FbHash: shema za izračun podobnosti datotek v digitalni forenziki

Timotej Knez
Fakulteta za računalništvo in informatiko
Univerza v Ljubljani

Sebastian Mežnar
Fakulteta za matematiko in fiziko
Univerza v Ljubljani

Jasmina Pegan
Fakulteta za računalništvo in informatiko
Univerza v Ljubljani

POVZETEK

Algoritmi za detekcijo podobnih datotek pomagajo digitalnim forenzikom pri obdelavi velikih količin podatkov. V delu predstavimo algoritem za detekcijo podobnih datotek `FbHash`, opisan v članku [5] in nekaj njegovih predhodnikov. Predstavimo in implementiramo tudi svojo različico algoritma `FbHash`. Implementacijo testiramo na istih množicah datotek kot avtorji članka ter predstavimo naše ugotovitve. Rezultati eksperimentov se ujemajo z rezultati v članku, saj naša implementacija algoritma doseže F-score 94%.

Kategorija in opis področja

E.3 [**Data encryption**]

Splošni izrazi

Zgoščevanje

Ključne besede

Prstni odtis datoteke, Podobnostni izvleček, Zabrisano zgoščevanje, TF-IDF, Kosinusna podobnost

1. UVOD

Živimo v obdobju shranjevanja ogromnih količin podatkov. Pri forenzičnih preiskavah se pogosto zgodi, da je pridobljenih datotek preveč za ročno pregledovanje. Digitalni forenziki se tako soočijo s problemom avtomatizacije preiskave datotek. Možna rešitev so algoritmi za detekcijo podobnih datotek (angl. *Approximate Matching algorithms*), kot so `ssdeep`, `sdhash` in `FbHash`, ki poskusijo filtrirati vnaprej znane "slabe" oziroma "dobre" datoteke. Ti algoritmi ugotavljajo delež ujemanja datotek s pomočjo (nekriptografskih) zgoščevalnih funkcij. Algoritma `ssdeep` in `sdhash` lahko preslepi aktivni napadalec, ki pametno napravi majhne spremembe na določenih mestih datoteke. Učinkovitega napada na algoritem `FbHash` pa ne poznamo [5].

1.1 Prispevek članka

V našem delu predstavimo članek [5], v katerem avtorji predstavijo zgoščevalno funkcijo `FbHash`, ki omogoča lažje filtriranje "dobrih" in "slabih" datotek pri preiskovanju.

Glavni prispevki raziskave v [5] so:

- Predstavijo shemo za približno ujemanje, ki je odporno proti aktivnemu napadalcu.
- Predstavijo algoritma za zgoščevanje stisnjenih in nestisnjenih datotek, ki temeljita na shemi TF-IDF [12].
- Podajo analizo algoritma `FbHash` z drugimi algoritmi, ki rešujejo isti problem.
- Prikažejo, da iskanje podobnosti na nivoju zlogov ni dovolj pri stisnjenih datotekah.
- Naredijo analizo varnosti za `FbHash` in pokažejo, da je varen proti napadom z aktivnim napadalcem.

Poleg tega v našem delu algoritem ponovno implementiramo in preizkusimo več različic funkcij za uteževanje, ki se pojavijo v algoritmu. Rezultati naše implementacije algoritma se ujemajo z rezultati v [5]. Algoritem vedno zazna podobnost v datotekah z vsaj 5% ujemanja. Ugotovimo tudi, da za dobre rezultate zadošča relativno majhna baza datotek.

1.2 Struktura članka

Naš članek je organiziran na sledeč način. V 2. poglavju predstavimo predhodnike algoritma `FbHash`. V 3. poglavju podrobneje predstavimo algoritem `FbHash` in našo implementacijo. V 4. poglavju povzamemo ugotovitve v delu [5]. V 5. poglavju opišemo izvedene eksperimente, v 6. pa predstavimo njihove rezultate. Narejeno delo povzamemo in zaključimo v 7. poglavju.

2. SORODNA DELA

Prvi algoritem namenjen iskanju približnih ujemanj je bil objavljen leta 2002 pod imenom `dcfldd`. Ta algoritem je razvil N. Harbour kot izboljšano verzijo ukaza `dd` [10]. Izboljšana različica tega algoritma je `ssdeep`. Pomembnejša predhodnika algoritma `FbHash` sta tudi `MRSH-v2` in `mvHash-B`. Obstaja še `bbhash`, ki pa je časovno potraten in ga ne bomo podrobneje opisali.

2.1 ssdeep

Algoritem `ssdeep` je implementacija kontekstno sprožene kosovno zgoščevalne funkcije (angl. *Context Triggered Piecewise Hash*, CTPH), ki jo je predstavil J. Kornblum septembra 2006 v raziskavi [11]. Algoritem temelji na detektorju neželene elektronske pošte `spamsum`, ki lahko zazna sporočila, ki so podobna znanim neželenim sporočilom.

CTPH uporablja zgoščevanje po kosih (angl. *piecewise hashing*), kar pomeni, da se zgoščena vrednost izračuna na posameznih delih datoteke fiksne dolžine. Za razliko od `dcfldd` algoritem CTPH uporabi poljubno zgoščevalno funkcijo.

Drugi princip, ki ga uporablja CTPH, je zgoščevalna funkcija z drsečim oknom (angl. *rolling hash*), ki preslika zadnjih k zlogov v psevdonaključno vrednost. Vsakega naslednika je tako možno hitro izračunati iz predhodno izračunane vrednosti. Pri tem je uporabljena zgoščevalna funkcija `FNV`.

