SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 328

Primjena genetskih algoritama u postupku otkrivanja propusta protokola

Branko Spasojević

Voditelj: doc. dr. sc. Marin Golub

Zagreb, lipanj, 2008

Sadržaj

1. Uvod 1
2. Otkrivanje sigurnosnih propusta 2
  2.1 Revizija sigurnosti 2
    2.1.1 Ručna revizija 2
    2.1.2 Automatizirana revizija 2
  2.2 Sigurnosni propust 3
    2.2.1 Prelijevanje spremnika 4
    2.2.2 Cjelobrojni preljevi 5
    2.2.3 Greške formatiranog ispisa 7
    2.2.4 Greške obrade 7
  2.3 Poslužitelj 8
  2.4 Protokol 8
    2.4.1 Sintaksa i semantika protokola 9
    2.4.2 Sjednica i paket 10
3. Primjena GA u analizi sigurnosti 11
  3.1 Rad genetskog algoritma 11
  3.2 Populacija 11
  3.3 Funkcija dobrote 11
    3.3.1 Dobrota pokrivenosti osnovnim blokovima 12
    3.3.2 Dobrota pokrivenosti funkcijama 12
    3.3.3 Dobrota heurističkom ocjenom pokrivenosti funkcija 12
    3.3.4 Dobrota jedinke 14
  3.4 Križanje 15
  3.5 Mutacija 16
  3.6 Selekcija 16
4. Rezultati ispitivanja 18
  4.1 Priprema testiranja 18
  4.2 GAzzy 19
  4.3 Ispitivanje rada sustava 20
  4.4 Nedostatci i buduća poboljšanja 25
5. Zaključak 27
6. Literatura 28
Sažetak 29

1. Uvod

Informatizacija društva uzrok je sve većem korištenju računala pri rješavanju svakodnevnih obaveza. Jedan od zahtjeva korisnika raznih usluga je mobilnost i dostupnost u svakom trenutku. Internet kao globalno dostupna mreža, pruža mogućnosti ostvarenja takvih zahtjeva, pa je veliki broj poslova moguće obaviti korištenjem mrežnih usluga. Danas je tako moguće plaćati račune, dogovarati poslove, kupovati stvari, od prehrambenih namirnica do automobila, i sve to preko Interneta. Kako Internet pruža samo infrastrukturu za prijenos podataka potrebno je razviti mehanizme tj. protokole koji će ostvariti komunikaciju korištenjem dostupne infrastrukture. Broj postojećih protokola nije poznat jer osim velikog broja javno dokumentiranih protokola, postoji i velik broj privatnih protokola, čije specifikacije nisu poznate.

Neželjena posljedica informatizacije društva je pružanje sučelja za pristup podacima svima s pristupom Internetu. To je navelo brojne pojedince da se pobliže upoznaju s detaljima implementacije mrežnih sustava i pronađu nedosljednosti koje dovode do propusta u sigurnosti. Sigurnosti na Internetu danas se posvećuje velika pažnja, i svaki kućni korisnik je upoznat s izrazima poput: virus i haker.

Kako bi se zaštitili od potencijalnih prijetnji i povećali razinu sigurnosti, softverske kompanije uvode ručne revizije sigurnosti. Cijena i vrijeme potrebno za ručno revidiranje koda ponekad nadilazi okvire mogućnosti i raspoložive resurse, stoga se sve veći broj kompanija okreće automatiziranom ispitivanju.

Profesor Barton Miller, 1989. godine na Wisconsin-Madison sveučilištu, razvija sustav ispitivanja nazvan *fuzzing[14]*. Prednost ovog načina testiranja je u iznimno jednostavnom dizajnu sustava za ispitivanje kojim je moguće provjeravati velik broj različitih aplikacija. Fuzzeri su vrsta *black-box* sustava za ispitivanje koji kao testne primjere koristi nasumično generirane nizove. Evolucijom veličine i kompleksnosti računalnih programa pojavila se potreba za proširenjem ovog jednostavnog modela testiranja.

Jedna od novih metoda testiranja sigurnosti je informirano ispitivanje korištenjem genetskih algoritama za poboljšanje tj. evoluciju testnih primjera. Ovim modelom je moguće poboljšati mogućnosti nasumičnog *fuzz* testiranja korištenjem heurističkog znanja dostupnog iz analize izvršnog koda programa, te dinamičkog znanja dostupnog praćenjem tijeka izvršavanja. Dosadašnja rješenja zbog niske razine (osnovni blokovi) promatranja nisu primjenjiva na stvarne programe. Naime, broj blokova koji se izvršavaju u produkcijskim aplikacijama je toliko velik da zbog naizgled malih vremenskih kašnjenja takvi sustavi za ispitivanje postaju neupotrebljivi. Predloženo rješenje je povećanje razine promatranja (funkcijska), te uvođenje heurističke evaluacije svake funkcije. Direktna posljedica personalizacije funkcija je veća količina informacija dostupna funkciji dobrote, te u konačnici preciznija evolucija.

2. Otkrivanje sigurnosnih propusta

Otkrivanje ili identificiranje sigurnosnih propusta sustava ili aplikacije moguće je obaviti na više načina, ovisno o tome obavlja li analizu računalni program ili čovjek osobno. Prvi način se naziva automatizirano otkrivanje propusta jer većim dijelom može funkcionirati neovisno o čovjeku, koristeći se predodređenim pravilima. Drugi način se naziva revizija sigurnosti (eng. *auditing*) koja se dijeli još na reviziju programskog koda i reviziju na strojnoj razini zvanu reverzni inženjering (eng. *reverse engineering*).

2.1 Revizija sigurnosti

2.1.1 Ručna revizija

Ideja ručne revizije sigurnosti programskog koda je identificiranje potencijalno ranjivih programskih konstrukcija, a trebala bi se prakticirati na razinama izrade programskog koda u modelu procesa programskog inženjerstva. Nedostaci u sigurnosti programske izvedbe mogu biti, ne samo vezani uz značajke programskog jezika izvedbe već i posljedica dizajna. Propuste dizajna teško je otkriti korištenjem automatiziranog testiranja, a postupci formalne verifikacije nisu uvijek dostupni i ostvarivi. Kao rješenje nameće se ručna analiza sigurnosti od strane čovjeka, pružajući mogućnost otkrivanja velikog spektra različitih ranjivosti. Iako čovjek naizgled predstavlja idealno rješenje problema, projekti od nekoliko milijuna linija programskog koda su vremenski prezahtjevni za ovakvu vrstu analize.

2.1.2 Automatizirana revizija

Kako bi se uklonili nedostaci ljudske revizije osmišljen je velik broj automatiziranih rješenja. Tri glavne kategorije s obzirom na razinu testiranja su: *black-box*, *gray-box* i *white-box*.

- *Black-box*: Ispitivanje koje se obavlja kroz dostupno sučelje bez uvida u unutarnji način rada programa. Također se istim imenom nazivaju i testiranja kod kojih nije dostupan izvorni programski kod već samo izvršna datoteka programa.
    - **Prednosti**: Može se koristiti za ispitivanje velikog broja različitih sustava ukoliko podržavaju slično pristupno sučelje. Relativno jednostavna implementacija ovakvog sustava.
    - **Nedostatci**: Ispitivanje nema uvid u promjene stanja programa, stoga je otežano ispitivanje specifičnih implementacijskih detalja.
- *White-box*: Ispitivanje se odvija uz poznavanje unutrašnjeg rada programa, te sustav za ispitivanje posjeduje veće znanje o radu aplikacije što ga čini sposobnijim u pronalasku propusta.
    - **Prednosti**: Sustav za ispitivanje se odnosi na određen program stoga je moguće testirati velik broj karakterističnih provjera za tu implementaciju.
    - **Nedostatci**: Moguće je testirati samo programe koji imaju identičnu unutarnju strukturu kao i program za kojeg je izgrađen sustav. Izrada ovakvih sustava zahtjevnija je od *black-box* tipa.

- *Gray-box*: Ispitivanje je mješavina prethodna dva načina gdje se kombinira slijepo *black-box* ispitivanje s vođenim ispitivanjem specifičnim za *white-box*.
    - **Prednosti**: Moguće preuzeti najbolje osobine prethodna dva tipa testiranja, te proširiti primjenjivost sustava na veći broj programa koristeći neka saznanja o unutarnjem radu svakog od njih.
    - **Nedostaci**: Potrebno je pažljivo balansirati omjer *black* i *white box* testiranja kako bi se postigli što bolji rezultati. Izrada ovakvog sustava tipično je zahtjevnija od *black-box* sustava.

2.2 Sigurnosni propust

Sigurnosni propust je naziv za nedostatak unutar programskog koda koji za posljedicu ima neželjeno ponašanje programa. To ponašanje može iskoristiti osoba koja ima pristup programu kako bi ostvarila svoj cilj koji je u suprotnosti s ispravnim funkcioniranjem programa.

