SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 152

Detekcija ubačenog zlonamjernog HTML koda na temelju HTML tagova u Web stranici

Mario Bašić

Zagreb, srpanj 2021.

SADRŽAJ

1. Uvod

Web je danas sveprisutna tehnologija na kojoj se temelje mnoge aktivnosti. Zahvaljujući Webu informacije se brže prenose nego ikada, a istovremeno imaju daleko širi doseg nego su to imale druge metode u povijesti. Međutim, Web nije bez opasnosti. Napadači koriste Web kako bi u njega ubacili zlonamjerne HTML i JavaScript kodove uz pomoć kojih pokušavaju, direktno ili indirektno, ostvariti nekakvu zaradu. Pravovremena detekcija takvih stranica izuzetno je bitna kako bi se zaštitili svi oni koji pristupaju Webu, ali i da bi sam Web ostao koristan. Detekcija ubačenih JavaScript i HTML fragmenata nije trivijalna te se rješavanju tog problema može pristupiti na mnogo načina.

U sklopu ovog završnog rada istražena je mogućnost detekcije zlonamjerno ubačenog HTML koda korištenjem samo HTML tagova i njihove strukture. Cilj je razviti metodu koja će koristiti uspoređivanje struktura HTML dokumenata za otkrivanje ubačenog HTML koda, tj. otkrivanje web-stranica koje su zaražene zlonamjernim HTML kodom. U tu svrhu napravljena je skripta koja vadi HTML strukturu iz dokumenta. Za skriptu je bilo potrebno odabrati HTML parser od više dostupnih. Također je napravljena skripta koja će za 2 HTML dokumenta pronaći sve novo ubačene HTML strukture. Skripta može ispravno pronaći sve novo ubačene strukture, no ne može otkriti ubačenu ili izmijenjenu HTML strukturu. Pomoću tih skripti je proveden eksperiment s podacima iz baze podataka koja sadrži URL-ove i HTML dokumente web-stranica iz `.hr` domene *Interneta*. Također se u sklopu ovog rada istražuje mogućnost uspoređivanja HTML strukture kao potencijalne metode za rješavanje problema gotovih duplikata HTML dokumenata.

U prvom poglavlju ovoga rada bit će objašnjeno kako je izvedeno vađenje HTML tagova i njihove strukture iz HTML dokumenta. Također će biti objašnjeni neka svojstva HTML dokumenata koja su bitna za cilj ovoga poglavlja.

U drugom poglavlju će biti prikazan algoritam koji se koristi za otkrivanje ubačene HTML strukture u novoj verziji HTML dokumenta koje nisu prisutne u staroj verziji dokumenta.

U trećem poglavlju ovoga rada će se opisati provedeni eksperiment pokušaja pronalaska kompromitiranih web-stranica pomoću programa opisanih u prvom i drugom poglavlju. Nakon toga će se prikazati, objasniti i analizirati rezultati toga eksperimenta.

U zadnjem poglavlju ovoga rada će se pojasniti problem gotovih duplikata HTML dokumenata s kojim se susreću web-pauci te kakav potencijalni doprinos mogu imati HTML strukture pri rješavanju ovog problema. Također će se usporediti količina prostora za pohranu podataka potrebnog u bazi podataka kako bi web-pauk koristio postojeće rješenje problema gotovih duplikata naspram količine prostora potrebnog kako bi se spremile HTML strukture svakog dokumenta u bazu podataka.

2. Vađenje HTML tagova i njihove strukture iz HTML dokumenta

HTML dokumenti [5] su nezamjenjiv dio današnjeg *Interneta*. Web-preglednici nakon posjete web-stranice prime od poslužitelja HTML dokument, te ga parsiraju i prikažu korisniku. Za prikaz HTML dokumenta, web-preglednici tijekom parsiranja dokumenta izgrađuju stablastu podatkovnu strukturu koja se zove DOM stablo [13], te pomoću nje prikažu dokument korisniku. HTML dokumenti se sastoje od HTML tagova koji su međusobno ugniježđeni jedan u drugome. HTML tagovi mogu biti otvarajući ili zatvarajući i najčešće se nalaze u dokumentu kao par otvarajućeg i zatvarajućeg taga. HTML tagovi mogu imati i proizvoljan broj atributa. Svaki atribut mora imati ime, a može imati i vrijednost. Između otvarajućeg i zatvarajućeg taga se može nalaziti sadržaj HTML taga. Imena, količina i način na koji su ugniježđeni HTML tagovi čine strukturu HTML dokumenta.

HTML dokumenti koje web-poslužitelj šalju klijentima mogu u sebi sadržavati zlonamjerne isječke HTML koda, tj. zlonamjerne HTML strukture. Takve zlonamjerne HTML strukture mogu biti rezultat uspješnog kompromitiranja web-poslužitelja od strane napadača, kojima je cilj zarada. Postoji puno strategija pomoću kojih HTML dokumenti s ubačenim malicioznim strukturama mogu donijeti zaradu napadačima. Te strategije mogu biti izravne ili neizravne. Neke od tih strategija nastoje sakriti ubačenu zlonamjernu HTML strukturu od korisnika i održavača web-stranice, tako da zlonamjerna struktura ostane što duže neprimijećena. Unatoč tome što je zlonamjerna struktura ne primijećena, to ne znači da ona ne može vršiti svoju zlonamjernu funkciju. Zlonamjerne strukture mogu biti vrlo složene, no mogu biti i vrlo jednostavne. Cilj ovog rada je otkriti upravo takve HTML strukture.

Primjer takvih napada su trovanje web-tražilica [4]. Web tražilice kao jedan od svojih kriterija za rangiranje web-tražilica koriste broj poveznica na tu stranicu pronađenih na drugim web-stranicama. Bolje rangirane web-stranice imaju veću šansu da ih korisnici web-tražilica posjete jer su više na listi rezultata web-tražilica. Trovanje

web-tražilica je umetanje skrivenih poveznica ili teksta u druge web-stranice koji vode k ciljnoj web-stranici. Trovanjem web-tražilica napadači pokušavaju postići da se ta ciljana web-stranica bolje rangira od strane web-tražilica.