Postopek CTPH se začne z izračunom zgoščenih vrednosti z drsečim oknom. Ob določeni sprožilni zgoščeni vrednosti (angl. *trigger value*) se vzporedno s tem sproži še algoritem zgoščevanja po kosih. Ob ponovni pojavitvi sprožilne vrednosti se dotlej zbrane vrednosti druge zgoščevalne funkcije zapišejo v končni prstni odtis. Tako se ob lokalni spremembi v datoteki sprememba pozna le lokalno tudi v prstnem odtisu.

Sledi primerjava prstnih odtisov datotek, ki temelji na uteženi Levenstheinovi razdalji (angl. *edit distance*), ki je nato še skalirana in obrnjena, da predstavlja 0 povsem različna prstna odtisa, 1 pa povsem enaka.

Algoritem `ssdeep`, ki je implementacija CTPH, se izkaže pri primerjavi podobnih besedilnih datotek in dokumentov [11]. Po drugi strani pa lahko aktivni napadalec popravi "slabe" datoteke na tak način, da se izognejo črni listi [5]. Ker je prstni odtis fiksne dolžine, je algoritem primeren le za relativno majhne datoteke podobnih velikosti.

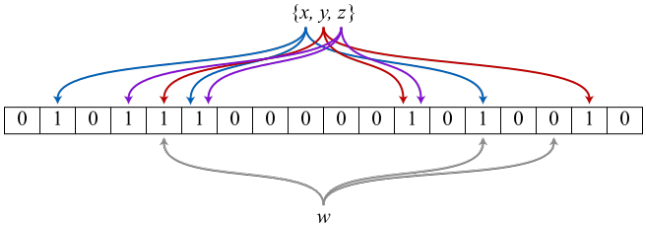
2.2 sdhash

Algoritem `sdhash` je opisal V. Roussev januarja 2010 v delu [13]. Glavna prednost tega algoritma pred predhodnimi je, da izbere statistično manj verjetne dele datotek kot izhodišče za računanje prstnega odtisa.

Postopek se začne z iskanjem statistično najmanj verjetnih delov datoteke. Izračuna se entropija skupin po k zlogov datoteke. Nato se izračuna rank vsake skupine glede na n sosednjih skupin. Izbrane so skupine, ki imajo rank večji ali enak postavljeni meji.

Sledi filtriranje skupin k zlogov, ki niso bistvene, povzročajo pa lažno pozitivne rezultate. Ocenili so, da je dobro zavreči skupine z oceno entropije pod 100 ali nad 990, ker so takšne skupine pogoste npr. v datotekah tipa JPEG.

Nato se generira prstni odtis datoteke kot zaporedje Bloomovih filtrov, ki so verjetnostni bitni vektorji, uporabljeni za prostorsko učinkovito predstavitev množic. Bloomov filter, ki predstavlja prazno množico, je vektor samih ničel. Ko dodamo element v tako predstavljeno množico, vzamemo fiksno število zgoščenih vrednosti elementa. Zgoščene vrednosti so definirane tako, da element preslikajo v pozicijo v vektorju. Nato dobljene pozicije v vektorju nastavimo na 1. Na sliki 1 vidimo, kako lahko preverimo, ali je nek element v Bloomovem filtru. Z več zgoščevalnimi funkcijami določimo pozicije v bitnem vektorju, ki morajo vse biti enake 1. Vidimo, da so x, y in z elementi filtra, w pa ne. Algoritem `sdhash` preveri za vsako izbrano skupino k zlogov, ali je že v množici, predstavljeni z Bloomovimi filtri. Če skupine ni v množici, jo algoritem doda.



Slika 1: Prikaz preverjanja, ali je element v Bloomovem filtru. Element w ni v filtru, ker zadnja zgoščevalna funkcija ne kaže na enico.

Nazadnje algoritem primerja prstne odtise datotek, torej zaporedje Bloomovih filtrov. Za vsak filter, ki predstavlja prvo datoteko, se izračuna maksimalna ocena podobnosti s filtri, ki predstavljajo drugo datoteko. Rezultat je povprečje tako pridobljenih ocen podobnosti.

Algoritem `sdhash` doseže boljša priklic in preciznost kot `ssdeep` [5]. A tudi ta algoritem ima več pomanjkljivosti: nekaterih datoteke ne more primerjati, primerjava datoteke same s seboj lahko vrne oceno med 50 in 100 ter prvih 15 zlogov sploh ne vpliva na končni prstni odtis. Poleg naštetega aktivni napadalec lahko spremeni "slabe" datoteke na tak način, da se izognejo črni listi oziroma "dobre" datoteke tako, da se obdržijo na beli listi [4].

2.3 MRSH-v2

Oktobra 2012 sta F. Breitinger in H. Baier predstavila algoritem `MRSH-v2` [3], ki se opira na predhodno razvit algoritem `MRSH` (angl. *multi-resolution similarity hashing*), ta pa temelji na algoritmu `ssdeep`.

Algoritem `MRSH` ima določene sprožilne točke $(-1) \bmod b$, kjer b pomeni povprečno velikost bloka. Namesto zgoščevalne funkcije z drsečim oknom uporabi polinomsko zgoščevalno funkcijo `djb2`, kot primitiv pa `MD5`. Namesto konkatenacije zgoščenih vrednosti `MRSH` kot prstni odtis uporabi seznam Bloomovih filtrov.

Algoritem `MRSH-v2` ponovno uporabi zgoščevalno funkcijo z drsečim oknom, kot `ssdeep`, namesto `FNV` pa uporabi funkcijo zgoščevanja `MD5`. Za večjo hitrost in v izogib napadu z dodajanjem sprožilnih točk je dodana tudi spodnja meja za velikost skupin zlogov $\frac{b}{4}$.

