FbHash: A New Similarity Hashing Scheme for Digital Forensics

Timotej Knez
63160163

Sebastian Mežnar
27192031

Jasmina Pegan
63170423

POVZETEK
nek povzetek

Kategorija in opis področja
E.3 [**Data encryption**]

Splošni izrazi
Hashing

Ključne besede
Data fingerprinting, Similarity digests, Fuzzy hashing, TF-IDF, Cosine-similarity

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, kot so `ssdeep`, `sdhash` in `FbHash`, ki poskusijo filtrirati vnaprej znane "slabe" oziroma "dobre" datoteke. Ti algoritmi (angl. *Approximate Matching algorithms*) 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` ne poznamo.[6]

1.1 Prispevek
V našem delu predstavimo članek [6], v katerem avtorji predstavijo zgoščevalno funkcijo `FbHash`, ki omogoča lažje filtriranje "dobrih" in "slabih" datotek pri preiskovanju.

Glavni prispevki članka [6] so:

- Predstavijo shemo za približno ujemanje, ki je odporno proti aktivnemu napadalcu.
- Predstavijo algoritma za zgoščevanje stisnjenih in nestisnjenih datotek, ki temeljita na shemi TD-IDF [11].
- Podajo analizo algoritma `FbHash` z drugimi algoritmi, ki rešujejo isti problem.
- Prikažejo, da iskanje podobnosti na nivoju bajtov 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 preiskusimo različne funkcije za uteževanje, ki se pojavijo v algoritmu.

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 opišemo izvedene eksperimente in v 5. poglavju opišemo rezultate. V 6. poglavju povzamemo narejeno delo in rezultate.

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`[9]. 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 članku [10]. 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 kosih 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 (bajtov) 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.

Algoritem `ssdeep`, ki je implementacija CTPH, se izkaže pri primerjavi podobnih besedilnih datotek in dokumentov [10]. Po drugi strani pa lahko aktivni napadalec popravi "slabe" datoteke na tak način, da se izognejo črni listi [6]. 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 članku [12]. 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 v datotekah tipa JPEG.

Nato se generira prstni odtis datoteke kot zaporedje Bloomovih filtrov, ki so verjetnostne strukture, uporabljene za prostorsko učinkovito predstavitev množic. 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.

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` [6]. 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 [**?**].

2.3 MRSH-v2

Oktobra 2012 sta F. Breitinger in H. Baier predstavila algoritem `MRSH-v2` [4], 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 $-1modb$, 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 [4] časovno učinkovitejši od predhodnih algoritmov. Vključuje način za odkrivanje fragmentov in način za odkrivanje podobnih datotek. Po analizi leta 2014 [1], ki primerja `ssdeep`, `sdhash` in `MRSH-v2`, se v povprečju najbolje obnese `sdhash`, `ssdeep` in `sdhash` izkazujeta dobro preciznost, vsi trije algoritmi pa imajo relativno slab priklic.

2.4 mvHash-B

Marca 2013 so F. Breitinger in sod. predstavili algoritem `mvHash-B` [3]. 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 [3]. Vendar tudi ta algoritem ni varen pred aktivnim napadalcem, saj je možno popraviti "slabo" datoteko tako, da se izogne črni listi [7].

3. OPIS ALGORITMA

Avtorji v članku [6] 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 datotekam, ki niso stisnjene. V opisu algoritma bomo za boljšo preglednost uporabili sledečo notacijo:

- Del datoteke: niz k zaporednih bajtov.
- ch_i^D: del datoteke D, ki se začne na i-tem bajtu.
- Frekvenca dela oziroma $chf_{ch_i}^D$: število pojavitev dela datoteke ch_i v datoteki D.
- Dokumentna frekvenca dela oziroma df_{ch}: število dokumentov, 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
- $docw_{ch_i}$: dokumentna utež dela datoteke ch_i
- $W_{ch_i}^D$: ocena dela datoteke ch_i v datoteki D

Algoritem je predstavljen v treh korakih. Prvi zavzema računanje frekvence delov datotek, drugi računanje dokumentnih uteži delov datotek in tretji računanje zgostitvenih uteži.

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 bajtih 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 bajt v datoteki 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 bajta 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 $2^56 = 255 \cdot 255^{k-1}$.