Listing 2.1: HTML dokument sa skrivenim linkom koji može trovati web-tražilice

```
<div style="display: none;">
    <a href="http://example.org">link</a>
</div>
```

Dokument prikazanom u primjeru 2.1 je primjer trovanja web-tražilica. U njemu se nalazi skrivena poveznica na URL `http://example.org`. Ta poveznica se neće prikazati u web-pregledniku jer je u HTML tagu `<div style="display: none;">` postavljen atribut `style="display: none;"`

2.1. Pojam dobro oblikovanog HTML dokumenta

Pojam dobro oblikovanog dokumenta (engl. *well formed*) se odnosi ponajprije na XML dokumente [7]. Dok je pojam dobro oblikovanog XML dokumenta opisan u XML specifikaciji [7], taj isti pojam nije definiran za HTML dokumente. Umjesto toga, moderni web-preglednici ne provjeravaju kakav je HTML dokument prije nego što ga parsiraju i prikažu. U svrhu lakšeg pojašnjenja ovoga rada se uvodi pojam dobro oblikovanog HTML dokumenta. Ovaj pojam će biti relevantan pri parsiranju HTML dokumenata u svrhu vađenja strukture HTML tagova. XHTML dokumenti [9] su XML verzije HTML dokumenata koje su u biti isti HTML dokumenti s dodatnim pravilima koja osiguravaju da je svaki valjan XHTML dokument moguće parsirati XML parserima. U ovom radu će se dobro oblikovan HTML dokument definirati tako da se uzmu samo sljedeća 2 pravila iz skupa pravila za valjani XHTML dokument, tj. za dobro oblikovan HTML dokument mora vrijediti:

- Za svaki otvoreni HTML tag mora postojati odgovarajući zatvoreni HTML tag
- HTML tagovi moraju biti pravilno ugniježđeni (engl. *properly nested*)

Listing 2.2: HTML dokument s nedostajućim zatvorenim tagom

```
<html>
  <head>
</html>
```

Listing 2.3: dobro oblikovan HTML dokument

```
<html>
```

```
2   <head>
3   </head>
4 </html>
```

U primjeru 2.2, otvarajući HTML tag `<head>` nema svoj odgovarajući zatvarajući HTML tag `</head>` stoga primjer 2.2 nije *well formed* HTML dokument. Ako bi se prethodno spomenuti zatvarajući HTML tag `</head>` bio dodan nakon `<head>` taga, onda bi dokument bio dobro oblikovan. Primjer 2.3 prikazuje kako bi dokument tada izgledao.

Listing 2.4: HTML dokument s nepravilno ugniježđenim HTML tagovima

```
1 <html>
2   <head>
3 </html>
4   </head>
```

U primjeru 2.4, otvarajući HTML tag `<head>` se nalazi nakon otvarajućeg HTML taga `<html>`, no zatvarajući HTML tag `</html>` se nalazi prije zatvarajućeg `</head>` taga, stoga primjer 2.4 nije *well formed* HTML dokument jer HTML tagovi nisu pravilno ugniježđeni.

2.2. Parsiranje HTML dokumenta

Pri posjeti *Internetu* i dohvaćanju HTML dokumenata, nema garancije da će dohvaćeni dokument biti dobro oblikovani HTML dokument. Kao posljedica toga, nastaje problem nejednoznačnosti interpretacije HTML dokumenata.

Cilj parsiranja HTML dokumenta je dobiti listu čvorova gdje svaki čvor predstavlja jedan HTML tag. U čvoru je potrebno spremiti iduće informacije:

- Ime HTML taga,
- listu imena atributa HTML taga,
- je li HTML tag zatvarajući ili otvarajući,
- koliko je duboko HTML tag ugniježđen u dokumenta (skraćeno dubina)

Rezultat mora biti dobro oblikovana HTML struktura, te svaki otvarajući HTML tag mora imati odgovarajući zatvarajući HTML tag. To i uključuje prazne HTML tagove (engl. *empty HTML tag*) kao npr. `<br>`, za koje HTML5 specifikacija [5] preporučuje da ostanu bez zatvarajućeg HTML taga, koji također trebaju imati odgovarajuće zatvarajuće HTML tagove u rezultatu parsiranja.

2.2.1. Nejednoznačnost parsiranja HTML dokumenta

Listing 2.5: Jednostavan primjer neispravno oblikovanog HTML dokumenta

```
1 <div>
2   <p>
```

Jednostavni primjer 2.5 sadrži 2 otvorena HTML taga koji nemaju odgovarajući zatvarajući HTML tag. Struktura HTML dokumenta se ovdje može interpretirati na više načina:

- kao `<div> <p> </p> </div>` ili
- kao `<div> </div> <p> </p>`

Te dvije različite interpretacije imaju različitu HTML strukturu. Uobičajeni pristup parsiranju dokumenata je parsiranje pomoću kontekstno neovisne gramatike. Međutim takav pristup nije moguć za parsiranje HTML dokumenata zbog nejednoznačnosti parsiranja dokumenata. Moderni web-preglednici mogu uspješno prikazati dokumente poput dokumenta u 2.5 unatoč tome što nisu dobro oblikovani. Moderni web-preglednici su vrlo praštajući i popustljivi kad je u pitanju sintaksa HTML dokumenta [12]. Zbog toga se čak i dokument iz primjera 2.5 može smatrati važećim, jer će ga moderni web-preglednici uspješno prikazati. Takvo popustljivo ponašanje nije posljedica prošlih specifikacija za HTML dokumente, nego posljedica odluke web-preglednika. Doduše, nova HTML5 specifikacija [5] ima nekoliko smjernica za rješavanje situacija kada HTML dokument nije dobro oblikovan. Popustljivost je jedan od razloga zašto web-programeri više vole HTML nego XML dokumente za web-stranice. Bitno je da kod vađenja HTML strukture dokumenata u svrhu analize ubačenih HTML struktura uvijek koristimo isti HTML raščlanjivač (engl. *parser*), kako ne bi došlo do slučaja gdje se isti originalni dokument tumači na više različitih načina.