Propuste je moguće podijeliti u skupine s obzirom u kojem su se ciklusu modela procesa programskog inženjerstva pojavili. U tom slučaju postoje:

- *Propusti dizajna* – svi oni problemi koji nastaju zbog greške ili previda prilikom dizajna programa. Takvi propusti rezultat su nepoznavanja sigurnosnih trendova i opasnosti. Primjer propusta dizajna je telnet protokol koji omogućava slanje korisničkih podataka preko nesigurnog kanala bez uporabe enkripcije.
- *Implementacijski propusti* – naziv je za propuste u načinu implementacije, tj. nepravilnim korištenjem mogućnosti programskog jezika implementacije. Primjer ovakvih propusta su prelijevanje spremnika (eng. *buffer overflow*), cjelobrojni preljevi (eng. *integer overflow*) i drugi.
- *Operacijski propusti* – odnose se na sigurnosne nedostatke nastale zbog okruženja u kojem se program izvodi. Primjeri ovakvih propusta su svi oni slučajevi kada programer pretpostavlja da određene operacije nisu dozvoljene, npr. kreiranje datotečnih poveznica (eng. *file links*) što može dovesti do *race condition* propusta.

Propusti koji će se razmatrati u ovom radu biti će iz implementacijske grupe jer pripadaju tipu propusta koje je najlakše otkriti. Naime implementacijski propusti najčešće za posljedicu imaju izazivanje iznimke (eng. *exception*) i rušenje programa, što je lako uočiti. Ostali tipovi propusta iako mogu imati za posljedicu rušenje programa (u tom slučaju se i oni mogu otkriti) daleko češće uzrokuju suptilne promjene u radu programa. Njih je moguće primijetiti jedino pažljivom analizom semantike stanja programa (npr. promjena privilegija izvođenja). Za otkrivanje takvih promjena potrebno je duboko razumijevanje rada programa i promjena stanja.

Implementacijski propusti čijem je otkrivanju posvećena posebna pažnja su: prelijevanje spremnika, cjelobrojni preljevi, greške formatiranog ispisa (eng. *format string*), pogreške obrade (eng. *parsing errors*).

2.2.1 Prelijevanje spremnika

Prilikom programiranja u jezicima sličnim C-u gdje programer ima izravan pristup memoriji, česti su slučajevi rezervacije prostora za pohranu podataka nedovoljne veličine. Korištenje funkcija koje ne koriste parametar maksimalne veličine odredišta rezultira prepisivanjem dijela memorije izvan željenog područja, što se popularno naziva prelijevanje spremnika. Prelijevanje spremnika se dijeli na dvije vrste ovisno o dijelu memorije na kojem se dogodio preljev.

Prvi slučaj je prelijevanje memorije na stogu. To je područje rezervirano za lokalne varijable čija je veličina unaprijed poznata prilikom pokretanja programa. Na stogu se osim programskih varijabli nalaze i neki dodatni podaci (vrijednosti nekih registara te argumenti funkcija) koje programer koristi implicitno. Procesor koristi neke od tih podataka za memoriranje toka izvršavanja. Dva značajna podatka na stogu su zapisi EIP (eng. *extended instruction pointer*) i EBP (eng. *extended base pointer*) registra. EIP zapis sadrži adresu na koju je potrebno vratiti tok izvršavanja nakon što se završi s trenutnom funkcijom, dok EBP posjeduje adresu okvira (eng. *frame*) koju je potrebno ažurirati završetkom trenutne funkcije. Ova dva zapisa su zanimljiva jer njihova promjena rezultira promjenom ponašanja programa. Naime kako te podatke koristi procesor, ukoliko njihove vrijednosti nisu ispravne, generira se iznimka koju će operacijski sustav zabilježiti i okončati rad programa. Ovakvo ponašanje je poželjno jer uvelike olakšava otkrivanje propusta.

```
void nesigurna_funkcija(char *argument)
{
    char buffer[256];
    strcpy(buffer, argument);
    ...
}
```

Slika 2.1 – Prelijevanje spremnika na stogu

Drugi slučaj je prelijevanje memorije na gomili (eng. *heap*). Taj dio memorije raspoloživ je programeru za dinamičku alokaciju prema potrebi. Preljevi na ovom dijelu se razlikuju od onih na stogu po tome što na gomili nisu pohranjeni podaci značajni za tok izvršavanja. Kako je gomila dinamički alocirana, za raspoznavanje rezerviranih područja koriste se različite programske strukture ovisno o operacijskom sustavu i korištenoj implementaciji alokatora. Zajednička osobina svih implementacija je pohrana informacijskih struktura između rezerviranih područja. Prepisivanjem tih struktura proizvoljnim podacima narušava se ispravnost alokatora koji će prilikom sljedeće alokacije korištenjem izmijenjenog zapisa generirati iznimku.

```
void nesigurna_funkcija2 (char *argument)
{
    char *buffer;
    if ( (buffer = malloc(256)) == NULL)
    {
        return;
    }
    strcpy(buffer, argument);
    ...
}
```

Slika 2.2 - Prelijevanje spremnika na gomili

Očigledan način za ispitivanje ranjivosti ulaznih parametara programa na prelijevanje spremnika je unos dugačkog niza znakova. Iako se tom metodom može otkriti veći broj nedostataka, postoje još neki slučajevi koje je također poželjno obuhvatiti testovima.

Preljev od nekoliko (eng. *off-by-one*, *off-by-few*) poseban je slučaj preljeva spremnika koji omogućava prelijevanje samo jednog ili pak svega nekoliko bajtova. Razlog tome je često zaboravljanje implicitnih definicija funkcija za rad s nizovima koje dodaju ili izostavljaju NULL (\x00) bajt na kraju niza.

```
void nesigurna_funkcija3 (char *argument)
{
    char buff[256];
    for (int i=0; i<256 && i<len(argument); i++)
        buff[i] = argument[i];
    buff[i] = '\0';
    ...
}
```

Slika 2.3 – Preljev od nekoliko

Za otkrivanje propusta ovog tipa korištene su sljedeće pretpostavke:

- Veličine alociranih spremnika potencije su broja dva (npr. 128, 256, 512, 1024...). Naime ovo je zaista istinito u velikom broju slučajeva, jer ovakve veličine omogućavaju lakšu optimizaciju pristupa i alokacije memorije.
- Preljev od nekoliko moguće je otkriti korištenjem ulaznog niza dužine potencije broja dva uvećanog ili umanjenog za nekoliko bajtova (npr. 126, 127, 128, 129, 130).
- Propust je pronađen ukoliko je program generirao iznimku kodnog broja 0xc000005, prestao reagirati na upite ili završio izvođenje prije vremena.

2.2.2 Cjelobrojni preljevi

Manje poznata klasa ranjivosti koja se odnosi na programske nedostatke nastale uslijed prelijevanja cjelobrojnih tipova podataka (char, short, int, long) u aritmetičkim operacijama. Cjelobrojni tipovi podataka mogu se koristiti za spremanje informacija o dužinama znakovnih nizova, ulaznih podataka ili prilikom alokacije memorije. U tim slučajevima potrebno je posvetiti posebnu pažnju aritmetičkim operacijama koje mogu vratiti neočekivan rezultat koji dovodi do sigurnosnih nedostataka.

```
int  nesigurna_funkcija4  (char  *buf1,  char  *buf2,
unsigned int duzina1, unsigned int duzina2)
{
    char mybuf[256];
    if((duzina1 + duzina2) > 256)
    {
        return -1;
    }

    memcpy(mybuf, buf1, duzina1);
    memcpy(mybuf + duzina1, buf2, duzina2);
    ...
    return 0;
}
```

Slika 2.4 – Cjelobrojni preljev

U primjeru je, na prvi pogled, ispravno napisana funkcija. Sadrži provjeru dužina argumenata (zaštita od preljeva spremnika) koji se kopiraju u lokalni spremnik. Navedena provjera ranjiva je na cjelobrojni preljev zbroja argumenata: *duzina1* i *duzina2*. Prilikom evaluacije izraza *duzina1 + duzina2* rezultat se implicitno pretvara (eng. *casting*) u tip unsigned int. Problem nastaje ukoliko zbroj prelazi maksimalnu vrijednost unsigned int

varijable koja iznosi *2^32-1*, nakon čega rezultat postaje *(duzina1 + duzina2) % 2^32-1*. Ako primjerice *duzina1*=2^32-1, a *duzina2*=1 *duzina1*+*duzina2*=0 čime će se izraz *if((duzina1 + duzina2) > 256)* evaluirati kao lažna, i zaobići zaštitu protiv preljeva spremnika. Konačno u funkciji `memcpy` dolazi do preljeva spremnika zbog pokušaja kopiranja većeg niza od veličine mybuf-a.

Slični propusti još su opasniji u mrežnim protokolima jer je moguće izravno mijenjati vrijednosti polja dužine, neovisno o stvarnoj dužini argumenta. Također čest propust u mrežnim protokolima, odnosi se na pretvorbu iz brojčanih tipova s predznakom (eng. *signed*) u one bez (eng. *unsigned*) i obratno. Ukoliko se kao argument funkciji, koja prima nepredznačen tip, preda negativan broj implicitna pretvorba će broj zapisan dvojnim komplementom preslikati u pozitivnu interpretaciju tog zapisa i rezultirati brojem visoke vrijednosti.

```
int procitaj_korisnicke_podatke( int sockfd
)
{
    int duzina, sockfd, n;
    char buffer[1024];
    duzina = dohvati_duzinu( sockfd );
    if( duzina > 1024 )
    {
        return 1;
    }
    if( read( sockfd, buffer, duzina ) < 0)
    {
        error("read: %m");
        return 1;
    }
    return 0;
}
```

Slika 2.5 – Cjelobrojni preljev

Nepažnja prilikom rukovanja brojčanim argumentima korisnika može neočekivano zaobići postavljene sigurnosne provjere. U prethodnom primjeru negativa vrijednost varijable *duzina* zaobići će provjeru maksimalne duljine spremnika, ali prilikom pretvorbe u pozitivan broj kao argument read funkcije postat će višestruko veći od rezerviranog spremnika. Rezultat ovog naizgled bezazlenog previda je preljev spremnika.