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 dokumentnih uteži

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

Najprej je potrebno izračunati dokumentno 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 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 pomembnosti za posamezen del s formulo:

$$W_{ch_i}^D = docw_{ch_i} \cdot chw_{ch_i}^D.$$

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 pomembnosti ($W_{ch_i}^D$).

3.2 FbHash-S

Med testiranje različnih datotek, so avtorji članka 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 bajtov. 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. Podobnost med datotekami je izračunana tako, da med sabo primerjamo slike in besedila. Končna podobnost je izračunana kot povprečje podobnosti med posameznimi datotekami.

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

3.3 Primerjanje dokumentov

Podobnost med datotekama izračunamo s pomočjo kosinusne razdalje [14]. Najprej poskrbimo, da ima vsaka izmed datotek izračunano tabelo ocen pomembnosti. Podobnost med datotekama tako izračunamo s formulo:

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

Pri implementaciji lahko opazimo, da lahko za vsak dokument vnaprej izračunamo $\sum_{i=0}^{n-1} W_{ch_i}^{D}$ 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 preiskusili za iskanje primernih uteži posameznih delov datoteke.

3.4.1 funkcije za računanje uteži dela besedila

Najprej si oglejmo funkcijo za računanje uteži dela besedila, ki je predstavljena v članku [6]:

$$W_{ch_i}^{D}(chf_{ch}^{D}) = 1 + \log_{10}\left(\frac{chf_{ch}^{d}}{n}\right),$$

pri čemer je n število vseh dokumentov. Bolj kot je kos dokumenta pogost, višja je njegova utež. Relativna frekvenca dela besedila je število med 0 in 1, torej je njegov logaritem med $-\infty$ in 0. Menimo, da je to napaka (ali pa so avtorji članka uporabili drugačno funkcijo normiranja kot mi, saj je nikjer ne opišejo), zato smo uteži dela besedila ($W_{ch_i}^{D}$) 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:

$$W_{ch_i}^{D}(chf_{ch}^{D}) = \log_2\left(1 + \frac{chf_{ch_i}^{D}}{n}\right).$$

Ker je ulomek povečan za 1 število med 1 in 2, smo vzeli logaritem z bazo 2, da 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 $[0, 1]$:

$$W_{ch_i}^{D}(chf_{ch}^{D}) = \frac{chf_{ch_i}^{D}}{n}.$$

3.4.2 funkcije za računanje dokumentne uteži dela besedila

Članek [6] definira dokumentno utež dela besedila tako:

$$docw_{ch_i}^{D}(df_{ch}) = \log_{10}\left(\frac{m}{df_{ch}}\right),$$

kjer je m je število vseh dokumentov. Notranji ulomek je število med 0 in m, torej je rezultat število med 1 in $\log_{10} m$. Bolj kot je kos datoteke pogost, manjšo utež dobi.

Mi smo dokumentno utež dela besedila izračunali kot relativno frekvenco dokumentov, ki vsebujejo posamezen kos ch_i in rezultat logaritmirali:

$$docw_{ch_i}^{D} = \log_{10}\left(\frac{m}{df_{ch}}\right).$$

Dobljeno število je očitno nenegativno in je med 1 in $\log_{10} m$.

(Če bi želeli normirano vrednost:

$$docw_{ch_i}^{D} = 1 - \log_{10}\left(\frac{df_{ch}}{m}\right).$$

)

4. UGOTOVITVE V ČLANKU

Avtorji v članku svoj algoritem testiranjo 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 [2] opišejo naslednji napad na `ssdeep` [10], 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 bajtov. Podpis datoteke na "črni listi" se ujema s podpisom datoteke, če se v njem ujema vsaj podniz dolg 7 bajtov. Želja napadalca je, da podpis povsem spremeni z čim manj spremembami v datoteki. To lahko naredi tako, da spremeni datoteko na način, da se vsak sedmi bajt 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 datoteki frekvenca delov datoteke le malo spremeni.

V članku [5] prikažejo, da se lahko oceno podobnosti algoritma `sdhash` zlahka zniža pod 28. Algoritem `sdhash` je razbit tudi v članku [8], ki pokaže, da lahko zgradimo več različnih datotek, katerih podobnost z podano datoteko bo enaka 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 članku [7] 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 fregmentov 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*

4.2.2 *Korelacija skupnega dela datoteke*

Testi korelacija skupnega dela datoteke so bili narejeni bazi podatkov, ki vsebujejo datoteke s skupnimi deli različnih velikosti. Te datoteke so tekstovne ter datoteke tipa docx.