Algoritem `MRSH-v2` je po [3] časovno učinkovitejši od predhodnih algoritmov. Vključuje način za odkrivanje fragmentov in način za odkrivanje podobnih datotek. Po analizi leta 2014 [8], ki primerja `ssdeep`, `sdhash` in `MRSH-v2`, se v povprečju najbolje obnese `sdhash`, `ssdeep` in `sdhash` izkazu-

jeta dobro preciznost, vsi trije algoritmi pa imajo relativno slab priklic.

2.4 mvHash-B

Marca 2013 so F. Breitinger in sodelavci predstavili algoritem `mvHash-B` [2]. Ideja algoritma je, da majhne lokalne spremembe ne spremenijo končnega rezultata.

V prvem koraku se izvede večinsko glasovanje po bitih z nastavljivo mejo t. Vsakih k zlogov se tako preslika v ničle, če je število enic v zaporedju bitov manjše od t, sicer pa v enice.

Nato se zaporedje bitov zapiše na bolj kompakten način – enake zaporedne bite nadomestimo z dolžino takega niza. Če se zaporedje bitov ne začne z nekaj ničlami, bo prvo število v seznamu 0. Dobimo kodirano zaporedje dolžin nizov.

Kodirano zaporedje se razdeli v prekrivajoče se skupine fiksne dolžine. Te skupine so dodane v Bloomov filter. Nazadnje se primerja Bloomove filtre s pomočjo Hammingove razdalje, ocene pa se povpreči in odšteje od 100.

Algoritem `mvHash-B` naj bi bil hiter skoraj kot `SHA-1`, torej blizu zgornje meje učinkovitosti [2]. Vendar tudi ta algoritem ni varen pred aktivnim napadalcem, saj je možno popraviti "slabo" datoteko tako, da se izogne črni listi [6].

3. OPIS ALGORITMA

Avtorji v članku [5] predstavijo algoritem `FbHash-B`, ki je namenjen nestisnjenim (angl. *uncompressed*) in `FbHash-S`, ki je namenjen stisnjenim (angl. *compressed*) datotekam. Iskanje želenih datotek poteka na sledeč način. Najprej datoteke, ki jih želimo preveriti zgostimo s pomočjo ustreznega algoritma glede na njihov tip in tako dobim bazo podatkov. Nato zgostimo še ciljne datoteke in jih primerjamo z datotekami iz baze podatkov. Tako lahko datoteke, ki imajo s ciljnimi dovolj podobnosti zavržemo (angl. *whitelist*) oziroma dodamo med iskane (angl. *blacklist*).

3.1 FbHash-B

`FbHash-B` je bolj primeren za datoteke, ki niso stisnjene. V opisu algoritma bomo za boljšo preglednost uporabili sledečo notacijo:

- Del datoteke: niz k zaporednih zlogov.
- ch_i^D: del datoteke D, ki se začne na i-tem zlogu.
- Frekvenca dela oziroma $chf_{ch_i}^D$: število pojavitev dela datoteke ch_i v datoteki D.
- Datotečna frekvenca dela oziroma df_{ch}: število datotek, ki vsebujejo del datoteke ch.
- N: število datotek v bazi podatkov.
- $RollingHash(ch_i)$: vrednost rekurzivne zgoščevalne funkcije imenovane rolling hash na delu datoteke ch_i
- $chw_{ch_i}^D$: utež dela datoteke ch_i v datoteki D
- $docw_{ch_i}$: datotečna utež dela datoteke ch_i
- $W_{ch_i}^D$: ocena dela datoteke ch_i v datoteki D

Algoritem bomo predstavili v treh delih. V prvem delu datoteko razdelimo v dele, ki jih zgostimo z zgostitveno funkcijo in pretvorimo v utež glede na število njihovih pojavitev v datoteki. V drugem delu izračunamo datotečne uteži glede na to, v koliko datotekah se zgostitev dela pojavi. V tretjem delu pa iz datotečnih uteži in uteži dela izračunamo zgostitveno oceno, ki predstavlja datoteko.

3.1.1 Računanje frekvence delov datotek

V tem koraku se izračunajo frekvence delov datotek in njihove uteži. Vsaka datoteka vsebuje $N_D - k$ delov datotek, kjer je N_D dolžina datoteke D v zlogih in k parameter. Tako ima i-ti del datoteke D obliko $B_i^D B_{i+1}^D \cdots B_{i+k-1}^D = ch_i^D$, kjer je B_j^D j-ti zlog datoteke D.

Najprej izračunamo $RollingHash(ch_0^D)$ s formulo:

$$RollingHash(ch_0^D) = B_0^D \cdot a^{k-1} + \cdots + B_{k-1}^D \cdot a^0 \pmod n,$$

kjer so a, n in k parametri. Zaradi rekurzivne strukture funkcije $RollingHash$ lahko zgostitve preostalih delov izračunamo na sledeč način:

$$RollingHash(ch_{i+1}^D) = a \cdot RollingHash(ch_i^D) - B_i^D \cdot a^k - B_{i+k}^D \pmod n.$$

Parametre a, k in n izberemo na sledeč način. Za a izberemo neko konstanto med 2 in 255. Če za zalogo vrednosti funkcije RollingHash vzamemo 64-bitna števila, lahko k izračunamo na sledeč način:

$$B_i^D \cdot a^{k-1} \leq 2^{64} - 1.$$

Ker je maksimalna vrednost zloga in parametra a enaka 255 dobimo neenačbo $255^k \leq 2^{64} - 1$ iz česar sledi, da je k manjši ali enak 7. Tako za k izberemo 7. Če želimo zagotoviti, da ne pride do trkov med različnimi deli datotek, moramo za n izbrati praštevilo, ki je večje od $2^{56} = 256 \cdot 256^{k-1}$.