2.3. Programska potpora za parsiranje HTML dokumenta

Programsko rješenje za vađenje HTML strukture dokumenata je napisano u programskom jeziku Python. Pri tome se za parsiranje HTML dokumenata koristi biblioteka *Beautiful soup 4* [10] (skraćeno *bs4*). Biblioteka *bs4* može koristiti više različitih *parsera* za HTML dokumente. Svaki *parser* različito rješava situacije gdje se struktura HTML dokumenta može tumačiti na više načina. Važno je napomenuti da će se rezul-

tati svih raspoloživih *parsera* moći uspješno prikazati u modernim web-preglednicima. Od svih raspoloživih *parsera*, najviše se ističu:

- *html5lib* [6]
- *lxml* [11]

2.3.1. Odabir odgovarajućeg HTML *parsera*

Prednost *parsera html5lib* je to što se pridržava HTML5 specifikacije, te će izgrađeno DOM stablo ovog *parsera* biti najviše slično DOM stablu koji moderni web-preglednici izgrađuju i prikazuju. Međutim, ako je cilj parsiranja detekcija ubačene HTML strukture, onda *parser html5lib* ima nedostatak. Taj nedostatak je njegova velika popustljivost i pokušaji da ispravi dio dokumenta zbog kojeg se dokument ne može smatrati dobro oblikovanim tijekom parsiranja. Problem može nastati ako novo ubačena HTML struktura pokrene jedan od algoritama za modificiranje HTML strukture koji je sadržan unutar *parsera html5lib*. Takvi algoritmi se mogu pokrenuti i za dobro oblikovane HTML strukture.

Listing 2.6: Primjer dobro oblikovanog HTML documenta čiju će strukturu *parser html5lib* promijeniti

```
1  <html>
2  <head></head>
3  <body>
4
5      <p>
6          <a>
7              <div> </div>
8          </a>
9      </p>
10
11 </body>
12 </html>
```

Listing 2.7: Rezultat parsiranja dokumenta 2.6 *html5lib parserom*

```
1  <html>
2  <head></head>
3  <body>
4
5      <p>
6          <a></a>
7      </p>
```

```
8
9       <div>
10        <a></a>
11      </div>
12
13      <a></a>
14
15      <p></p>
16
17  </body>
18  </html>
```

U primjeru 2.7 se vidi rezultat parsiranja dokumenta iz primjera 2.6 korištenjem *html5lib parsera*. Primijetite da je parsirani dokument dobro oblikovan. U slučaju da je novo ubačena struktura okidač za jedan od algoritama za modificiranje HTML strukture koji je sadržan unutar *parsera html5lib*, takva struktura može izbjeći otkrivanje jer će nova verzija dokumenta u tom slučaju imati znatno drugačiju HTML strukturu od prijašnje verzije dokumenta.

Zbog navedenog nedostatka *parsera html5lib*, prikladnije je odabrati *parser lxml*. DOM stablo koje *parser lxml* vraća kao rezultat neće biti u svakom slučaju isto onomu koje moderni web-preglednici izgrađuju, tj. ono stablo koje korisnici *Interneta* vide, međutim, *parser lxml* nema algoritama za modificiranje HTML strukture tijekom parsiranja. HTML struktura iz primjera 2.6 ostaje ne promijenjena nakon prolaženja kroz *parser lxml*.