Razlika između cjelobrojnih preljeva i preljeva spremnika je u načinu testiranja. Iako je dio cjelobrojnih preljeva posljedica dugačkog ulaznog niza, propusti kao primjer *2-4* ovise o brojčanim vrijednostima. Prilikom izrade testnih primjera poželjno je ubaciti provjere koje mogu ispitati takve propuste, tj. neke tipične pozitivne i negativne brojeve, a korištene su sljedeće pretpostavke:

- Većinu cjelobrojnih preljeva moguće je otkriti dugim nizom znakova ili umetanjem posebno osmišljenim brojevima.
- Veća je vjerojatnost izazivanja pogreške korištenjem rubnih slučajeva brojeva nego očekivanim „normalnim“ vrijednostima.
- Preferirane predodređene testne vrijednosti su: 0, 1, 10, 100, 0xffff, 0x80000000, 0xfffffffa, 0xfffffff.

2.2.3 Greške formatiranog ispisa

Ova klasa ranjivosti je usko vezana uz funkcije formatiranog ispisa, tipične za C jezik, koje neispravnim korištenjem dovode do sigurnosnog propusta. Propusti su posljedica izostavljanja formata ispisa uslijed čega zlonamjerni korisnik može umetnuti svoj format ispisa. Formatiranje ispisa se može koristiti kako u svrhu čitanja, tako i u svrhu pisanja po memoriji. Za pisanje se može iskoristiti opcija *%n* koja zapisuje broj ispisanih znakova na ekran u memorijsku adresu predanu kao argument. Ovaj način formatiranja posebno je pogodan za ispitivanje ranjivosti na ovu klasu pogreške, jer nasumično zapisivanje podataka u memoriju ima za posljedicu nestabilnost i rušenje testiranog programa.

```
int log_error(char *fmt, ...)
{
   char buf[BUFSIZ];
   va_list ap;
   va_start(ap, fmt);
   vsnprintf(buf, sizeof(buf), fmt, ap);
   va_end(ap);
   syslog(LOG_NOTICE, buf);
}
```

Slika 2.6 – Greška formatiranog ispisa

Funkcije potencijalno ranjive na propuste formatiranja su: *printf, fprintf, sprintf, snprintf, vfprintf, vprintf, vsprintf, vsnprintf, syslog, scanf, fscanf*. Za potrebe otkrivanja grešaka pretpostavljeno je sljedeće:

- Vjerojatnost rušenja programa koji sadrže ovaj tip propusta u ovisnosti o argumentima poredana je ovim padajućim redoslijedom: *%n, %s, %f, %d, %x*.
- Vjerojatnost pojave ovog tipa sigurnosnog propusta manja je od ostalih spomenutih propusta (prelijevanje spremnika, cjelobrojnih preljeva i pogrešaka obrade).

2.2.4 Greške obrade

Prilikom promatranja strukture programa moguće je uočiti više različitih logičkih cjelina. Tako najčešće postoji: dio korisničkog sučelja (eng. *graphical user interface*), inicijalizacijski dio, dio zapisivanja informacija (eng. *logging*), dio za obradu te mnogi drugi. Programski kod za obradu (eng. *parsing*) zadužen je za razlaganje i raspoznavanje naredbi iz određenog strukturiranog zapisa, bio on u obliku datoteke ili mrežnog protokola. Obrada se odvija raspoznavanjem niza bajtova pridjeljujući im značenje. Veće mogućnosti i sloboda zapisa zahtijevaju veću pažnju i broj stanja programskog koda za obradu. Posljedica je povećanje mogućnosti pogreške, pogotovo u jezicima sličnim C-u gdje je semantika varijabli izuzetno važna i sadrži brojne implicitne pretvorbe koje lako dovode do sigurnosnih propusta. Za razliku od prethodno spomenutih propusta koji su usko vezani uz određenu programsku konstrukciju jezika, ova klasa propusta podrazumijeva puno širi spektar konstrukcija, tj. sve one koje za posljedicu imaju pogrešku obrade. Primjer ove klase propusta je pogreška zbrajanja „<>“ znakova u obradi adrese „MAIL FROM“ ili „RCPT TO“ zahtjeva SMTP protokola kod *Sendmail* programskog paketa[1]. Navedeni primjer ima kao posljedicu krive

[1] http://www.securityfocus.com/archive/1/313757

obrade „<>“ znakova prelijevanje spremnika koje se može iskoristiti za preuzimanje kontrole nad poslužiteljem. Iako je sigurnosni propust posljedica prelijevanja spremnika, svrstan je u klasu pogrešaka obrade zbog preduvjeta koje je potrebno zadovoljiti za izazivanje pogreške.

Ispitivanje ovih propusta se također razlikuje od prethodno navedenih, a temelji se na mutaciji sintaksnih cjelina (polja) protokola. Mutacija je naziv za proces u kojem se ispravni članovi (jedan ili više znakova) ulaznog niza razmještaju, umnožavaju i uklanjaju kako bi se testirao što veći broj mogućih interpretacija i rubnih slučajeva dijela koda za obradu. Za potrebe otkrivanja grešaka pretpostavljeno je sljedeće:

- Pogreške obrade vezane su uz obradu neispravnih sintaksnih konstrukcija sličnih ispravnim.
- Testne primjere moguće je generirati iz ispravnih konstrukcija korištenjem pravila za mutaciju.
- Veća je vjerojatnost pogreške obrade specijalnih znakova (eng. *delimiters*) nego korisničkih vrijednosti.
- Pogreške obrade dovode do preljeva spremnika i cjelobrojnih varijabli.

2.3 **Poslužitelj**

Poslužitelj je naziv za program koji pruža usluge korištenjem mrežnog sučelja. Pristup mrežnom sučelju najčešće je ostvaren preko priključnice (eng. *socket*) koja omogućava slanje i primanje podataka. Za komunikaciju koriste jedan ili više protokola. Za razliku od programa njihov rad podrazumijeva komunikaciju s više različitih osoba ili programa, što ga čini izloženijim napadima zlonamjernih pojedinaca. Mogućnost kompromitiranja čitave mrežne infrastrukture iskorištavanjem sigurnosnog propusta unutar jednog poslužitelja čini ih iznimno vrijednima za napadača, dok korporacije izdvajaju velik novac kako bi što kvalitetnije osigurali svoje poslužitelje od napada. Rezultat ove utrke je velik broj dostupnih sustava za ispitivanje mrežnih poslužitelja i protokola. Na žalost, većina njih temelji se na *black-box* modelu koji nije dovoljan za iscrpno ispitivanje kompleksnih sustava, zbog čega raste potreba za *gray-box* rješenjima koja omogućuju ispitivanje velikog broja različitih poslužitelja i protokola uz minimalne preinake.

2.4 **Protokol**

Protokol predstavlja konvenciju ili standard koji osigurava konekciju, komunikaciju i razmjenu podataka između dva ili više krajeva. Primjer standarda kojim se opisuje protokol je RFC (eng. *Request for Comments*) koje izrađuju stručnjaci IETF-a (*Internet Engineering Task Force*). Protokole je moguće grupirati u nekoliko skupina s obzirom na njihovu sintaksu i semantiku. Jedna od takvih podjela je na otvorene (eng. *clear text protocols*) i kriptirane (eng. *encrypted protocols*). Otvorene protokole simbolizira njihova čitljivost i pristup izvornim podacima u slučaju kada prisluškujemo (eng. *eavesdropping*) komunikacijski kanal. Kriptirani protokoli ne dijele tu osobinu, pa ukoliko prisluškujemo takav kanal nismo u mogućnosti raspoznati semantiku razgovora. Zbog ovakvih osobina otvoreni protokoli omogućuju iskorištavanje prisluškivanja za učenje nepoznatih inačica protokola i na taj način proširiti primjenjivost sustava za ispitivanje sigurnosti i na nepoznate protokole. Primjeri

otvorenih protokola koji posjeduju ova zanimljiva svojstva su: SMTP, POP3, IMAP, LDAP, HTTP i ostali.

2.4.1 Sintaksa i semantika protokola

Prilikom izrade općenitog sustava za ispitivanje protokola važno je proučiti načinjene pretpostavke i njihov utjecaj na cjelokupni proces. Sljedeće pretpostavke su korištene prilikom izrade i analize protokola prezentiranog sustava:

- Klijent šalje poslužitelju jednu ili više poruka.
- Svaka poruka sadrži jednu ili više naredbi i može sadržavati proizvoljne podatke.
- Svaka naredba je odvojena od korisničkih podataka graničnikom (eng. *delimiter*) koji može biti jedan ili više znakova.
- Korisnički podaci mogu biti zapisani kao slova, brojevi (*ascii*) ili brojevi (veličine 8, 4, 2, 1 bajtova).
- Korisnički podaci su neovisni o odgovoru poslužitelja.