Strukturo datoteke lahko tako predstavimo kot razpršeno tabelo, kjer je ključ vrednost, ki jo dobimo s funkcijo $RollingHash$ na določenem delu datoteke, vrednost pa število pojavitev pripadajočega dela datoteke (oziroma vrednosti, ki smo jo dobili s funkcijo $RollingHash$).

Prvi korak zaključimo tako, da izračunamo uteži za dele besedila znotraj datoteke ($chw_{ch_i}^D$) s pomočjo ene izmed funkcij iz 3.4.

3.1.2 Računanje datotečnih uteži

Drugi korak je računanje datotečnih uteži delov besedila. Te nakazujejo koliko informacije prinese posamezen del datoteke znotraj določene baze podatkov.

Najprej je potrebno izračunati datotečno frekvenco za dele datotek, ki se v bazi podatkov pojavijo. To naredimo tako, da zgradimo razpršeno tabelo katere ključ je vrednost $RollingHash(ch_i)$, vrednost pa število pojavitev posameznega ključa. To s tabelo lahko sestavimo tako, da se sprehodimo čez vse ključe razpršenih tabel za posamezni datoteke in

zgoščevalni tabeli prištejemo 1, če se ključ v datoteki pojavi.

Iz frekvence posameznega dela nato izračunamo njegovo datotečno utež ($docw_{ch_i}$) s pomočjo ene izmed funkcij iz 3.4.

3.1.3 Računanje zgostitvenih ocen

V tretjem koraku izračunamo zgostitvene ocene za posamezne datoteke. S temi ocenami lahko kasneje datoteke med seboj primerjamo, kot je predstavljeno v 3.3.

V vsaki datoteki izračunamo oceno posameznega dela datoteke s formulo:

$$W^D_{ch_i} = docw_{ch_i} \cdot chw^D_{ch_i}.$$

Zgostitveno oceno za datoteko predstavimo kot razpršeno tabelo, kjer za ključ vzamemo vrednosti funkcije $RollingHash$, ki smo jih dobili v datoteki, vrednosti pa so ocene dela datoteke ($W^D_{ch_i}$).

3.2 FbHash-S

Med testiranjem različnih datotek so avtorji dela ugotovili, da algoritem `FbHash-B` na stisnjenih datotekah (tipov docx, pptx, pdf, zip ...) ne deluje najbolje. Tako datoteki, ki imata 90% podobnost v datoteki tipa docx, nimata veliko podobnosti na nivoju zlogov. Zato so razvili tudi verzijo algoritma `FbHash-S`, ki dosega dobre rezultate tudi na stisnjenih datotekah.

Algoritem `FbHash-S` deluje na sledeč način. Najprej datoteko razširimo (angl. *uncompress*) in uporabimo algoritem `FbHash-B` na vsaki izmed dobljenih datotek. Končna ocena pomembnosti je izračunana kot povprečje ocen posameznih datotek.

Zaradi dodatnega dela je algoritem `FbHash-S` časovno bolj zahteven od algoritma `FbHash-B`.

3.3 Primerjanje datotek

Podobnost med datotekama izračunamo s pomočjo kosinusne razdalje [9]. Najprej poskrbimo, da ima vsaka izmed datotek izračunano tabelo ocen pomembnosti. Podobnost med datotekama tako izračunamo s formulo:

$$Similarity(D_1, D_2) = \frac{\sum_{i=0}^{n-1} W^{D_1}_{ch_i} \cdot W^{D_2}_{ch_i}}{\sqrt{\sum_{i=0}^{n-1}(W^{D_1}_{ch_i})^2} \cdot \sqrt{\sum_{i=0}^{n-1}(W^{D_2}_{ch_i})^2}} \cdot 100.$$

Pri implementaciji lahko opazimo, da lahko za vsako datoteko vnaprej izračunamo $\sqrt{\sum_{i=0}^{n-1}(W^D_{ch_i})^2}$ in da je v števcu dovolj, da zmnožimo ocene pomembnosti, ki so v preseku ključev.

3.4 Uporabljene funkcije za uteževanje

Tukaj bomo predstavili različne funkcije, ki smo jih preizkusili za iskanje primernih uteži posameznih delov datoteke. Prve se uporabljajo za računanje uteži posameznega dela datoteke, druge pa za računanje datotečne uteži dela datoteke.

3.4.1 Funkcije za računanje uteži dela besedila

Najprej si oglejmo funkcijo za računanje uteži dela besedila, ki je predstavljena v delu [5]:

$$W^D_{ch_i}(chf^D_{ch}) = 1 + \log_{10}\left(\frac{chf^D_{ch}}{N^D}\right),$$

pri čemer N^D predstavlja število delov v datoteki, ki poskrbi za normalizacijo števila delov. Bolj kot je kos datoteke pogost, višja je njegova utež. Relativna frekvenca dela besedila je število med 0 in 1, torej je njegov logaritem med $-\infty$ in 1, saj je $\log_{10}(0)$ enak $-\infty$, $\log_{10}(1)$ pa 0. Menimo, da je to napaka (ali pa so avtorji članka uporabili drugačno funkcijo normiranja kot mi, saj je nikjer ne opišejo), saj v delu omenjata *TF-IDF* shemo, katera za izračun uteži pogosto uporabi formulo

$$\log\left(1 + \frac{chf^D_{ch}}{N^D}\right).$$

Zato smo uteži dela besedila izračunali nekoliko drugače.

