvježava tablica raspršivanja. Čitanje se vrši pomoćnom funkcijom _read_bytes, a izračun sažetka funkcijom lzp_hash.

```
1.  # initialise hash table dict and output buffer
2.  hash_table = {}
3.  output = b""
4.  i = 0
5.
6.  # read and output initial context
7.  i, context = _read_bytes(i, 4, chunk)
8.  output += context
9.
10. # hash intial context and update hash table
11. hsh = lzp_hash(context)
12. hash_table[hsh] = len(output)
```

Kod 4.1 Inicijalizacija tablice raspršivanja i obrada početnog konteksta

U glavnoj *dok* petlji funkcije odvija se dekomprimiranje predanih podataka, prikazano kodom (Kod 4.2 Glavna petlja za dekompresiju. Petlja se izvodi dok je veličina izlaza manja od maksimalne veličine bloka 64kB i dok je preostalo okteta na ulazu. Svaki prolaz petlje započinje čitanjem okteta zastavica u varijablu mask. Potom se pomoću generatora `_binary` iterira po bitovima okteta zastavica, za svaki bit dohvaća oktet te mu se ovisno o zastavici pridaje značenje. Ako je trenutni bit 0, oktet je znak te ga se ispisuje u izlazni spremnik. U suprotnom, ako je bit 1, oktet je duljina podudaranja. Dohvaća se indeks prvog znaka podudaranja iz tablice raspršivanja te se podudaranje ispisuje u izlazni spremnik. Ključno je da je ispis podudaranja iterativan na razini okteta, odnosno da je implementiran kao petlja koja ispisuje oktet po oktet. Ako se koristi neki od ugrađenih mehanizama poput *slicea* kako bi se dohvatilo podudaranje iz izlaznog međuspremnika, ispis ne mora biti ispravan jer postoji mogućnost da okteti koje treba ispisati još nisu prisutni u izlaznom međuspremniku. Primjerice, ako posljednji oktet izlaznog međuspremnika ima indeks 10, podudaranje kreće od indeksa 8, a duljina podudaranja je 4, *slice* bi dohvatio samo posljednja 3 okteta, od indeksa 8 do 10. Za razliku od *slicea, za* petlja iterativno dohvaća i ispisuje oktet po oktet, tako da kada stigne do četvrtog znaka podudaranja, odnosno indeksa 11, taj znak je prisutan u izlaznom međuspremniku. Nakon obrade svake zastavice, osvježava se tablica raspršivanja novim pomakom trenutnog konteksta. Prije povratka iz funkcije i vraćanja izlaznog međuspremnika, ukoliko je međuspremnik veći od 64kB, odrezuju se okteti viška kako je prikazano u kodu Kod 4.3 Ograničavanje izlaza na veličinu 64kB. Moguća je pojava okteta viška jer se veličina izlaznog spremnika provjerava kao uvjet vanjske *dok* petlje, ali ne unutar *za* petlje.

Ukoliko je nakon posljednjeg pročitanog okteta zastavica preostalo manje od 8 okteta podataka, nije jednoznačno određeno što je potrebno učiniti za preostale zastavice. Tijekom

implementacije ove funkcije odlučeno je da će se značenje zastavice zanemariti te ispisati oktet 0xFF u izlazni međuspremnik.

```
1.  # loop while output is smaller than the maximum chunk size (64KB) and there is input
    data left
2.  while len(output) < FW_CHUNK_SIZE and i < len(chunk):
3.      # read mask byte and convert to binary string representation
4.      i, mask = _read_bytes(i, 1, chunk)
5.      mask = mask[0]
6.      # for each bit in mask
7.      for b in _binary(mask):
8.          # read next octet from input
9.          i, current = _read_bytes(i, 1, chunk)
10.         # if there is no octet left, replace with 0xFF
11.         if len(current) < 1:
12.             current = b"\xFF"
13.             b = 0 # if no byte was read, set current byte to 0xFF and output it
    (padding)
14.         # get current context, length of output and current context hash
15.         context = output[-4:]
16.         loutput = len(output)
17.         hsh = lzp_hash(context)
18.         if b == 0:
19.             # current octet is a literal, output literal
20.             output += current
21.         else:
22.             # current octet is a match length, convert to integer value, and output
    match
23.             length = int.from_bytes(current, FW_ENDIANNESS)
24.             # get last occurence of context
25.             offset = hash_table[hsh]
26.             # output match
27.             for j in range(offset, offset + length):
28.                 output += output[j].to_bytes(1, FW_ENDIANNESS)
29.
30.         #update
31.         hash_table[hsh] = loutput
32.
```