Primjenom ovih pretpostavki omogućeno je korištenje postojećeg znanja o protokolu tj. predodređenih heuristika. Kako bi se mogao testirati nepoznati protokol potrebno je poznavati njegovu sintaksu. Ručno proučavanje i opisivanje protokola sustavu za ispitivanje je nepraktično za potrebe adaptivnog sustava. Prisluškivanje sjednice poslužitelja i klijenta nekog protokola smatra se izvedivo zbog pretpostavke da se radi o otvorenim protokolima. Ovime se otvara mogućnost snimanja primjera (jednog ili više) sjednica, te njihovo korištenje kao predloška za daljnje ispitivanje. Jedan očigledan nedostatak ovog pristupa je međuovisnost kvalitete[2] zabilježene sjednice i pokrivenosti[3] (eng. *coverage*) testiranja servisa. Ukoliko je zabilježen velik broj različitih mogućnosti protokola, sustav će biti u stanju i testirati te slučajeve dok u suprotnom sustav neće biti obaviješten o njihovom postojanju.

Graničnici

Kako bi se razdvojile naredbe protokola od podataka većina otvorenih protokola koristi neki oblik graničnika. Njihova uloga je sintaksno odvajanje individualnih polja protokola. Ukoliko se pretpostavi postojanje graničnika, moguće je heuristički razgraničiti paket na dijelove i proglasiti ih poljima. Promjene na razini polja smanjuju vjerojatnost generiranja sintaksno neispravnih paketa protokola. Ovo je važna pretpostavka jer se sintaksna analiza protokola odvija prije obrade podataka, stoga ukoliko testni primjer bude odbačen zbog neispravne sintakse neće se moći obaviti ispitivanje. Lista graničnika proizvoljna je i može se proširiti novim heuristikama. Korišteni graničnici su: '#', '@', '/', '\\', '%', '^', '&', '*', '(', ')', '_', '+', '!', '$', '%', '[', ']', '{', '}', '|', '"', '"', ':', '?', ';', '-'.

---

[2] Kvaliteta sjednice promatra se obzirom na broj različitih ključnih riječi (tj. mogućnosti) protokola.

[3] Pokrivenost je naziv za često korištenu metriku testiranja sustava. Kako nije moguće testirati sigurnost na sve moguće ulaze, umjesto ocjene postupka testiranja obzirom na broj provjera, promatra se količina koda izvršena njihovom obradom. Dvije najčešće razine na kojima se prati pokrivenost su: funkcijska i razina osnovnih blokova (eng. basic blocks).

2.4.2 Sjednica i paket

Komunikacija između dvije krajnje točke započinje uspostavom veze. Nakon uspješno uspostavljene veze moguća je izmjena podataka sve dok jedna strana ne zatvori svoju priključnicu. Komunikacija od početka do zatvaranja priključnice naziva se sjednica (eng. *session*) protokola. Podaci sjednice šalju se u obliku paketa, koji predstavljaju građevnu jedinicu sjednice. Pakete je moguće promatrati kao logičke cjeline, gdje svaki paket sadrži naredbu protokola, podatak ili oboje. Prilikom testiranja nepoznatih protokola korištene su sljedeće pretpostavke:

- Sjednica se sastoji od tri glavna dijela: inicijalizacija konekcije, prijenos podataka, zatvaranje veze.
- Paketi na početku sjednice pripadaju inicijalizacijskom dijelu.
- Paketi u sredini sjednice sadrže naredbe protokola i podatke.
- Posljednji paket sjednice služi zatvaranju veze.
- Moguće je kombinirati pakete različitih sjednica i stvoriti novu ispravnu sjednicu poštujući zahtjev za postojanjem inicijalizacijskih paketa na početku sjednice i paketa za završetak sjednice na kraju.
- Postoje paketi ovisni o redoslijedu pojave, i potrebno ih je poštivati prilikom izgradnje novih sjednica.
- Genetski algoritam neispravnim sjednicama dodijeliti će malu vrijednost dobrote i na taj način neizravno poštivati pravila redoslijeda i ostale zahtjeve strukture protokola.

3. Primjena GA u analizi sigurnosti

Black-box sustavi analize testiraju sigurnost sustava predajući programu nasumično generirane podatke preko sučelja. Ukoliko program kao ulaz prima niz znakova maksimalne dužine 100 znakova, broj kombinacija koje je potrebno generirati kako bi se u potpunosti testirao ispravan rad programa na sve moguće ulaze iznosi 256^100. Kako stvarni programi posjeduju više od jednog ulaza koji često prihvaćaju puno dulje nizove, ovakav način iscrpnog testiranja je nepogodan za primjenu.

Genetski algoritmi su heuristička metoda pretraživanja velikog prostora rješenja i kao takvi posjeduju mogućnost primjene na problem izgradnje sustava za ispitivanje sigurnosti.

3.1 Rad genetskog algoritma

```
Evoluiraj:
  populacija = Generiraj_populaciju()
  za svaku jedinku iz populacije:
    jedinka.dobrota = Evaluiraj_dobrotu(jedinka)
    dok broj_mutacija > 0:
      jedinka.dobrota  =  1./veličina_populacije  *  Evaluiraj_dobrotu(
jedinka.Mutiraj() )
      broj_mutacija -= 1
  populacija = Selekcija(populacija)
```

Slika 3.1 – Pseudokod genetskog algoritma

3.2 Populacija

Skup potencijalnih rješenja problema predstavlja populaciju. Rješenja u postupku testiranja sigurnosti predstavljena su sjednicama koje demonstriraju propust. Populacija sustava za ispitivanje sastoji se od predložaka sjednica (jedinki), koje će se koristiti prilikom testiranja protokola.

Jedan od postavljenih zahtjeva za ispitivanje protokola je postojanje uhvaćenih stvarnih sjednica. Te sjednice posebno su važne jer predstavljaju svo dostupno znanje o sintaksi i semantici protokola. Iz tog razloga čuvaju se kao posebna populacija predaka i koriste se prilikom generiranja novih predložaka testiranja.

Osobine značajne za populaciju su:

- Broj jedinki u populaciji.
- Početna populacija.
- Prosječna i maksimalna dobrota jedinki.

Stvaranje početne populacije podudara se s postupkom selekcije stoga će biti opisano u poglavlju *Selekcija*.

3.3 Funkcija dobrote

Kako bi genetski algoritam uspješno usmjeravao ispitivanje potrebno je osmisliti takvu funkciju dobrote koja će nagrađivati one testove koji su vjerojatniji izazvati neželjenu

posljedicu, a ispravne konstrukcije preskočiti. Postojeći sustavi temeljeni na GA koriste kao funkciju dobrote pokrivenost koda na razini osnovnih blokova i funkcija. Iako ovakva metrika upućuje na kvalitetu testnih primjera, posjeduje neke ozbiljne nedostatke za primjenu na velik broj različitih programskih sustava.

3.3.1 Dobrota pokrivenosti osnovnim blokovima

Prilikom testiranja dobrote temeljene na osnovnim blokovima uočeni su neki ozbiljni nedostatci ove metode. Prilikom razvoja idejnog modela sustava za ispitivanje ne posvećuje se dovoljna pažnja primjenjivosti modela za izradu funkcionalnih sustava i podrške stvarnih aplikacija. Naime, iako ovaj model funkcije dobrote sadrži veliku količinu informacija i daje dobre rezultate prilikom ispitivanje demonstrativnih aplikacija, njegova primjena je teško izvediva za programske pakete korporativnih aplikacija. Kao primjer je moguće promotriti IBM Informix bazu podataka koja posjeduje 500,000 osnovnih blokova i 130,000 funkcija. Održavanje referenci i praćenje izvođenja ovako velikog broja blokova pokazalo se u praksi neizvedivo korištenjem postojećih desktop računala nove generacije (više jezgreni procesori s 2+ Gb radne memorije) i dostupnih platformi za razvoj i praćenje toka izvršavanja.

Osim praćenja izvršavanja blokova korištenjem prekidnih točaka (eng. *breakpoints*), Intel procesori podržavaju postavljanje procesora u stanje[4] praćenja naredbi grananja. Iako nešto grublja granulacija toka izvođenja još uvijek sadrži sve informacije kao i metoda osnovnih blokova. Iako po prirodi puno brža tehnika praćenja izvođenja, ni ova metoda nije pogodna za računanje dobrote zbog sporosti. U programima s malim brojem skokova ova metoda je daleko brža od tradicionalnog praćenja (eng. *tracing*), ali prilikom testiranja velikih programa s dugim tokom izvođenja, suma kašnjenja pri svakom skoku nagomila se dovoljno da onemogući normalan rad programa.

3.3.2 Dobrota pokrivenosti funkcijama

Probleme praćenja izvršavanja na razini osnovnih blokova moguće je djelomično riješiti korištenjem praćenja izvršavanja funkcija. Ideja je i dalje ista, ocjenjivanje dobrote testova brojem izvršenih funkcija. Nedostatak je očigledan gubitak količine informacija zbog osobina funkcija. Funkcijsko programiranje je nastalo težnjom izdvajanja logičkih cjelina u posebne dijelove. Prilikom testiranja nije poznato koje funkcije su zadužene za obradu protokola, a koje za prikaz rezultata na grafičko sučelje. Ovo ima nepoželjne posljedice za funkciju dobrote. Testni primjeri nerelevantni za ispitivanje sigurnosti mogu poprimiti veću vrijednost dobrote ukoliko pozivaju veći broj funkcija, čak i ako nisu logički vezane uz interesne cjeline obrade podataka (npr. funkcije za grafičke prikaz). Drugi nedostatak izjednačavanja funkcija je zanemarivanje njihovih metrika kao što su instrukcije, veličina, vrijeme potrebno za izvršavanje, broj ulazno/izlaznih parametara i druge. Navedene osobine čine ovu metodu nepogodnom za korištenje u funkciji dobrote.