Funkcijo smo popravili, da je rezultat število na intervalu med 0 in 1. Izračunamo relativno frekvenco dela in za 1 povečan rezultat logaritmiramo:

$$W^D_{ch_i}(chf^D_{ch}) = \log_2\left(1 + \frac{chf^D_{ch_i}}{N^D}\right).$$

Ker je ulomek povečan za 1 število na intervalu $[1, 2]$, smo vzeli logaritem z bazo 2. Tako kot utež dobimo število med 0 in 1.

Predlagamo še drugo funkcijo za računanje uteži, ki je razmerje med frekvenco dela besedila in številom takih delov besedila n. Tako je rezultat na intervalu $[0, 1]$:

$$W^D_{ch_i}(chf^D_{ch}) = \frac{chf^D_{ch_i}}{N^D}.$$

3.4.2 Funkcije za računanje datotečnih uteži

Prvo verzijo funkcije za izračun datotečne uteži dela besedila smo povzeli po raziskavi v [5]. Raziskava poda datotečno utež kot logaritmirano relativno frekvenco datotek, ki vsebujejo posamezen kos ch_i:

$$docw^D_{ch_i}(df_{ch}) = \log_{10}\left(\frac{N}{df_{ch}}\right).$$

Notranji ulomek je število med 1 in N, torej je rezultat število med 0 in $\log_{10}(N)$. Bolj kot je kos datoteke pogost, manjšo utež dobi.

Predlagamo še drugo verzijo funkcije za računanje datotečne uteži. Ulomek obrnemo in odštejemo od 1, da dobimo vrednost na intervalu $[0, 1]$, ki je še vedno obratno sorazmerna pogostosti dela datoteke:

$$docw^D_{ch_i}(df_{ch}) = 1 - \log_{10}\left(1 + \frac{df_{ch}}{N}\right).$$

4. UGOTOVITVE ČLANKA

Avtorji v delu svoj algoritem testirajo in pokažejo, da je varen proti aktivnemu napadalcu.

4.1 Varnost proti aktivnemu napadalcu

Napad z aktivnim napadalcem je napad, ko ima napadalec možnost spreminjanja datoteke. Varnost algoritma je prikazana tako, da opišemo znane napade na algoritme iskanja približnih ujemanj in podamo argumente, zakaj tak napad na `FbHash` ne deluje.

V članku [1] opišejo napad na `ssdeep` [11], ki datoteke spremeni tako, da se prikažejo, čeprav so na "črni listi". `Ssdeep` datoteko razdeli na 64 delov, ki jih zgosti s pomočjo kriptografske funkcije (naprimer md5). Tako je nastali podpis dolg 64 zlogov. Podpis datoteke na "črni listi" se ujema s podpisom datoteke, če se v njem ujema vsaj podniz dolg 7 zlogov. Želja napadalca je, da podpis povsem spremeni s čim manj spremembami v datoteki. To lahko naredi tako, da datoteko spremeni na način, da se vsak sedmi zlog spremeni ali pa vstavi delčke, ki zagotovijo, da se bo datoteka delila na različnih mestih in imela posledično različen podpis. Ta napad na algoritem `FbHash` ne deluje, saj se ob majhni spremembi datoteke frekvenca delov datoteke le malo spremeni.

V delu [4] prikažejo, da se lahko oceno podobnosti algoritma `sdhash` zlahka zniža pod 28%. Algoritem `sdhash` je razbit tudi v raziskavi v [7], ki pokaže, da lahko zgradimo več različnih datotek, ki bodo imele podobnost z podano datoteko enako 100%. Ta napada sta možna zato, ker je končna ocena sestavljena le iz dela datoteke. V `FbHash` je končna ocena sestavljena iz celotne datoteke, zato je vsaka sprememba opazna in ne prevelika.

Chang v delu [6] prikaže, da lahko napadalec algoritem `mvhash-v` prelisiči in dobi nizko oceno podobnosti z drugo datoteko tudi, ko sta datoteki zelo podobni. To je možno zato, ker `mvhash-v` uporabi funkcijo, ki podatke zgosti. `FbHash` zgostitvene funkcije ne uporabi, zato je na tak napad odporen.

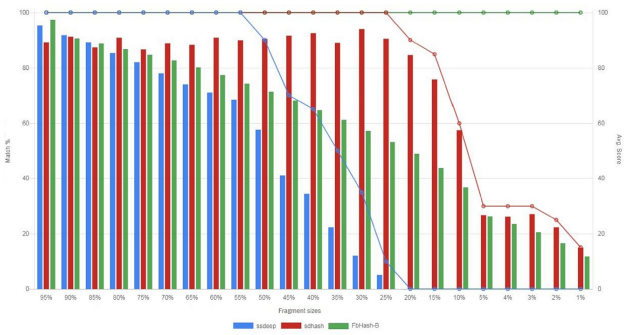
4.2 Testiranje algoritma

Algoritem `FbHash` je v članku primerjan z algoritmoma `ssdeep` in `sdhash`. Njegovo delovanje je primerjano po dveh metrikah: detekcija fragmentov in korelacija skupnega dela datoteke. Detekcija fragmentov nam pove, kako dobro algoritem zazna, da del datoteke (fragment) spada k določeni datoteki. Korelacija skupnega dela datoteke pa nam pove, kako dobro algoritem zazna, da dve datoteki vsebujeta enak del.

4.2.1 Detekcija fragmentov

Testi detekcije fragmentov so bili narejeni na bazi podatkov, ki je sestavljena iz fragmentov različnih velikosti in vzetih iz različnih mest v datoteki. Pri tem so uporabljene tekstovne datoteke in datoteke tipa docx iz [14].