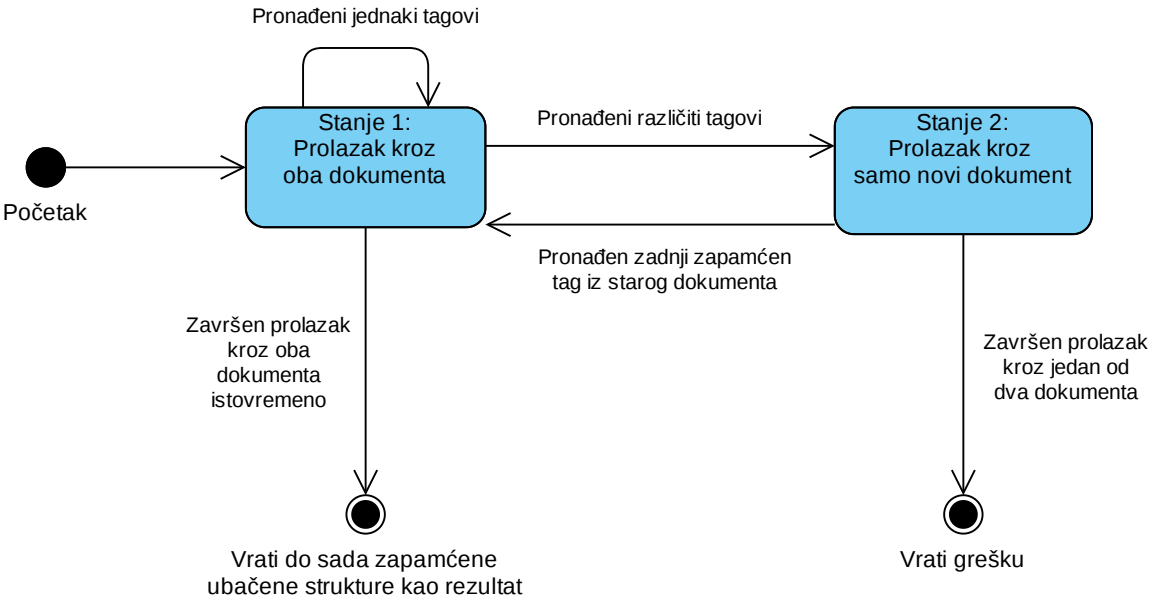
3. Pronalazak ubačene HTML strukture

Kada je web-stranica kompromitirana, u često slučajeva je HTML dokument koji posjetitelji te web-stranice dobiju na prvi pogled isti kao da web-stranica nije komprimirana. Drugi riječima, maliciozni HTML kod pokušava ostati neotkriven od održavača i posjetitelja te web-stranice. Potreban je algoritam koji će za cilj imati otkrivanje bilo kakvih ubačenog HTML kodova promatrajući samo HTML strukturu dokumenta, tj. algoritam će gledati samo HTML tagove, njihovu strukturu i opcionalno koje atribute tagovi sadrže. Nakon toga se otkrivene HTML strukture moraju ocijeniti kao zlonamjerne ili benigne. Vrijednost HTML atributa se ne gleda, nego samo njihova imena. Tijekom vađenja HTML strukture su HTML atributi poredani abecedno i njihove vrijednosti su izbačene. Prednost ovakvog pristupa naspram traženja bilo kakvog ubačenog HTML koda je to što će se male izmjene unutar web-stranice koje ne mijenjaju HTML strukturu zanemariti.

3.1. Algoritam za otkrivanje ubačene HTML strukture

Algoritmu se na ulaz predaju HTML struktura, tj. rezultat parsiranja novog dokumenta i HTML struktura starog dokumenta, gdje je stari dokument prijašnja verzija novog dokumenta nakon koje se sumnja da je u dokument ubačena zlonamjerna HTML struktura tj. zlonamjerni HTML kod. U privitku 8 se nalazi izvorni kod algoritma u programskom jeziku Python. Kao rezultat vraća dio liste čvorova novog dokumenta koje je otkrio kao novo ubačenu HTML strukturu.

Diagram stanja algoritma je prikazan u slici 3.1. Algoritam istovremeno prolazi kroz (engl. *iterate through*) liste čvorova koje su dobivene kao rezultat parsiranja novog i starog HTML dokumenta. Svaki čvor u toj listi reprezentira jedan HTML tag u strukturi dokumenta te sadrži sve potrebne informacije o tom tagu. Algoritam počinje u stanju 1 u kojem nastavlja prolaziti kroz liste čvorova dokle god algoritam nailazi na



Slika 3.1: Diagram stanja algoritma

iste čvorove. Dva čvora pri usporedbi mogu biti jednaka ako i samo ako su im jednaka imena HTML taga, lista imena atributa HTML taga i imaju jednaku dubinu. Drugim riječima, usporedba se vrši po svim vrijednostima spremljenim u čvorovima. Nakon što algoritam naiđe na različite čvorove, algoritam prelazi u stanje 2 u kojem nastavlja prolaziti kroz listu čvorova samo novog dokumenta sve dok u novom dokumentu ne pronađe isti čvor koji je zadnji pronađen u starom dokumentu te pri tome pamti svaki čvor kroz koji je prošao. Nakon što se pronađe u novom dokumentu zadnji pronađen u starom dokumentu, do tada zapamćeni čvorovi se označe kao ubačena HTML struktura te algoritam nastavlja u stanju 1. Ako algoritam istovremeno završi s prolaženjem kroz obje liste, onda algoritam vraća sve do tada označene ubačene strukture kao rezultat. Ako algoritam završi s prolaženjem kroz jedne liste čvorova, dok s prolaženjem kroz drugu listu nije završio, u tom slučaju se radi o kombinaciji izbačenih i ubačenih HTML struktura te algoritam javlja da nije otkrio niti jednu ubačenu strukturu.

Algoritam može samo otkriti ubacivanje HTML strukture, ne može kombinaciju izbacivanja i ubacivanja HTML tagova. Pronalazak izbačenih HTML struktura je moguć tako da se novi dokument predaje algoritmu kao stari dokument, a stari dokument kao novi, no pronalazak izbačenih struktura nije potreban.

4. Eksperiment detekcije ubačenog HTML koda

Zadaća web-pauka (engl. *web crawler*) [3] je posjećivati web-stranice tako da se za svaku posjećenu web-stranicu posjeti svaka poveznica na toj web-stranici. Pri tom pretraživanju se bilježe posjećene web-stranice i spremaju u nekakvu bazu podataka.

Provedeno je testiranje skripte za detekciju ubačenog HTML koda. Testiranje je provedeno nad MongoDB bazom podataka [1] koja sadrži URL-ove i HTML dokumente naslovnih web-stranica u `.hr` domeni koju popunjava web-pauk. MongoDB baza podataka se sastoji od kolekcija, a svaka kolekcija se sastoji od dokumenata. Svaki dokument ima jedan ili više parova ime polja i vrijednost polja. Po svojoj strukturi su slični rječnicima u skriptim jezicima poput Pythona. Eksperiment je proveden pomoću programskog jezika Python, a spajanje na MongoDB bazu podataka je ostvareno uz pomoć biblioteke *PyMongo* [2]. Baza podataka sadrži dvije relevantne kolekcije: kolekciju s URL-ovima i kolekciju s HTML dokumentima. Svaki dokument u kolekciji s URL-ovima ima polje `url` koje sadrži URL s kojeg je dokument dohvaćen, te ima polje `checks` u kojem je sadržana lista objekata. Svaki objekt u toj listi predstavlja jednu provjeru URL-a, tj. posjet URL-u pri kojem se računa kriptografski sažetak dobivenog HTML dokumenta. Dokumenti u kolekciji s HTML dokumentima u sebi sadrže polje `hash`. Vrijednost tog polja je kriptografski sažetak dokumenta, a sam dokument je sadržan u polju `page`. Baza podataka se nadopunjava i ažurira na dnevnoj bazi. Svaki URL se u određenim vremenskim intervalima ponovno provjerava je li se HTML dokument promijenio te ako je, onda se to bilježi i sprema se nova verzija dokumenta dohvaćenog s web-stranice.