Kod 4.2 Glavna petlja za dekompresiju

```
1.  # trim output to maximum chunk size
2.  if len(output) > FW_CHUNK_SIZE:
3.          output = output[:FW_CHUNK_SIZE]
```

Kod 4.3 Ograničavanje izlaza na veličinu 64kB

Funkcija `decompress` prima jedan parametar `data` tipa `bytes`, za predaju podataka proizvoljne veličine komprimiranima Siemensovim algoritmom LZP3.

`decompress(data: bytes)->Tuple[int, bytes]`

Očekivani format podataka odgovara načinu na koji su komprimirani blokovi pohranjeni unutar A00000 sekcije .upd datoteke. Svaki blok komprimiranih podataka mora biti prefiksiran s 32 bitnom cjelobrojnom konstantom bez predznaka u *little-endian* poretku okteta. Glavna *dok* petlja po podacima prolazi blok po blok, prvo čitajući 4 okteta veličine trenutnog bloka te potom čitajući sam blok (Kod 4.4). Petlja se prekida kada dođe do prvog

neuspjeha čitanja. Neuspjehom se smatra čitanje manje od 4 okteta prilikom čitanja veličine bloka i čitanje bloka manjeg od specificirane veličine. Za dekomprimiranje bloka poziva se funkcija `decompress_chunk`. Funkcija vraća `tuple` u kojem je prvi element broj pročitanih znakova iz ulaznih podataka, a drugi element sami dekomprimirani podaci.

```
1.  while True:
2.      # read compressed chunk size
3.      nread, chunk_size_bytes = _read_bytes(nread, 4, data)
4.      if len(chunk_size_bytes) < 4:
5.          break
6.      chunk_size = int.from_bytes(chunk_size_bytes, FW_ENDIANNESS)
7.      # read compressed chunk
8.      nread, chunk = _read_bytes(nread, chunk_size, data)
9.      if len(chunk) < chunk_size:
10.         break
11.     # decompress chunk
12.     output += decompress_chunk(chunk[2:])
```

Kod 4.4 Glavna petlja funkcije `decompress`

Posljednja funkcija za dekompresiju `decompress_upddata`, je omotač funkcije `decompress`.

`decompress_upddata(data: bytes)->Tuple[bool, bytes]`

Funkcija očekuje podatke formata .upd datoteke. Automatizira izvlačenje A00000 sekcije s *firmwareom* i njeno dekomprimiranje pomoću pomoćne funkcije `extract_A00000_info`. Vraća `tuple` u kojem je prvi element tipa `bool` koji označava uspjeh ili neuspjeh dekompresije. Drugi element su uspješno dekomprimirani podaci tipa `bytes`.

4.2.3. Pomoćne funkcije

Pomoćna funkcija `extract_A0000_info` očekuje podatke formata .upd datoteke. Funkcija pronalazi prva dva pojavljivanja A00000 zaglavlja unutar preambule .upd datoteke te je prikazana kodom (Kod 4.5). Prije prvog zaglavlja nalaze se metapodaci koji sadrže veličinu i CRC32 vrijednost komprimiranog *fimwarea*. Nakon drugog pojavljivanja nalazi se početak same *firmware* sekcije. Funkcija parsira metapodatke te vraća `tuple` koji sadrži redom indeks početka *firmware* sekcije, veličinu *firmwarea*, CRC32 vrijednost.

```
1.  # slice preamble from firmware
2.  fw_preamble = data[:FW_PREAMBLE_SIZE]
3.  # find indexes of the first and second occurrence of the firmware header"""
4.
5.  # find offset at which A00000 metadata starts
6.  info_index = fw_preamble.find(FW_HEADER) - 8
7.  # find offset where A00000 section starts
8.  fw_start_index = fw_preamble.find(FW_HEADER, info_index + 9) + len(FW_HEADER)
9.  if info_index == -1 or fw_start_index == -1:
10.     return -1, -1, 0
11. # read size and crc (4 bytes each, located before the first occurrence of the firmware
    header)
12. firmware_size = int.from_bytes(fw_preamble[info_index: info_index + 4], FW_ENDIANNESS)
13. firmware_crc = int.from_bytes(fw_preamble[info_index + 4: info_index + 8],
    FW_ENDIANNESS)
```