3.3.3 Dobrota heurističkom ocjenom pokrivenosti funkcija

Zbog nemogućnosti praćenja izvođenja na nižim razinama od funkcijske, potrebno je ispraviti njene nedostatke. Najveći nedostatak je izjednačavanje svih funkcija. Predloženo

---

[4] MSR branch tracing - http://download.intel.com/design/processor/manuals/253669.pdf

rješenje je individualizacija funkcija na temelju njihovih osobina koje se heuristički boduju. Na taj način svaka funkcija dobiva vrijednost koja odgovara njenom potencijalu testiranja, tj. vjerojatnosti da sadrži sigurnosni propust. Izvorni programski kod nije dostupan i sva analiza se izvodi na razini strojnog koda (eng. *assembly language*). Odabrana platforma testiranja zbog svoje zastupljenosti i dostupnosti je Intel x86, a sva analiza strojnog koda prilagođena je tom razredu procesora.

Prelijevanje spremnika može biti posljedica korištenja nesigurnih funkcija kopiranja ili ručnim kopiranjem memorije unutar petlje. Pozivanje nesigurnih funkcija otkriva se pretragom funkcijskih blokova za instrukcijom *call*. Svakoj *call* instrukciji analizira se argument tj. ime poziva, i ako se ime nalazi na popisu potencijalno opasnih funkcija, dobrota se povećava za 6. Funkcije čiji pozivi se nagrađuju su: `_strcpy`, `_strncpy`, `_memset`, `_memcpy`, `_sprintf`, `_snprintf`, `_strcmpi`, `_strcmp`, `_strncmp`, `_strncmpi`, `_strcat`, `_strncat`, `lstrcpyA`, `lstrcmpA`, `wsprintfA`, `__strdup`, `_memcmp`. Ukoliko pozivana funkcija nije na popisu, dobrota se također povećava, ali za 1. Razlog povećanja i u drugom slučaju je nagrađivanje povezanosti funkcije. Što je funkcija više povezana, veća je vjerojatnost da će testni primjeri koji dohvate (testiraju) tu funkciju, dohvatiti i određen broj pozvanih funkcija i time povećati pokrivenost koda. Za otkrivanje petlji koristi se jednostavna provjera odredišne adrese skokova. Ukoliko je adresa skoka manja (niža) od adrese instrukcije skoka, pretpostavlja se povratak na prethodni kod tj. skok petlje. Ukoliko je prepoznata petlja, dobrota funkcije uveća se za 10.

Jedna od naznaka mogućnosti cjelobrojnih preljeva su implicitne pretvorbe varijabli, dijeljenja/množenja predznačenih i nepredznačenih brojeva. Na razini strojnog koda ove slučajeve je moguće otkriti pretragom za instrukcijama[13] *movsx*, *movzx*, *sar*, *shr*, *sal, shl*, *idiv*, *div*, *imul* i *mul*. Sve navedene instrukcije doprinose dobroti s 1. Korištenje navedenih instrukcija u matematičkim formulama ne predstavlja prijetnju i u tom slučaju trebalo bi zanemariti njihovu važnost. Zbog pretpostavke testiranja poslužitelja moguće je zanemariti postojanje tog slučaja i osloniti se na pretpostavku da takvi programi ne rade intenzivne matematičke izračune. Iako se matematičke kalkulacije intenzivno koriste u enkripcijskim[5] protokolima, zbog zahtjeva otvorenosti protokola možemo zanemariti taj slučaj.

Greške formatiranog ispisa nisu uključene u vrijednost dobrote zbog zahtjevne analize strojnog koda. Ovaj tip propusta moguće je otkriti analizom argumenata funkcije. Ukoliko je jedini argument korisnički definiran niz znakova tada postoji velika mogućnost postojanja ranjivosti. Argumenti se predaju funkciji preko stoga u obliku željene vrijednosti ili pokazivača na dio memorije. Kako na strojnoj razini nisu sačuvane informacije o granicama varijabli u memoriji, raspoznavanje sadržaja memorijske adrese zahtijeva unazadno praćenje korištenja adrese argumenta i analizu mjesta korisničkog pridruživanja vrijednosti. Kako navedeni postupak nije u mogućnosti uvijek dohvatiti željene informacije, a identifikacija ranjivosti je trivijalna testnim primjerom, odlučeno je korištenje testne metode.

[5] Prilikom testiranja protokola koji podržavaju enkripciju, spomenuta tehnika identifikacije matematičkih instrukcija može poslužiti kao dobar heuristički mehanizam identifikacije enkripcijskih funkcija. Kod takvih protokola poželjno je detaljno testirati te funkcije zbog njihove važne uloge u sigurnosti cjelokupnog protokola.

Dodatne nagradne vrijednosti funkcija dobiva za korištenje sljedećih instrukcija: relativno adresirajuća `mov`[6], `lea`[7], `stos`, `lods`[8]. Postojanje instrukcija doprinosi dobroti za 1.5.

Posljednje obilježje koje doprinosi dobroti funkcije je njena kompleksnost koja se promatra preko broja osnovnih blokova. Definicija osnovnih blokova nameće ovaj kriterij kao pogodan za aproksimaciju kompleksnosti funkcije, te doprinositi dobroti sljedećom formulom:

*dobrota_funkcije += broj_blokova_funkcije * μ* *μ - heuristička procjena uloge kompleksnosti funkcije* (μ = 0.1)	(3.1)

3.3.4 Dobrota jedinke

Funkcija dobrote je ostvarena korištenjem heurističke procjene osobina funkcije. Kao što je prije napomenuto, jedinke ne predstavljaju većinske testne primjere, nego samo predložak testiranja. Iz tog razloga, osim dobrote (potencijala) predloška, u dobroti jedinke je zastupljena i dobrota svakog testnog primjera generiranog iz nje. Omjer zastupljenosti dobrote predloška i dobrota testova generiranih iz predloška je 50:50. To znači da se dobrota svakog testnog primjera generiranog iz predloška množi s inverzom broja generiranih testova i dodaje ukupnoj vrijednosti dobrote jedinke.

ukupna_dobrota = 0
za svaku pogođenu funkciju predloška:
  ukupna_dobrota += dobrota_funkcije
za svaki generirani test iz predloška:
  *ukupna_dobrota += 1.0 / broj_generiranih_testova * dobrota_generiranog_testa*
	(3.2)

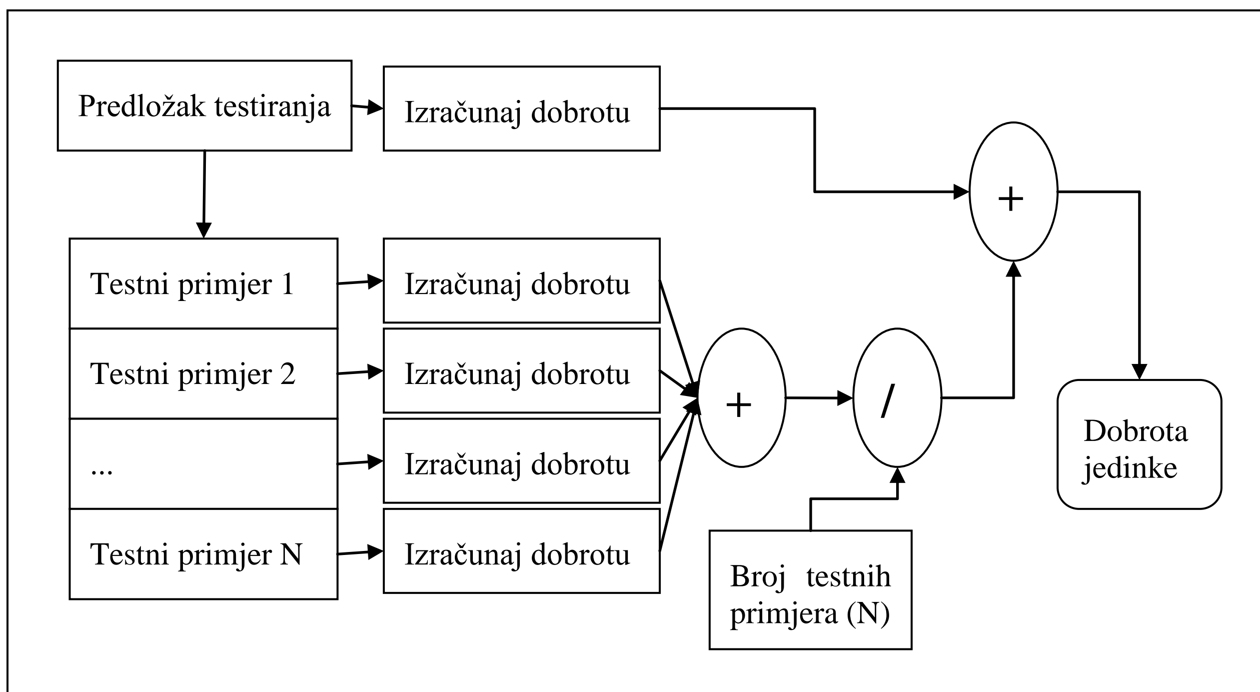
Slika 3.2 – Funkcija dobrote

Ideja ovakve ocjene dobrote je osiguravanje da predlošci s nižom dobrotom, ukoliko omogućavaju generiranje iznimno dobrih testova, nadmaše svoju heurističku dobrotu i kompetitivno sudjeluju u procesu selekcije.