Proces testiranja je u koracima izgledao ovako:

1. Prvo se u kolekciji koja sadrži URL-ove traže URL-ovi koji u bazi podataka imaju najmanje dva dohvata HTML dokumenta s URL-a u različitim vremenskim trenutcima, te među tim dohvatima se nalaza najmanje dvije različite verzije dokumenta. Različite verzije se mogu utvrditi uspoređivanjem kriptograf-

skog sažetka spremljenog u bazi podataka za svaki dohvat HTML dokumenta s URL-a.

2. Za sve URL-ove pronađene u 1. koraku se dohvaćaju najnovija i druga najnovija verzija HTML dokumenta iz baze podataka i spremaju se lokalno. Pri tome se vodi računa da je web-pauk koji ispunjava bazu podataka dobio u HTTP odgovoru uspješan status pri zadnjoj posjeti URL-u. Također se sprema podatak s kojeg URL-a su ti HTML dokumenti dohvaćeni.

3. Za svaki uspješno dohvaćeni i spremljeni par dokumenata iz 2. koraka se provodi vađenje HTML strukture i otkrivanje ubačenih HTML struktura između dvije verzije dokumenata.

4. Ako je uspješno otkrivena jedna ili više ubačenih HTML struktura, stvara se zapis koji u sebi sadrži izgled otkrivenih struktura i s kojeg URL-a su ti HTML dokumenti dohvaćeni. Lokalno spremljeni HTML dokumenti se onda mogu obrisati.

5. Za svaki zapis se ručno posjećuje URL i pokušava pronaći ubačena HTML struktura u dokumentu koji se dobije pri posjeti URL-a, tj. web-stranice.

6. Ako je ubačena struktura uspješno pronađena, daje se ručna ocjena je li struktura zlonamjerna ili benigna.

4.1. Prednosti i nedostaci metode eksperimenta

Metoda može uspješno otkriti maliciozni HTML kod samo ako sljedeće pretpostavke vrijede:

1. Maliciozni HTML kod je ubačen u dokument prilikom zadnjeg ažuriranja dokumenta, tj. maliciozni HTML kod je prisutan u najnovijoj verziji dokumenta, a u drugoj najnovijoj verziji nije prisutan.

2. Zadnje ažuriranje HTML dokumenta sadrži samo ubacivanje nove HTML strukture, ne smije sadržavati i izbacivanje.

3. U vremenskom intervalu između dohvaćanja najnovijeg dokumenta iz baze podataka i ručnog posjeta URL-a s kojeg su dohvaćeni dokumenti, dokument koji je ručno dohvaćen ne smije sadržavati promjene koje na bilo koji način mijenjaju dio HTML dokumenta koji je otkriven kao ubačena HTML struktura.

Navedeni uvjeti su ograničenja metode. Osim tih ograničenja, nedostatak metode je to što se mora ručno posjetiti URL, pronaći ubačenu HTML strukturu u dobivenom dokumentu i ocijeniti ju kao zlonamjernu ili benignu. Ugrađeni alati modernih web-preglednika mogu pomoći pronaći sva ponavljanja pojedinih HTML tagova, međutim čak i uz tu pomoć, proces ručnog traženja strukture je dugotrajan.

Prednost navedene metode je što je ipak veći dio posla automatiziran. Preciznije, dohvaćanje i provjeravanje URL-ova iz baze podataka, dohvaćanje HTML dokumenata te otkrivanje ubačenih HTML struktura je sve automatizirano.

4.2. Rezultati provedenog eksperimenta

Metodom 4 je obavljen eksperiment pomoću podataka iz dostupne baze podataka.

Tablica 4.1: Rezultati

Opis	Vrijednost
Broj provjerenih URL-ova	7000
Broj parova dokumenata	1090
Broj URL-ova gdje je otkrivena ubačena struktura	20
Broj URL-ova duplikata	6
Broj uspješno ručno pronađenih ubačenih struktura	12
Broj struktura ocijenjenih kao benigne	12

U tablici 4.1 je prikazano da je tijekom eksperimenta je iz baze podataka provjereno 7000 URL-ova sadrže li dvije verzije HTML dokumenata gdje je jedna verzija najnovija te se prilikom dohvaćanja najnovije verzije dokumenta s *Interneta* dobilo uspješni HTTP odgovor (engl. *HTML response*). Provjerom je pronađeno 1090 URL-ova koji zadovoljavaju navedene uvjete, tj. dohvatilo se iz baze podataka 1090 parova najnovije i druge najnovije verzije HTML dokumenata. Nad tim 1090 parova je provedeno otkrivanje ubačenih HTML struktura. Algoritam je vratio rezultat u 20 slučaja, tj. otkrio je ubačene HTML strukture za 20 web-stranica, međutim u 6 slučaja se radilo o duplikatima ili gotovim duplikatima HTML dokumenata, tj. sasvim istom web-stranicom. Za preostalih 14 slučaja je obavljena ručna analiza tako da se posjeti URL i onda traži ubačena HTML struktura koju je algoritam vratio kao rezultat, te se ubačena HTML struktura uspješno pronašla unutar originalnog HTML dokumenta u 12 slučajeva. Svih 12 uspješno pronađenih ubačenih HTML struktura su ocijenjene kao benigne.

Primjer takve jedne benigne ubačene HTML strukture je na web-stranici `http://turizam-bilogorabjelovar.com.hr/`. Otkrivena je ubačena struktura `<div> <a> <img> </img> <div> </div> </a> </div>`. Ručnom analizom se utvrdilo da se radi o dodavanju poveznice i slike na kraj niza mnogo takvih istih struktura, koje zajedno čine dinamički dio web-stranice. Nijedna takva struktura u nizu nije sadržavala zlonamjerni HTML kod.