Kod 4.5 Traženje A00000 zaglavlja i parsiranje metapodataka

Funkcija `lzp_hash` je funkcija sažetka za izvorni algoritam LZP3. Očekuje kontekst u obliku `bytes` objekta duljine 4 te vraća sažetak tipa `int`.

```
lzp_hash(context: bytes)->int
```

Funkcija `_read_bytes` uzima tri parametra: početni indeks, veličinu sadržaja kojeg treba pročitati u broju okteta i međuspremnik sa sadržajem za čitanje.

```
_read_bytes(i: int, size: int, src: bytes)->Tuple[int, bytes]
```

Funkcija čita niz okteta , najviše duljine `size`, iz međuspremnika `src`, počevši od indeksa `i`. Ujedno i uvećava indeks za broj pročitanih okteta. Nakon čitanja, vraća uvećani indeks `i` i `bytes` objekt s pročitanim oktetima.

Posljednja pomoćna funkcija `_binary`, je Python funkcija generator prikazana kodom (Kod 4.6). Funkcije generatori se mogu koristiti na isti način kao iteratori u za petlji, međutim vrijednosti po kojima se iterira su generirane dinamički prema definiciji funkcije [14]. Ne koriste uobičajenu ključnu riječ `return`, već ključnu riječ `yield`. Funkcija se poziva u prvom prolazu za petlje te se izvodi do prve pojave ključne riječi `yield`, koja vraća specificiranu vrijednost i pauzira izvođenje generatora. Vrijednost koja je vraćena ključnom riječi `yield` će biti prvi element u izvođenju petlje. U svakom idućem prolazu petlje izvođenje generatora se nastavlja dok ne vrati iduću vrijednost korištenjem ključne riječi `yield`. Konkretnije, u slučaju funkcije `_binary`, generator sadrži za petlju koja iterira po bitovima predanog cijelog broja koristeći *bitwise* I (&) i posmak (>>) te koristeći `yield` vraća svaku binarnu znamenku kao cijeli broj tipa `int`. Redoslijed kojim se iterira po bitovima ovisi o parametru `ms_first` tipa `bool`.

```
1.  def _binary(n: int, ms_first=True):
2.      """Generates binary digits of passed integer, starting from the most significant
    ones.
3.      :param n: integer for which digits will be generated
4.      """
5.      if (ms_first):
6.          for i in reversed(range(8)):
7.              yield n >> i & 0x01
8.      else:
9.          for i in range(8):
10.             yield n >> i & 0x01
```

Kod 4.6 Funkcija generator _binary

4.3. Skripta fw_decompress.py

Uz modul `siemens_lzp`, napisana je popratna Python skripta fw_decompress.py. Svrha skripte je olakšati dekomprimiranje .upd datoteka iz naredbenog retka. Napisan je i pomoćni modul util.py u koji su izdvojene funkcija za parsiranje argumenata predanih iz naredbenog retka te funkcija za ispis grešaka na standardni izlaz za greške.

Skripta se pokreće iz naredbenog retka s dva argumenta, putanjom do ulazne i izlazne datoteke. Ukoliko se ne specificira jedna od putanja skripta će ispisati kratki ispis za pomoć prikazan ispisom (Ispis 4.1).

```
usage: fw_decompress.py [-h] [-k] input_file output_file
```