Rezultati, ki jih avtorji dobijo na tekstovnih podatkih so prikazani na sliki 2. X-os grafa prikazuje velikost fragmenta, stolpci povprečno oceno podobnosti med delom in datoteko, točke povezane z daljicami pa prikazujejo procent pravilne korelacije dela z datoteko med poskusi. Vidimo lahko, da so rezultati algoritma `ssdeep` najslabši in da ocena podobnosti zelo pade, ko je fragment manjši od 50%. Za `sdhash` je opazno, da je ocena podobnosti skoraj v vseh primerih najvišja, a da pravilnost korelacije zelo pade, ko je del manjši od 15%. `FbHash` v vseh poskusih zazna, da sta del in datoteka korelirana, a je povprečna ocena podobnosti manjša od ocene algoritma `sdhash`.



Slika 2: Rezultati testiranja avtorjev na tekstovnih datotekah. Vir: [5]

Poleg tega preverijo natačnost algoritmov z metriko `F-score`, katere formula je:

$$F - score = 2 \cdot \frac{preciznost \cdot priklic}{preciznost + priklic},$$

kjer sta vrednosti *preciznost* in *priklic* izračunani s formulama

$$preciznost = \frac{TP}{TP + FP}, \quad priklic = \frac{TP}{TP + FN}.$$

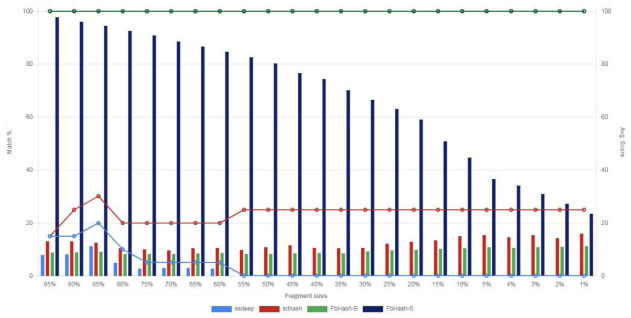
Tukaj TP predstavlja pravilno pozitiven primer, FP napačno pozitiven primer in FN napačno negativen primer. Na tekstovnih datotekah se je `FbHash` v vseh testih z metriko F-score izkazal najbolje, saj je imel rezultat med 94% in 98%. `Sdhash` in `ssdeep` pa sta imela rezultat 89,5% in 69%.

Rezultati testiranja na datotekah tipa docx so vidni na sliki 3. Vidimo lahko, da je tako kot na tekstovnih datotekah tudi tukaj `ssdeep` v obeh kategorijah najslabši in da ima `sdhash` višjo povprečno oceno kot algoritem `FbHash-B`, a del datoteke pravilno korelira z datoteko v manj primerih. `Sdhash` ima tukaj pri vsaki velikosti fragmenta podobno oceno (okoli 10%) in verjetnost korelacije (okoli 24%). Podobno je mogoče opaziti tudi v algoritmu `FbHash-B`, katerega ocena se giblje okoli 9% in verjetnost korelacije okoli 100%. Tako verjetnost korelacije ima v vseh primerih tudi algoritem `FbHash-S`, ki je bil narejen prav za datoteke tega tipa. To je vidno pri povprečni oceni, ki je pri vseh velikostih občutno višja od ocene drugih algoritmov.

Tudi mera `F-score` na datotekah tipa docx pokaže, da je `FbHash-S` za njih najbolj primeren. Ta ima oceno 92%, dočimer algoritmi `ssdeep`, `sdhash` in `FbHash-B` dosežejo 40%, 41% in 24%.

4.2.2 Korelacija skupnega dela datoteke

Testi korelacije skupnega dela datoteke so bili narejeni na bazi podatkov, ki vsebujejo datoteke s skupnimi deli različnih velikosti. Te datoteke so tekstovne ter datoteke tipa docx iz [14].



Slika 3: Rezultati testiranja avtorjev na datotekah tipa docx. Vir: [5]

Rezultati testov pokažejo, da se na tekstovnih datotekah `ssdeep` odreže najslabše in sicer z verjetnostjo $\geq 75\%$ zazna, da se datoteki ujemata, ko imata skupni del velikosti $\geq 30\%$, a ta verjetnost pri manjših delih hitro pade. `Sdhash` se bolje izkaže, saj zazna skupne dele, ki so veliki $\geq 3\%$ z verjetnostjo $\geq 93\%$. `FbHash` se izkaže najbolje, saj pade verjetnost, da zazna datoteki s skupnim delom pod 100% šele, ko imata skupen del, ki je velikosti le 1%. Takrat je verjetnost $\geq 93\%$.

Tudi na datotekah tipa docx rezultati testov pokažejo, da se `ssdeep` odreže najslabše. Ta ima največjo verjetnost, ko imata datoteki skupen del velikosti 10% in sicer 30%. Drugi velikosti skupnega dela pa dajo verjetnost $\leq 20\%$. Boljše rezultate doseže `sdhash`, ki ugotovi ujemanje datotek z skupnim delom velikosti 5% in 10% z verjetnostjo 90%. Zanimiv je rezultat funkcij `FbHash-B` in `FbHash-S`. Rezultat `FbHash-S` pokaže ujemanje 100% dokler imata datoteki skupnega več kot 2% in nato 90%. `FbHash-B` pa ugotovi, da imata datoteki skupen del z verjetnostjo 100% ko imata datoteki skupnega manj kot 40%. Ko imata skupnega več ta verjetnost pade na 80%.

5. NAŠI EKSPERIMENTI

Glavni cilj naših testov je bil ugotoviti, kateri izmed načinov določanja uteži dosega najboljše rezultate. V ta namen smo algoritma `FbHash-B` in `FbHash-S` implementirali [15], pripravili okolje za testiranje delovanja zgoščevalne funkcije ter preizkusili več načinov izračunavanja uteži.