[6] Instrukcija oblika *mov imm16/32|reg16/32, [base + disp]* ili *mov [base + disp], reg16/32* koriste se za relativno adresiranje unutar struktura, dok se strukture koriste za interpretaciju protokola. Stoga postojanje ovih instrukcija navodi da se radi o kodu obrade protokola.

[7] *Lea* (eng. load effective address) instrukcija dohvaća adresu varijable. Iako se može koristiti i u druge svrhe, najčešća je uporaba za dohvat adrese elementa strukture i zbog toga se vrednuje kao i mov naredba.

[8] *Stos* i *lods* (eng. store/load string) instrukcije se koriste za spremanje i učitavanje niza znakova. To ih čini kandidatima za ispitivanje na prelijevanje spremnika ili pogreške obrade.

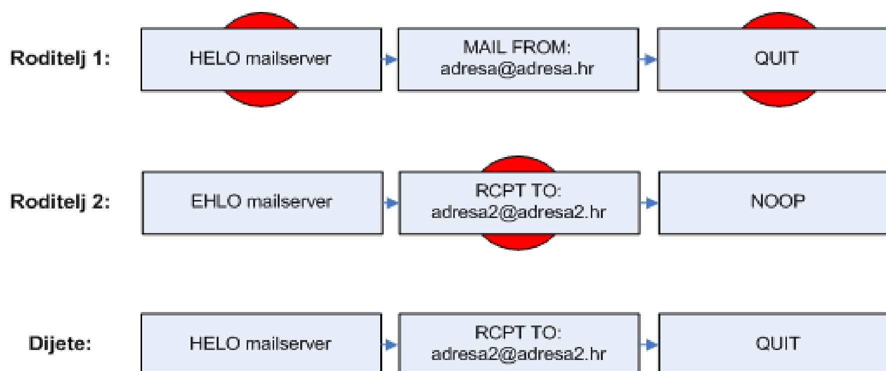


Slika 3.3 – Dobrota jedinke

3.4 Križanje

Križanje je genetski operator koji radi na principu zamjene genetskog materijala roditelja. Prilikom križanja odabire se *n* prekidnih točaka u genetskom lancu roditelja, između kojih dijete nasumično nasljeđuje gene jednog od roditelja. Analogija kromosoma u mrežnim protokolima je sjednica, dok su geni paketi. Ukoliko poslužitelj zamislimo kao automat stanja ovisan o naredbama protokola, paketi predstavljaju ulaze stanja. Pokrivenost koda proporcionalna je broju dohvatljivih stanja, ali i redoslijedu promjene stanja. Naime, prilikom promjena stanja može doći do nedosljednosti promjene zavisnih varijabli i uzrokovati sigurnosne probleme. Zbog toga je posvećena posebna pozornost pretraživanju prostora različitih puteva kroz poslužitelj, tj. različitih redoslijeda paketa unutar sjednice.

Križanje pruža jednostavan način pretraživanja prostora stanja automata zamjenom paketa roditeljskih sjednica i izgradnjom novih testnih primjera. Kombiniranjem naredbi disjunktnih skupova, izgrađene sjednice testiraju nove puteve i osiguravaju kvalitetno ispitivanje semantike protokola.

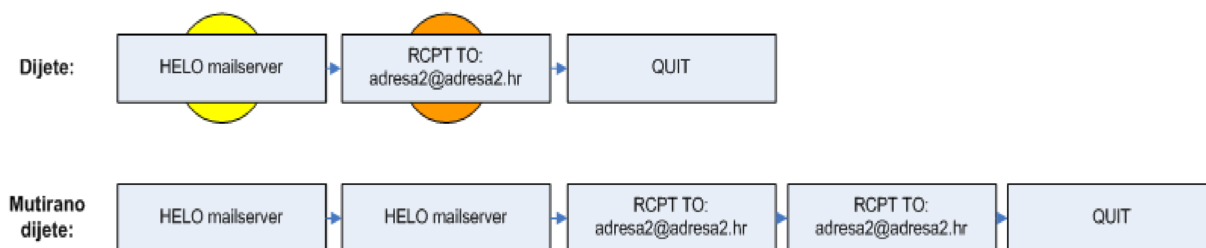


Slika 3.4 – Križanje

3.5 Mutacija

Mutacija je naziv za promjene u genetskoj strukturi čiji je zadatak doprinijeti raznolikosti. Prisutne su dvije idejno različite izvedbe operatora mutacije.

Mutacija sjednica je prva, i odnosi se na mutaciju sjednica za vrijeme križanja. Prilikom ispitivanja pogrešaka obrade, poželjno je promotriti ponašanje poslužitelja na pojavu identičnih paketa i omogućiti drugoj razini mutacije bolje ispitivanje. *Mutacija sjednica* je postupak kojim se paketi odabrani u procesu križanja mogu umnožiti proizvoljan broj puta. Za svaki paket nasumično se odlučuje o primjeni operatora mutacije i duljine mutiranog niza. Važno je primijetiti kako ovako izgrađene sjednice ne predstavljaju testne primjere već samo populaciju koja će primjenom druge razine mutacije postati testni primjeri.



Slika 3.5 - Mutacija sjednice

Mutacija paketa je druga razina mutacije. Odnosi se na izmjenu sadržaja paketa, i zaslužna je za izradu testnih primjera. Na ovoj razini postoji više različitih izvedbi operatora mutacije, od kojih je svaki zaslužan za ispitivanje jedne ili više skupina ranjivosti. Šest je osnovnih operatora mutacije koji se primjenjuju i svi tretiraju pakete kao nizove znakova proizvoljne dužine:

- Zamjeni – vrši se zamjena dvaju nizova.
- Zamjeni bajt – mijenja se vrijednost proizvoljnog bajta.
- Izmjeni – mijenja se vrijednost podataka (bajtova) između dvije točke.
- Umnoži graničnik – pojava graničnika u nizu zamijeni se njegovom umnoženom inačicom.
- Umnoži između graničnika – podaci između dva graničnika zamijene se njihovom umnoženom inačicom.
- Zanemari – zanemari izvođenje operatora mutacije i ostavi originalan niz.

3.6 Selekcija

Selekcija je postupak kojim se usmjerava evolucija populacije. Nekoliko je stvari kojima je potrebno posvetiti pozornost prilikom dizajna postupka selekcije kako bi se izbjegli problemi lokalnog optimuma ili divergencije od rješenja. Tijekom testiranja potrebno je osigurati konstantan dotok novih testnih primjera i izbjeći dugotrajna jednolična testiranja.

Kao metoda selekcije odabrana je eliminacijska turnirska selekcija. Ova selekcija održava n turnira u kojem sudjeluje m jedinki. Vjerojatnost odabira jedinke za sudjelovanje u turniru jednaka je za cijelu populaciju. Nakon odabira m jedinki za sudjelovanje u turniru, odbacuje

se jedinka najniže vrijednosti dobrote. Odbačena jedinka zamjenjuje se nasumičnim odabirom jedne od 3 reprodukcijske metode: *kloniranje*, *križanje* i *kombiniranje*.

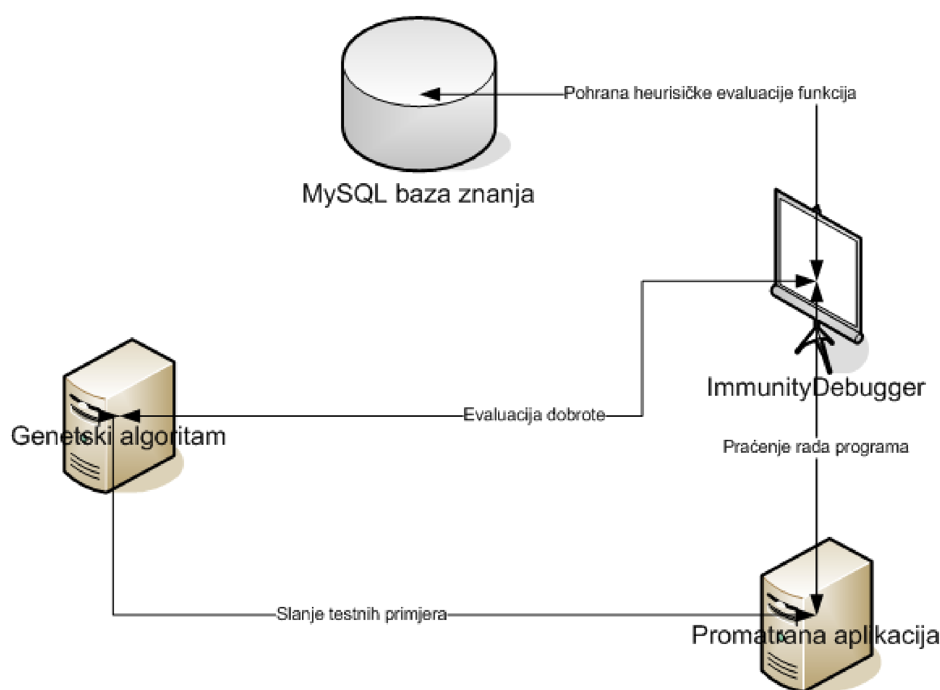
Kloniranje je metoda kojom se jedinke iz populacije predaka kopiraju kao predložak za ispitivanje. U slučajevima kada je sintaksa protokola kompleksna ova tehnika osigurava način generiranja ispravnih predložaka za ispitivanje.