Ispis 4.1 Kratki ispis za pomoć pri korištenju

Potom provjerava postoje li specificirana ulazna datoteka te ima li korisnik potrebne dozvole za čitanje i pisanje. Ukoliko neka od datoteka ne postoji ili korisnik nema potrebne dozovole, skripta ispisuje odgovarajuću poruku na standardni izlaz za greške te završava sa statusom 1. U suprotnom čita sve podatke iz ulazne datoteke te ih dekomprimira koristeći `decompress_upddata` funkciju iz `siemens_lzp` modula. Ukoliko je dekompresija uspješna pohranjuje dekomprimirani *firmware* u izlaznu datoteku te poruku `Success` na standardni izlaz (Ispis 4.3). U suprotnom, ispisuje poruku `Failed` na standardni izlaz za greške i završava sa statusom 2. Zastavicom `-k` ili `--keep-output` moguće je specificirati da se u izlaznu datoteku pohrani rezultat dekompresije neovisno o uspješnosti postupka. Pokretanjem skripte sa zastavicom `-h` ili `--help` ispisuje se kratki ispis za pomoć pri korištenju (Ispis 4.2).

```
usage: fw_decompress.py [-h] [-k] input output

positional arguments:
input              compressed firmware file name
output             decompressed firmware output file name
optional arguments:
-h, --help         show this help message and exit
-k, --keep-output  keep output in case of failure
```

Ispis 4.2 Ispis za pomoć pri korištenju

```
$ ./fw_decompress.py frmwr_v04.02.00 frmwr_unpacked
Success.
```

Ispis 4.3 Primjer uspješne dekompresije

4.3.1. Modul util.py

U pomoćnom modulu util.py nalazi se funkcija `errx`, napisanu po uzoru na `errx` funkciju iz standardne biblioteke programskog jezika C te funkcija `parse_argv` za parsiranje argumenata naredbenog retka.

Funkcija `errx` prima 2 argumenta, status tipa `int` te `msg` tipa `str`. Ispisuje na standradni izlaz poruku `msg` te izlazi sa statusom `status`.

Funkcija `parse_argv` koristi `argparse` modul iz standardne biblioteke za stvaranje parsera argumenata iz naredbenog retka. U parseru su specificirana dva obvezna pozicionalna argumenta `input_file` i `output_file`, za predaju putanja do ulazne i izlazne datoteke te zastavica `-k` ili `–keep-output`, koja specificira da je potrebno ispisati izlaz dekompresije u izlaznu datoteku neovisno o uspjehu dekompresije.

4.4. Implementacija CRC32

U sklopu razvoja `siemens_lzp` modula pokušano je implementirati provjeru CRC32 vrijednosti za komprimiranu A00000 sekciju. Njena CRC32 vrijednost nalazi se u 4 okteta prije prve pojave zaglavlja A00000 u .upd datoteci. Za izračun CRC32 vrijednosti korišten je python zlib modul te alat *cksum*. Međutim, nijedna implementacija nije vraćala vrijednost koja bi odgovarala vrijednosti specificiranoj metapodacima sekcije A00000. Uz navedene, korištena je i implementacija rekonstruirana reverznim inženjerstvom dekomprimiranog *firmwarea*. Naposljetku, pomoću alata *objcopy* obrnut je redoslijed okteta komprimiranog

firmwarea, uz pretpostavku 32-bitnih memorijskih riječi, ali svejedno nije dobivena tražena vrijednost. Svi pokušaji provedeni su na .upd datotekama inačice 4.02.00 i 4.05.02.

5. Zaključak

Opskurnost algoritma kompresije nije dobar izbor sredstva za održavanje tajnosti nekih podataka. Međutim, specifičnost varijante algoritma LZP3 korištene za kompresiju *firmwarea* PLC-ova tvrtke Siemens ispostavila se kao prepreka u sigurnosnom istraživanju i analizi. Cilj rada bio je istražiti implementacijske detalje Siemensovog algoritma LZP3 i olakšati daljnja istraživanja u području sigurnosti automatiziranih industrijskih postrojenja. S tom namjerom, izrađena je javno dostupna implementacija Siemensovog LZP3 dekompresora te pripadajući Python modul. U budućem radu potrebno je istražiti formate *firmware* datoteka drugih modela PLC-ova te proširiti `siemens_lzp` modul kako bi bio kompatibilan s njima. Uz navedeno, potrebno je i istražiti te implementirati CRC32 provjeru za komprimirani *firmware*. Naposlijetku, implementacijom kompresora omogućilo bi se stvaranje modificiranih .upd datoteka s zloćudnim *firmwareom* u svrhu istraživanja mogućih vektora napada i razvoja odgovarajućih sigurnosnih zakrpa kako bi se spriječili novi potencijalni napadi.