5.1 Testiranje algoritma

Pri testiranju algoritma smo sledili postopku, ki so ga opisali avtorji dela [5].

5.1.1 Generiranje testnih primerov

Algoritem smo testirali na prosto dostopni bazi datotek [14]. Iz baze smo uporabili 100 besedilnih ter html datotek.

Testne primere za algoritem smo generirali tako, da iz zbirke vzamemo dve različni datoteki in naključen blok prvega vstavimo na naključno mesto v drugi datoteki na tak način, da je znan delež druge datoteke sedaj enak delu prve datoteke. Tako dobimo par datotek z natančno znanim ujemanjem.

5.1.2 Ocenjevanje delovanja algoritma

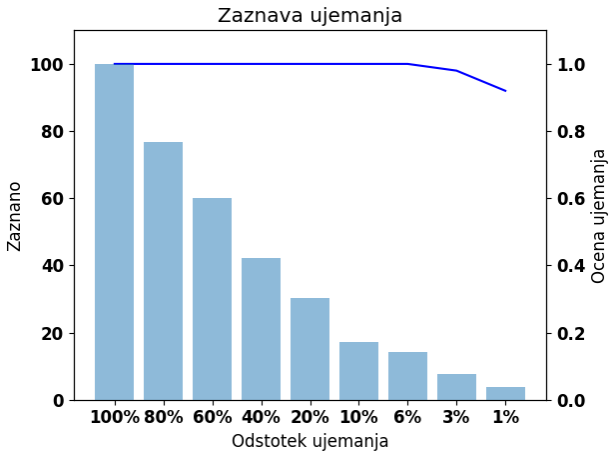
Da so naši rezultati čimbolj primerljivi s temi, ki so jih objavili v članku [5], smo tudi mi algoritem testirali na parih datotek z različnimi deleži ujemanja in opazovali dve vrednosti. Prva nam pove v kolikšnem deležu parov je algoritem odkril kakršnokoli ujemanje, druga pa predstavlja povprečno vrednost ujemanja, ki jo je algoritem izmeril. S tem lahko preverjamo kako majhne bloke besedila algoritem še prepozna in kako močna povezava obstaja med napovedano ter dejansko podobnostjo.

Tako kot v originalnem delu, smo se tudi mi odločili meriti vrednost F-score. Pri tem smo naključno generirali datoteke z različnimi merami ujemanja, med katerimi polovica datotek sploh ni imela nobenega ujemanja. Nato smo z algoritmom določili stopnjo ujemanja in opazovali, ali je ujemanje zaznal.

6. REZULTATI

Testirali smo več različnih funkcij za izračun uteži delov datotek.

Najprej si oglejmo delovanje funkcije, ki jo predlagajo avtorji članka [5] in je bila prva opisana v poglavju 3.4.1. Rezultati so predstavljeni na sliki 4. Algoritem s to formulo za uteži je dosegel F-score 94%, kar se ujema z vrednostjo, ki so jo dobili v članku.

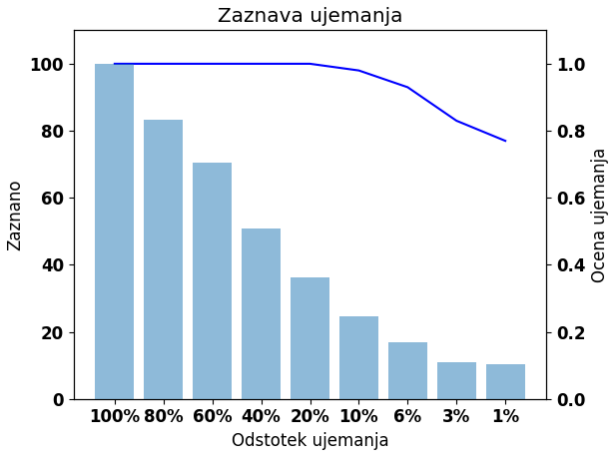


Slika 4: Rezultati testiranja z utežmi opisanimi v članku.

Rezultati, ki smo jih dobili se skladajo s tem, kar je predstavljeno v raziskavi v [5]. Vidimo lahko, da resnično ujemanje datotek ter izmerjena podobnost dobro sovpadata, vidimo pa tudi, da je algoritem v vseh primerih prepoznal delno ujemanje za vse dele datotek večje od 4% datoteke.

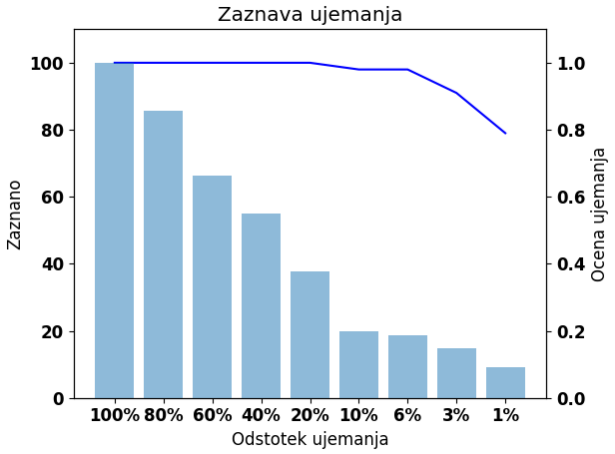
Ker se nam formula za izračun uteži ne zdi smiselna, smo jo nekoliko spremenili v obliko, ki je prav tako opisana v poglavju 3.4.1.

V tem primeru smo dosegli F-score 93%, kar je zelo blizu prejšnjega rezultata.