Metoda se pokazala kao nedovoljnom da istraži mogućnost detektiranja zlonamjernog HTML koda promatrajući samo HTML strukturu dokumenata. Glavni problem je težina i dugotrajnost ručnog pronalaženja ubačenih struktura u originalnom dokumentu te njihovo ručno ocjenjivanje.

5. Problem gotovih duplikata HTML dokumenata

Značajan problem *web crawlera* su gotovi duplikati (engl. *near duplicate*) HTML dokumenata. Gotovi duplikati su HTML dokumenti koji su gotovo identični, ali se razlikuju u malim detaljima kao na primjer u jednom bajtu. Ako *web crawler* koristi kriptografske sažetke HTML dokumenata u svrhu usporedbe web-stranica, onda neće imati sposobnost prepoznavanja gotovih duplikata. Gotovi duplikati mogu nastati kao posljedica postojanja dinamičkih elemenata na web-stranici kao npr. oglasa. Takvi dinamički elementi će često biti drukčiji pri svakom posjetu web-stranici. Idealno bi bilo kad bi *web crawler* mogao raspoznati skore duplikate.

5.1. Postojeća rješenja problema gotovih duplikata

Postojeće rješenje koje koristi *web crawler* tvrtke Google je uporaba sim-sažetka (engl. *simhash*). U radu [8] je opisano kako funkcionira *simhash* te kako efikasno riješiti problem računanja Hammingove udaljenosti (engl. *Hamming distance*) između *simhasheva* u bazi podataka koju ispunjava *web crawler*. Hammingova udaljenost je jednaka broju bitova u kojima se 2 *simhasha* razlikuju, tj. broju potrebnih zamjena bitova da bi se iz jednog *simhasha* dobio drugi *simhash*.

Za razliku od kriptografskih sažetaka, *simhash* nema svojstvo da se sažetak u potpunosti promjeni za bilo kakvu promjenu u dokumentu iz kojeg se radi sažetak čak i ako je ta promjena u samo jednom bitu. Kod *simhasha* mala promjena u dokumentu rezultira malom promjenom u sažetku. *Simhash* se računa tako da se prvo dokument za koji se računa *simhash* rastavi na po volji odabrane značajke. Te značajke mogu za primjer biti trigrami riječi dokumenta. Zatim se za svaku značajku izračuna sažetak koji nije kriptografski kao npr. FNV-1a koji je iste duljine kao i ciljni *simhash* kojeg se želi izračunati. Nakon što je izračunat sažetak svake značajke se mogu dobiti bitovi *simhasha* na idući način: bit *simhasha* težine **w** jednak je zaokruženom bitovnom prosjeku

(engl. *bitwise average*) svih bitova težine **w** sažetaka svih značajki. Bitovni prosjek se dobije tako da se zbroje svi bitovi, podjele s ukupnim brojem bitova te zaokruže na 1 ili 0. Pomoću *simhasha* se mogu prepoznati skori duplikati tako da se dva HTML dokumenta smatraju gotovim duplikatima ako je Hammingova udaljenost *simhashova* ta dva dokumenta manja od zadane vrijednosti **k**.

5.2. Potencijalni doprinos HTML strukture problemu gotovih duplikata

Korištenje *simhasha* je rješenje koje se već pokazalo da radi u praksi. Kako bi se razmotrila usporedba HTML struktura kao potencijalna alternativa korištenju *simhasha* kao rješenja za problem skorih duplikata, treba provesti eksperimente koji će procijeniti kolika je uspješnost usporedbe HTML struktura kao metode otkrivanja skorih duplikata. Nakon toga se dobivena uspješnost treba usporediti s uspješnošću metode koju koristi *simhasha*. Ovakav pristup će zahtijevati ručnu analizu ili skup podataka koji sadrži parove verzija HTML dokumenata koji su već etiketirani (engl. *labeled*) kao skori duplikati ili različiti.

Alternativni pristup je koristiti usporedbu *simhasha* kako bi se napravio skup podataka parova verzija HTML dokumenata koji su etiketirani kao skori duplikati ili različiti pomoću *simhasha*. Kod ovakvog pristupa se ne može direktno metoda usporedbe HTML struktura usporediti s metodom usporedbe *simhasha*, međutim tada se proces može automatizirati.

5.2.1. Usporedba veličine podataka potrebne za simhash i HTML strukture

Ako bi baza podataka koristila *simhash* i računanje Hammingove udaljenosti za prepoznavanje gotovih duplikata, potrebno je spremiti u bazu podataka *simhash* HTML dokumenta uz sam dokument. Kako bi se računanje Hammingove udaljenosti *simhashova* obavljalo efikasno, također su potrebne dodatne tablice u bazi podataka. Broj tih dodatnih tablica ovisi o parametru **k**. Preporučena vrijednost parametra **k** je $\mathbf{k} = 3$ [8], te uz tu vrijednost **k** je potrebno 20 dodatnih tablica koje sadrže permutacije *simhashova*.

Za bazu podataka koja sadrži **n** dokumenata i koja sprema dokumente u *utf-8* kodiranju, dodatnih 20 tablica za efikasno računanje Hammingove udaljenosti *simhashova*

će trebati $\mathbf{n} * 64 * 20$ bitova za pohranu pod pretpostavkom da se koristi *simhash* od 64 bita, te da je potrebno ne više od 64 bita za spremanje *simhasha*. Za istu bazu će biti potrebno $\mathbf{n} * 64$ za spremanje *simhasha* HTML dokumenta uz sam dokument. Ukupno, ako se koristi *simhash* i računanje Hammingove udaljenosti za prepoznavanje gotovih duplikata, potrebno je $\mathbf{n} * 64 * 21$ dodatnih bitova ili $\mathbf{n} * 168$ dodatnih bajtova količine podataka u bazi podataka.