6. Literatura

[1] Kopte, T., Pai, A. Programmable Logic Controller and Its Applications.

[2] Sayood, K., Introduction to data compression. Morgan Kaufmann, 5. izdanje, 2017.

[3] Al-Ani, M. S., & Awad, F. H., "The JPEG image compression algorithm.", International Journal of Advances in Engineering & Technology 6.3 (2013): 1055.

[4] Bloom, C. LZP: a new data compression algorithm. In Data Compression Conference (pp. 425-425). IEEE Computer Society., (1996, March).

[5] History of Lossless Data Compression Algorithms, ETHW, (2019., siječanj). Poveznica: https://ethw.org/History_of_Lossless_Data_Compression_Algorithms#LZMA; pristupljeno 29. svibanj 2022.

[6] Salomon, D. Data compression: the complete reference. Springer Science & Business Media., (2004).

[7] Dawson, T., Who Were the Leading Vendors of Industrial Controls in 2017?, Interact Analysis, (2018, studeni). Poveznica: https://www.interactanalysis.com/who-were-the-leading-vendors-of-industrial-controls-in-2017/; pristupljeno 30. svibanj 2022.

[8] Bédrune, J. B., Gazet, A., & Monjalet, F. (2015). Supervising the Supervisor: Reversing Proprietary SCADA Tech.

[9] A., Abbasi, T., Scharnowski, T., Holz, Doors of Durin: The Veiled Gate to Siemens S7 Silicon, Black Hat Europe 2019

[10] Scott, C., javascript paket *compressjs*, https://github.com/cscott/compressjs

[11] Bédrune, J.P., s7unpack, https://github.com/jibeee/s7unpack

[12] Python zlib modul, Poveznica: https://docs.python.org/3/library/zlib.html

[13] zlib, (2022., ožujak), Poveznica: https://www.zlib.net/; pristupljeno 1. lipanj 2022

[14] Generators, Python wiki, (2020., prosinac) Poveznica: https://wiki.python.org/moin/Generators; pristupljeno 3. lipanj 2022.

[15] Siemens LZP3 implementacija, Poveznica: https://gitlab.com/lgrguric/siemens_lzp3

[16] Kopte, T., & Pai, A., Programmable Logic Controller and Its Applications., International Journal of Engineering Research and General Science Volume 3, Issue 5, (2015, rujan-listopad)

[17] Kushner, D., The real story of stuxnet. *ieee Spectrum*, *50*(3), 48-53., (2013).

Sažetak

Implementacija algoritma LZP3 za kompresiju

Tvrtka Siemens *firmware* za svoje logičke kontrolere komprimira izmijenjenom varijantom algoritma LZP3. Opskurnost ove varijante predstavlja prepreku u sigurnosnom istraživanju i analizi jer je *firmware* nemoguće dekomprimirati standardnim alatima. Stoga istraživačima preostaje samo analiza pristupom crne kutije, za koji je potrebno imati pristup PLC sklopovlju na koje će se *firmware* učitati.
U ovom radu analiziran je općeniti algoritam LZP, njegove četiri izvorne varijante te teorijski model na kojem se algoritam zasniva. Opisana je specifikacija Siemensove varijante algoritma LZP3, odgovarajuća implementacija dekompresora te popratni modul za dekompresiju napisan u programskom jeziku Python.

Ključne riječi: programirljivi logički kontroleri; PLC; kompresijski algoritmi; LZP3

Summary

Implementation of the LZP3 compression algorithm

Siemens compresses firmware for its programmable logic controllers with a modified version of the LZP3 algorithm. The obscurity of this variant presents an obstacle in security research and analysis because the firmware is impossible to decompress with standard decompression tools. Therefore, researchers can only apply a black box approach to analysis, which requires access to the PLC hardware on which the firmware will be loaded.
In this paper, the general LZP algorithm, its variants and the underlying theoretical model are analysed. The specification of the Siemens variant of the LZP3 algorithm is described and a Python implementation of the decompressor is presented along with the accompanying decompression module.

Keywords: programmable logic controllers; PLC; compression algorithms; LZP3