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 verjetnostko $\geq 93\%$. `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 naše raziskave je ugotoviti kateri izmed načinov določanja uteži proizvaja najboljše rezultate. V ta namen smo algoritem fbhash-B implementirali in pripravili okolje za testiranje delovanja zgoščevalne funkcije in preizkusili več načinov izračunavanja uteži.

5.1 Testiranje algoritma

Pri testiranju algoritma smo sledili postopku, ki so ga opisali avtorji članka [6].

5.1.1 *Generiranje testnih primerov*

Algoritem smo testirali na prosto dostopni bazi dokumentov [13].Iz baze smo uporabili 100 besedilnih ter html dokumentov, saj je algoritem FbHash-B namenjen tovrstnim besedilnim dokumentom.

Testne primere za algoritem smo generirali tako, da iz zbirke vzamemo dva različna dokumenta in naključen blok prvega vsavimo na naključno mesto v drugem dokumentu na tak način, da je znan delež drugega dokumenta sedaj enak delu prvega dokumenta. Tako dobimo par dokumentov z natančno znanim ujemanjem.

5.1.2 *Ocenjevanje delovanja algoritma*

Da so naši rezultati čimbolj primerljivi s temi, ki so jih objavili v članku [6], smo tudi mi algoritem testirali na parih dokumentov 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.

6. REZULTATI

Testirali smo več različnih funkcij za izračun uteži delov dokumentov.

Najprej si oglejmo delovanje funkcije, ki jo predlagajo avtorji članka [6] in je bila opisana v poglavju 3.4.1.

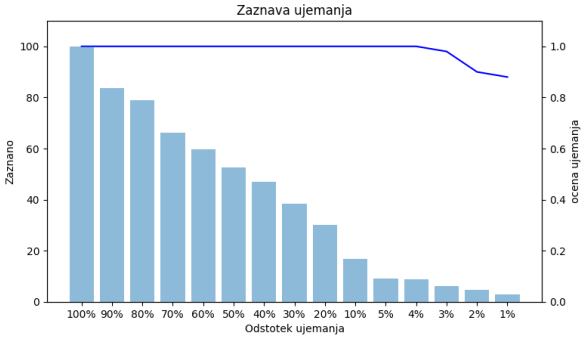

Figure 1: Rezultati testiranja z utežmi opisanimi v članku.

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.

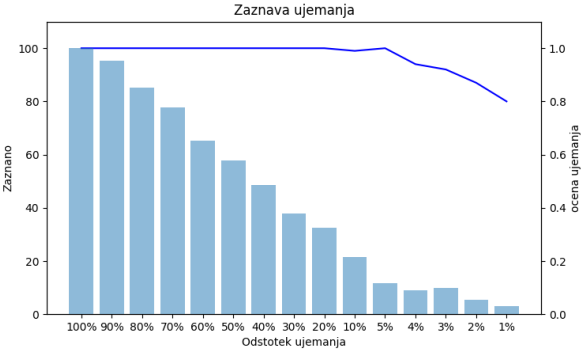

Figure 2: Rezultati testiranja z utežmi opisanimi v članku.

7. ZAKLJUČEK

8. ZAHVALA

Mogoče zahvala avtorjem za narjeno delo al kej.

9. REFERENCES

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

[2] 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.

[3] 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.

[4] F. Breitinger and H. Baier. Similarity preserving hashing: Eligible properties and a new algorithm mrsh-v2. In *Digital Forensics and Cyber Crime,*, pages 167–182, October 2012.

[5] 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.

[6] 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.

[7] 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.

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

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

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

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

[12] V. Roussev. Data fingerprinting with similarity digests. *IFIP Advances in Information and Communication Technology*, 337:207–226, September 2010. Advances in Digital Forensics VI. DigitalForensics.

[13] V. Roussev. An evaluation of forensic similarity hashes. *digital investigation*, 8:S34–S41, 2011.

[14] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *INFORMATION PROCESSING AND MANAGEMENT*, pages 513–523, 1988.