U svrhu utvrđivanja prosječne količine podataka potrebne za spremanje HTML strukture dokumenta je proveden eksperiment nad uzorkom od 1000 dokumenata. U eksperimentu se računa prosječna veličina HTML strukture spremljene kao niz znakova kodiran u *utf-8* kodiranju. Kao rezultat eksperimenta je dobivena vrijednost $\mathbf{s} = 8108$ gdje je **s** prosječna veličina strukture u bajtovima. Za usporedbu, prosječna veličina cijelog HTML dokumenta spremljenog na identičan način je $\mathbf{s} = 102348$ bajtova.

Ako bi se u bazi podataka za svaki HTML dokument spremala i njegova struktura, bilo bi potrebno u bazu podataka spremiti dodatnih $\mathbf{n} * 8108$ bajtova, što je za $\mathbf{n} * 7940$ bajtova više od količine podataka potrebno ako bi se koristio *simhash* i računanje Hammingove udaljenosti za prepoznavanje gotovih duplikata.

6. Načini na koje se može poboljšati postupak

U odjeljku 4.1 su navedeni nedostaci testiranog postupka. Proces pronalaženja ubačenih struktura u originalnom dokumentu se može automatizirati ako se osmisli algoritam za pronalazak isječka HTML koda u originalnom dokumentu za zadanu HTML strukturu. Ako se to postigne, onda će se vrlo brzo ručno moći ocijeniti ubačena HTML struktura kao benigna ili kao zlonamjerna.

Drugi način na koji se može postupak 4 poboljšati je da se tijekom otkrivanja ubačenih HTML struktura automatski filtriraju strukture po sljedećim kriterijima:

1. Otkrivena ubačena HTML struktura se jednom ili više puta već pojavljuje u obje verzije HTML dokumenta.

2. Otkrivena ubačena HTML struktura je po broju HTML tagova koje sadrži veća od zadane vrijednosti **n**

Ako neka otkrivena ubačena HTML struktura zadovoljava oba uvjeta, onda se ubačena struktura filtrira, tj. ne vraća se kao dio rezultata funkcije. Primjenjivanjem ovog filtriranja se mogu predvidjeti benigne promjene koje su nastale kao rezultat uobičajenih ažuriranja web-stranica. Npr. ako se neki web-portal ažurira s novim člankom, tada će struktura novog članka biti otkrivena ali filtrirana jer će zadovoljavati oba uvjeta. 2. uvjet je važan jer se time može izbjeći filtriranje zlonamjernih ubačenih HTML struktura. Npr. ako je u neku web-stranicu ubačena skrivena maliciozna poveznica (engl. *link*) koja ima HTML strukturu `<a> </a>`, nije poželjno da se ta poveznica filtrira i zanemari. Vjerojatnost da prosječni HTML dokument na *Internetu* sadrži HTML tag `<a> </a>` je puno veća nego da sadrži neku, po broju HTML tagova, veću HTML strukturu.

Za utvrđivanje je li se ubačena HTML struktura već pojavljuje u dokumentu se koristi uspoređivanje po HTML strukturi. Alternativa tomu je da se pronađe originalni

isječak HTML koda kojem ubačena HTML struktura pripada, te da se izračuna njegov *simhash*. Nakon što se izračuna *simhash*, treba utvrditi ima li neki drugi isječak u dokumentu isti ili slični *simhash*, tj. je li ubačeni HTML isječak gotovi duplikat nekog drugog već postojećeg isječka u dokumentu te ako je, onda se otkrivena HTML struktura filtrira i zanemaruje. Računanje *simhasha* umjesto računanja kriptografskog sažetka ili uspoređivanja stringova je u ovom slučaju može biti bolje jer se u protivnom slučaju neće moći tražiti gotovi duplikati HTML isječka nego samo doslovni duplikati.

7. Zaključak

U ovom radu je, uz cilj da se istraži mogućnost detekcije zlonamjerno ubačenog HTML koda korištenjem samo HTML tagova i njihove strukture, osmišljena metoda koja će HTML dokument neke web-stranice pokušati otkriti postoji li maliciozni HTML kod u tom dokumentu. Metoda je koristila detekciju novo ubačenog HTML koda u najnovijoj verziji HTML dokumenta naspram druge najnovije verzije tog dokumenta. Iako je tijekom eksperimenta provedenog u sklopu ovog rada uspješno pronađeno više ubačenih HTML struktura, pokazalo se da je ručno traženje ubačenih HTML struktura u originalnom HTML dokumentu proces koji je previše težak. Zbog toga opisana metoda nije prikladna za istraživanje mogućnosti detekcije zlonamjerno ubačenog HTML koda korištenjem samo HTML tagova i njihove strukture, te ta mogućnost ostaje neistražena. Unatoč tome, opisana metoda će možda biti dovoljno dobra ako se nadogradi s prijedlozima navedenim u odjeljku 6.

Za istraživanje mogućnosti uspoređivanja HTML strukture kao potencijalne metode za rješavanje problema gotovih duplikata HTML dokumenata je dan prijedlog kako bi se to pomoću eksperimenta moglo istražiti. Također je opisano koliko bi prostora za pohranu podataka bilo potrebno za spremanje HTML struktura u bazu podataka naspram koliko bi prostora za pohranu podataka bilo potrebno za spremanje svih potrebnih podataka već postojećeg rješenja upotrebe *simhasha* za rješavanje problema gotovih duplikata.

LITERATURA

[1] Mongodb documentation. https://docs.mongodb.com/. 14.6.2021.

[2] Pymongo 3.11.4 documentation, May 2021. URL https://pymongo.readthedocs.io/en/stable/index.html. [Online; accessed 2. Jul. 2021].

[3] Web Crawler 101: What Is a Web Crawler? (And How It Works), Mar 2021. URL https://www.webfx.com/blog/internet/what-is-a-web-crawler. [Online; accessed 2. Jul. 2021].

[4] Rodney Brazil. Search engine poisoning and how it can affect you - Namecheap Blog. *Namecheap Blog*, Jun 2021. URL https://www.namecheap.com/blog/search-engine-poisoning.