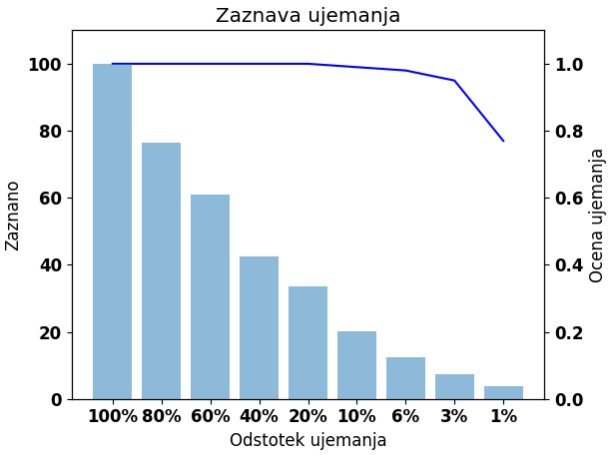
Slika 5: Rezultati testiranja s prilagojenimi utežmi.

Kot utež smo poskusili uporabiti tudi razmerje med frekvenco dela besedila in številom delov besedila. Tako smo ponovno dobili podobne rezultate (glej sliko 6). Pri tem smo izmerili F-score 92%.



Slika 6: Rezultati testiranja z razmerjem med frekvenco dela ter številom vseh delov

Pri teh testih smo opazili, da se rezultati pri različnih utežeh ne razlikujejo zelo veliko, zato smo poskusili tudi, kaj bi se zgodilo, če bi začetno analizo zbirke datotek izpustili in kot uteži uporabili konstantno vrednost (glej sliko 7). V tem primeru smo dosegli F-score vrednost 87%. Pri tem smo ugotovili, da zaradi tega zaznava podobnosti datotek res deluje slabše, saj algoritem že pri 20% ujemanju datotek v nekaterih primerih ujemanja ni prepoznal, vendar pa povezava med napovedano in resnično podobnostjo še vedno ostaja dovolj močna. Ta rezultat pomeni, da bi lahko predlagani algoritem uporabili za primerjavo datotek tudi v primerih, kjer nimamo velike zbirke gradiva na katerem lahko izračunamo uteži.



Slika 7: Rezultati testiranja brez uporabe zbirke besedil.

Vprašali smo se tudi, kaj bi se zgodilo v primeru, da bi namesto konstantnih uteži uporabili naključne vrednosti. V tem primeru so bili rezultati podobni tisti, kjer smo uporabili konstantno vrednost, F-score vrednost pa je dosegla 89%.

7. ZAKLJUČEK

V delu smo opisali algoritem za detekcijo podobnih delov datotek `FbHash`. Algoritem smo implementirali z več različicami funkcij za uteženje posameznih delov datotek. Implementacijo smo testirali na istih množicah kot so jih uporabili avtorji algoritma. Naša implementacija je dosegla F-score 94%, kar je enako kot v članku [5]. Ugotovili smo, da je algoritem uporaben tudi z relativno majhno bazo datotek.

8. ZAHVALA

V tem delu smo se močno opirali na članek [5], zato se iskreno zahvaljujemo avtorjem članka za opravljeno delo in razvoj algoritma `FbHash`. Algoritem `FbHash` je pomemben prispevek digitalni forenziki, saj pohitri preiskovanje datotek in poskrbi, da aktivni napadalec ne more zakriti relevantnih datotek.

9. REFERENCES

[1] H. Baier and F. Breitinger. Security aspects of piecewise hashing in computer forensics. In *2011 Sixth International Conference on IT Security Incident Management and IT Forensics*, pages 21–36, 2011.

[2] F. Breitinger, K. P. Astebøl, H. Baier, and C. Busch. mvhash-b - a new approach for similarity preserving hashing. In *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, pages 33–44, March 2013.

[3] F. Breitinger and H. Baier. *Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2*. October 2012.

[4] F. Breitinger, H. Baier, and J. Beckingham. Security and implementation analysis of the similarity digest sdhash. In *First international baltic conference on network security & forensics (nesefo)*, August 2012.

[5] D. Chang, M. Ghosh, S. K. Sanadhya, M. Singh, and D. R. White. Fbhash: A new similarity hashing scheme for digital forensics. In *The Digital Forensic Research Conference*, volume 29, pages S113–S123. DFRWS, July 2019.

[6] D. Chang, S. Sanadhya, and M. Singh. Security analysis of mvhash-b similarity hashing. *Journal of Digital Forensics, Security and Law*, 11(2):2, 2016.

[7] D. Chang, S. K. Sanadhya, M. Singh, and R. Verma. A collision attack on sdhash similarity hashing.

[8] Frank Breitinger, Vassil Roussev. Automated evaluation of approximate matching algorithms on real data. volume 11, pages S10 – S17, April 2014. Proceedings of the First Annual DFRWS Europe.

[9] S. Gerard and B. Christopher. *Term-weighting approaches in automatic text retrieval*. 1988.

[10] N. Harbour. Dcfldd. defense computer forensics lab. *online*, 2002.

[11] J. Kornblum. Identifying almost identical files using context triggered piecewise hashing. volume 3, pages 91–97, September 2006. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

[12] J. Ramos. Using tf-idf to determine word relevance in document queries.

[13] V. Roussev. *Data Fingerprinting with Similarity Digests*, volume 337. September 2010. Advances in Digital Forensics VI. DigitalForensics.

[14] Roussev, V. The t5 corpus. `http://roussev.net/t5/t5.html`. Accessed: 2020-04-29.

[15] Sebastian Mežnar, Timotej Knez, Jasmina Pegan. df-seminarska, 2020. `https://github.com/jasminapegan/df-seminarska`.