Križanje je metoda kombiniranja genetskog materijala (paketa) dviju jedinki iz populacije predaka. Sve jedinke imaju jednaku vjerojatnost da budu odabrane, a odabir roditeljskih gena vrši se nasumično. Osim križanja ova metoda omogućava i nasumičnu primjernu *selekcijske mutacije* koja omogućuje umnožavanje proizvoljnog paketa.

Kombiniranje je metoda slična križanju, ali puno slobodnijih zahtjeva. U kombiniranju može sudjelovati proizvoljan broj jedinki predaka i prilikom odabira genetskog materijala (paketa) nije potrebno poštivati zahtjev linearnosti. Prilikom izgradnje nove jedinke nasumično se odabire podskup skupa svih paketa odabranih predaka, gdje redoslijed elemenata u podskupu određuje redoslijed paketa u sjednici.

4. Rezultati ispitivanja

Za potrebe testiranja osmišljenog sustava testiranja izrađen je skup aplikacija koje demonstriraju njegov rad. Dva su glavna međusobno nezavisna dijela sustava za ispitivanje: sustava za praćenje rada testirane aplikacije i sustav genetskog algoritma. Njihova interakcija opisana je sljedećom slikom:



Slika 4.1 – Interakcija sustava za ispitivanje

4.1 Priprema testiranja

Prije početka testiranja potrebno je sakupiti podatke o strukturi programa, izračunati heurističke vrijednosti funkcija te postaviti prekidne točke (eng. *breakpoints*) na početak promatranih funkcija.

Promatranje toka izvršavanja obavlja se upotrebom Immunity Debuggera[15] platforme. Odabrana je ova platforma jer pruža programerima korištenje python sučelja za pristup funkcijama *debuggera*, te izradu vlastitih dodataka. Sustav za praćenje toka izvršavanja i izračun dobrote testnih primjera implementiran je kao python skripta.

Nakon odabira poslužitelja koji će se testirati potrebno je stvoriti bazu podataka funkcija pozivom skripte !*gamon 1*. Ova naredba će stvoriti novu bazu podataka imena izvršne datoteke promatranog poslužitelja i popuniti ju informacijama o adresama funkcija, adresama osnovnih blokova funkcije, te pripada li funkcija dinamičkoj biblioteci (eng.

dynamic link library, *dll*) ili izvršnoj datoteci[9]. Primjer poziva skripte i njenog ispisa ilustriran je slikom *4-3*.

Sljedeći korak je popunjavanje tablice dobrotama heurističkih vrijednosti. Analiza strojnog koda zahtjevan je posao ukoliko *disassembler* ne posjeduju odgovarajuću razinu apstrakcije. Kako Immunity Debugger ne pruža zadovoljavajuću razinu apstrakcije strojnog koda, za potrebe izračuna dobrote funkcija korištena je IDA[16] (Interactive Disassembler). IdaFitness je python dodatak za IDA-u koji nakon pokretanja popunjava tablicu funkcija vrijednostima dobrote.

Nakon uspješno popunjene dobrote, potrebno se priključiti Immunity Debuggerom na željeni poslužitelj te učitati vrijednosti dobrote u bazu znanja *debuggera* pozivanjem skripte *!gamon 2*. Posljednji korak je postavljanje prekidnih točaka na sve funkcije izvršnog programa pozivom skripte *!gamon 3*. Prilikom okidanja prekidnih točaka debugger će zabilježiti adrese prekida u bazu znanja kojoj se na zahtjev može pristupiti. Ovime su završene pripreme testiranja i moguće je započeti s radom genetskog algoritma.

4.2 **GAzzy**

Genetski sustav za ispitivanje (GAzzy) odvojen je od sustava za praćenje izvršavanja, što omogućava mobilnost na odvojeno računalo u slučaju velikih procesorskih zahtjeva ili implementaciju drugačijeg algoritma generiranja testnih primjera.

Nakon pokretanja Gazzy sustava potrebno je unijeti ime datoteke u kojoj su pohranjene uhvaćene sjednice protokola koji se želi testirati. Dodatna mogućnost je odabir imena datoteke u koju će se pohraniti generirane sjednice (testni primjeri), i koristit prilikom analize pronađene ranjivosti ili kao predložak za novo ispitivanje.

```
C:\Documents and Settings\Desktop\GAzzy>GAzzy.py
wHELL come!
Input sessions file, empty line for end
Sessions xml file> SMTP_9-5-2008-19-22-7_C.xml
ret- {0: 'HELO mailserver\r\n', 1: 'MAIL FROM: dabar@dabrovina.com\r\n', 2: 'RCPT
TO: none
@noone.com\r\n', 3: 'DATA\r\n', 4: 'asdasda\r\n', 5: 'asdasda\r\n', 6: '\r\n', 7:
'.\r\n',
 8: '\r\n', 9: 'NOOP\r\n', 10: 'QUIT\r\n'}
Generated sessions:  {0: ['HELO mailserver\r\n', 'MAIL FROM:
dabar@dabrovina.com\r\n', 'RC
PT TO: none@noone.com\r\n', 'DATA\r\n', 'asdasda\r\n', 'asdasda\r\n', '\r\n',
'.\r\n', '\r
\n', 'NOOP\r\n', 'QUIT\r\n']}
Aboriginal population:  {0: ['HELO mailserver\r\n', 'MAIL FROM:
dabar@dabrovina.com\r\n',
'RCPT TO: none@noone.com\r\n', 'DATA\r\n', 'asdasda\r\n', 'asdasda\r\n', '\r\n',
'.\r\n',
'\r\n', 'NOOP\r\n', 'QUIT\r\n']}
Logfile prefix string> logout1
127.0.0.1 25 logout16-6-2008-0-32-45
Population random choice>  1
```

[9] Informacija o pripadnosti funkcije značajna je za bolju kontrolu razine promatranja izvođenja. Iako su sva testiranja vršena ignorirajući funkcije dinamičkih biblioteka, postoje slučajevi kada je poželjno uključiti ispitivanje i ovih funkcija. Jedan takav slučaj je kada poslužitelj zbog modularnosti zapakira dio funkcija kao dll-ove i korsiti pozivanjem u izvršnoj datoteci.

```
['HELO mailserver\r\n', 'MAIL FROM: dabar@dabrovina.com\r\n', 'RCPT TO:
none@noone.com\r\n
', 'DATA\r\n', 'asdasda\r\n', 'asdasda\r\n', '\r\n', '.\r\n', '\r\n', 'NOOP\r\n',
'QUIT\r\
n']
```

Slika 4.2 – Ispis sustava za ispitivanje

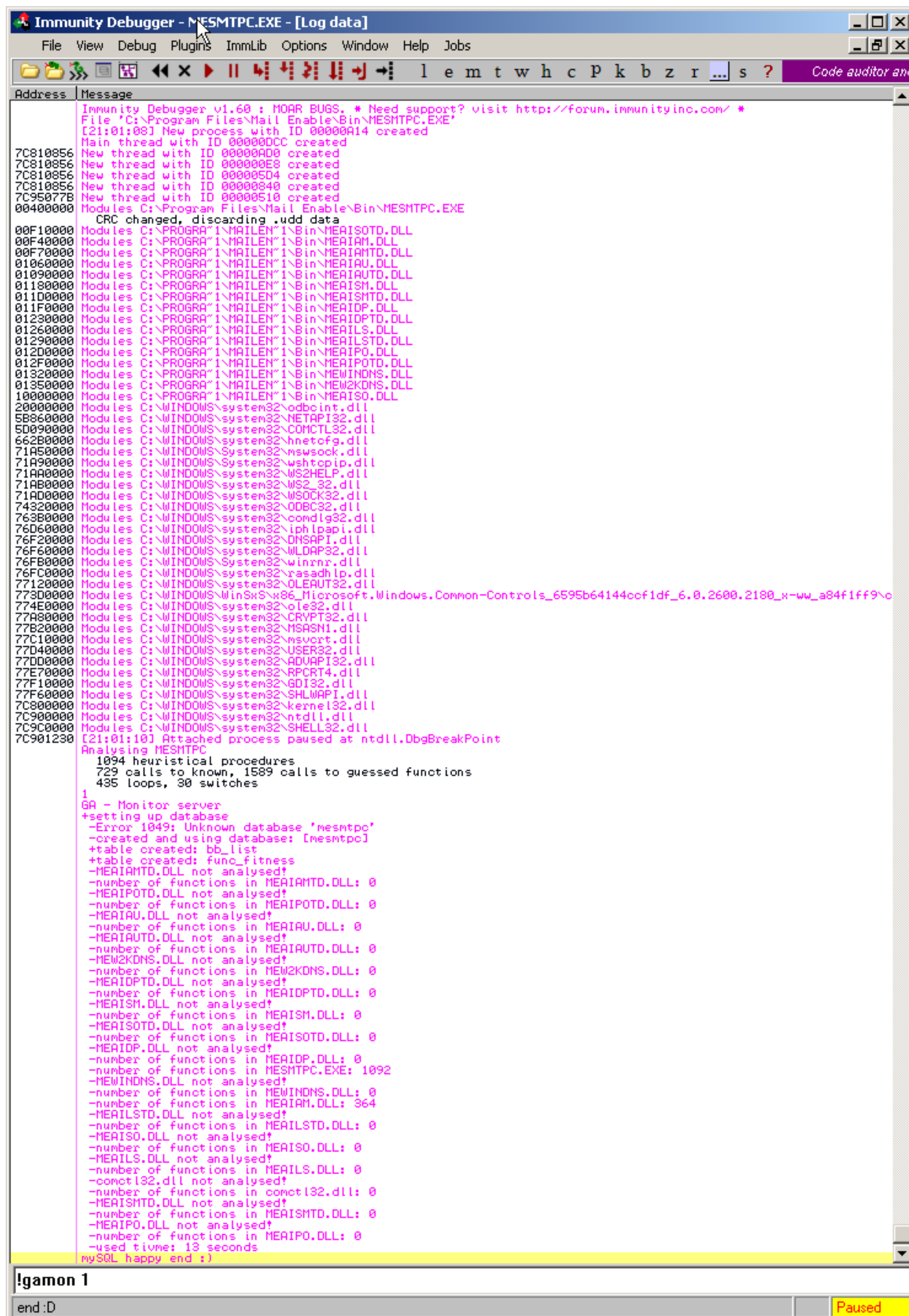
Za vrijeme testiranja pojavljuju se različiti testni primjeri čije vrijednosti dobrote imaju raspon od 2000.0 za 100 pogođenih funkcija, do 1500.0 za 220 pogođenih funkcija. Jedna od pretpostavki, da neispravni testni primjeri pogađaju manji broj funkcija, ispostavila se ispravna. Iako su neispravni testni primjeri ostvarili mali broj pogođenih funkcija, zbog činjenice da prilikom neispravne sjednice poslužitelj bilježi veći broj grešaka, takve sjednice su često postizale zamjetne vrijednosti dobrote (3000-6000). Kako bi se ispravilo nagrađivanje, smanjene su nagrade za pozivanje funkcija ispisa, tj. smanjene su nagrade za potencijal testiranja grešaka ispisa. Kako se taj tip pogreške posredno ispituje u procesu mutacije, a takve greške su rijetke, smanjenje nagrada bitno ne utječe na mogućnost otkrivanja grešaka ispisa.