[5] Terence Eden, Scott O'Hara, Sangwhan Moon, Bruce Lawson, Patricia Aas, Xiaoqian Wu, i Shwetank Dixit. HTML 5.3. Technical report, W3C, Siječanj 2021. https://www.w3.org/TR/2021/NOTE-html53-20210128/.

[6] Sam Sneddon James Graham. html5lib-python. https://html5lib.readthedocs.io/en/latest/.

[7] Eve Maler, Jean Paoli, Tim Bray, François Yergeau, i Michael Sperberg-McQueen. Extensible markup language (XML) 1.0 (third edition). W3C recommendation, W3C, Veljača 2004. https://www.w3.org/TR/2004/REC-xml-20040204/.

[8] Gurmeet Singh Manku, Arvind Jain, i Anish Das Sarma. Detecting near-duplicates for web crawling. U *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, stranica 141–150, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936547. doi: 10.1145/1242572.1242592. URL https://doi.org/10.1145/1242572.1242592.

[9] Steven Pemberton. XHTML™ 1.0 the extensible hypertext markup language (second edition). W3C recommendation, W3C, Ožujak 2018. https://www.w3.org/TR/2018/SPSD-xhtml1-20180327/.

[10] Leonard Richardson. Beautiful soup documentation. `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`.

[11] Stephan Richter. lxml - xml and html with python. `https://lxml.de/index.html#documentation`. 21.3.2021.

[12] Paul Irish Tali Garsiel. How browsers work: Behind the scenes of modern web browsers. `https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/`. 5.8.2011.

[13] Yongsheng Zhu. W3C dom 4.1. WD not longer in development, W3C, Ožujak 2020. https://www.w3.org/TR/2020/NOTE-dom41-20200317/.

Detekcija ubačenog zlonamjernog HTML koda na temelju HTML tagova u Web stranici

Sažetak

Ovaj rad je kao glavni cilj imao prepoznavanje zlonamjernog HTML koda u web-stranicama. U svrhu tog cilja se pokušalo istražiti mogućnost detekcije zlonamjerno ubačenog HTML koda korištenjem samo HTML tagova i njihove strukture. Prije svega je opisano kako se vadi HTML struktura iz HTML dokumenta pomoću gotovih programskih podrška za parsiranje HTML dokumenata. Nakon što se HTML struktura uspješno izvuče iz dokumenta, opisan je algoritam za otkrivanje novo ubačenih HTML struktura nekog dokumenta. Algoritmu se treba predati najnovija verzija i druga najnovija verzija HTML dokumenta, te će algoritam kao rezultat vratiti sve otkrivene HTML strukture ako ih uspije otkriti. Uz pomoć tog algoritma je proveden eksperiment nad malim uzorkom dokumenata. Metoda koja je korištena u eksperimentu se pokazala nedovoljnom da ostvari cilj eksperimenta, te mogućnost detekcije zlonamjerno ubačenog HTML koda korištenjem samo HTML tagova i njihove strukture ostaje ne istražena. Za metodu su u ovom radu predložena unaprjeđenja uz koja bi metoda mogla biti dovoljna da istraži problem.

Drugi cilj ovog rada je istraživanje doprinosa promatranja HTML strukture u rješavanju problema skorih duplikata kod web-pretraživača (engl. *web crawler*). Za navedeni problem već postoji rješenje, te se u sklopu ovog rada je objašnjava kako bi trebao izgledati eksperiment koji bi mogao utvrditi je li promatranje HTML strukture potencijalna alternativa već postojećeg rješenja.

Ključne riječi: HTML, zloćudni kod, sigurnost, internet

Detecting injected malicious HTML code based on HTML tags in Web page

Abstract

The main goal of this paper was to identify malicious HTML code in web pages. For this purpose, the possibility of detecting inserted malicious HTML code using only HTML tags and their structure was investigated, and in general things about HTML documents and parsing HTML documents in general were explained. First of all, it is described how to extract HTML structure from HTML document using ready-made software for parsing HTML documents. After the HTML structure is extracted from the document, the algorithm for detecting newly inserted HTML structures of a document is explained. The latest version and the second latest version of the HTML document should be passed to the algorithm, and the algorithm will return all detected HTML structures as a result if it manages to detect any. Using this algorithm, an experiment was conducted on a small sample of documents. The method used in the experiment proved insufficient to achieve the goal of the experiment, and the possibility of detecting maliciously inserted HTML code using only HTML tags and their structure remains unexplored. For the method, improvements are proposed in this paper with which the method could be sufficient to investigate the problem.

Another goal of this paper is to investigate the contribution of observing the HTML structure in solving the problem of recent duplicates in the web browser. There is already a solution to this problem, and this paper explains what an experiment that could determine whether observing the HTML structure is a potential alternative to an existing solution should look like.

Keywords: HTML, malicious code, security, internet

8. Privitak: izvorni kod algoritma

U ovom privitku se nalazi izvorni kod algoritma za pronalazak ubačene HTML strukture opisanog u poglavlju 3.1 u programskom jeziku Python.

Listing 8.1: Izvorni kod algoritma

```
def structure_difference(oldNodes, newNodes):
    i = 0
    j = 0

    imax = len(oldNodes)
    jmax = len(newNodes)

    result = []
    insertedStruct = []

    state = 0

    while i < imax and j < jmax:

        iNode = oldNodes[i]
        jNode = newNodes[j]

        if state == 0:

            if iNode == jNode:
                state = 0
                i += 1
                j += 1
                continue
            else:
                state = 1
                continue

        elif state == 1:
```

```
30
31              if jNode == iNode:
32                  result.append(insertedStruct)
33                  insertedStruct = []
34
35                  state = 0
36                  i += 1
37                  j += 1
38                  continue
39              else:
40                  insertedStruct.append(jNode)
41
42                  state = 1
43                  j += 1
44                  continue
45
46      if i == imax and j == jmax:
47          return result
48      else:
49          return None
```