Prilikom testiranja rada sustava uočene su brojne greške u Immunity Debuggeru. Osobine grešaka su takve da uslijed dužeg testiranja dolazi do rušenja programa i nemogućnosti nastavka. Iz tog razloga nije bilo moguće detaljnije testirati evoluciju testnih primjera na većem broj epoha.

4.3 Ispitivanje rada sustava

Za potrebe ispitivanja rada razvijenog sustava instaliran je *Mail Enable*[10] poslužitelj elektroničke pošte. Promatrani poslužitelj podržava nekoliko različitih protokola, a za potrebe testiranja odabran je SMTP protokol. Izvršna datoteka zadužena za obradu ovog protokola je MESMTPC.exe. Na slici 4.3 moguće je uočiti kako su pronađene 1092 funkcije unutar izvršne datoteke poslužitelja. Vrijeme potrebno za pronalaženje i upis potrebnih informacija u bazu podataka iznosi 13 sekundi. Vrijeme potrebno za izračun vrijednosti dobrote funkcija i upis u bazu podataka iznosi 6 sekundi.

[10] http://www.mailenable.com/



Slika 4.3 – Inicijalizacija sustava

Statistika broja funkcija s obzirom na ostvarenu dobrotu dana je na slici 4.4. Mali broj funkcija velike dobrote je očekivan, jer je dobrota proporcionalna veličini funkcije. Uz to, i funkcijsko programiranje pretpostavlja razlaganje velikih funkcija na manje, što omogućava lakše recikliranje koda.

```
mysql> select count(*) from func_fitness where fitness < 50;
+----------+
|      896 |
+----------+
mysql> select count(*) from func_fitness where fitness < 100 and fitness > 50;
+----------+
|      104 |
+----------+
mysql> select count(*) from func_fitness where fitness < 200 and fitness > 100;
+----------+
|       62 |
+----------+
mysql> select count(*) from func_fitness where fitness < 300 and fitness > 200;
+----------+
|       16 |
+----------+
mysql> select count(*) from func_fitness where fitness < 500 and fitness > 300;
+----------+
|        5 |
+----------+
mysql> select count(*) from func_fitness where fitness > 500;
+----------+
|        7 |
+----------+
```

Slika 4.4 – Distribucija vrijednosti dobrote

Znanje, tj. predložak protokola, zabilježen je u datoteci SMTP_9-5-2008-19-22-7_C.xml i biti će korišten za generiranje testnih primjera.

```
['HELO mailserver\r\n']
['MAIL FROM: dabar@dabrovina.com\r\n']
['RCPT TO: none@noone.com\r\n']
['DATA\r\n']
['asdasda\r\n']
['asdasda\r\n']
['\r\n']
['.\r\n']
['\r\n']
['NOOP\r\n']
['QUIT\r\n']
```

Slika 4.5 – Predložak protokola

Za parametre genetskog algoritma odabrane su sljedeće vrijednosti:

```
Veličina populacije = 10
Broj ispitivanja po jedinki = 10
Broj turnira = 3
Broj jedinki koje sudjeluju u turniru = 3
```

Slika 4.6 – Parametri genetskog algoritma

Prilikom ispitivanja generirano je 8 ispitnih primjera koji su poslani poslužitelju. Broj pogođenih funkcija i dobrota ispitnih primjera dan je na sljedećoj slici, dok je prikaz rada sustava za praćenje tijekom evaluacije dobrote prikazan slikom 4.8:

Broj pogođenih funkcija	*Dobrota ispitnog primjera*
208	9916
120	3739
121	3402
135	4101
134	3933

122	3571
134	4187
172	6268

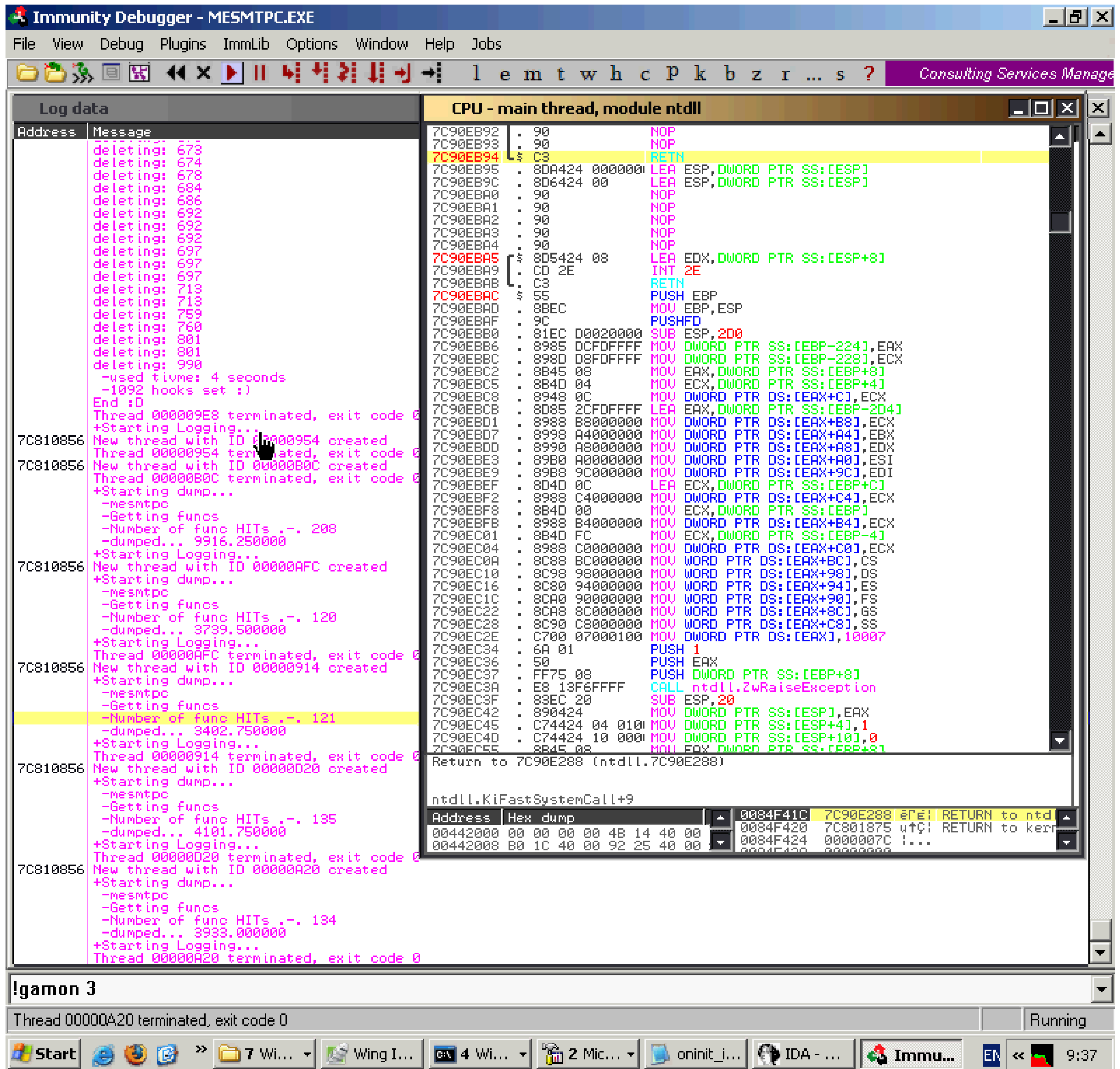
Slika 4.7 – Podaci ispitivanja

Prvi ispitni primjer predstavlja predložak ispitivanja čijom mutacijom nastaju ostali ispiti. Predložak je nastao *kloniranjem* ispravne sjednice protokola iz populacije predaka. Kako je ovaj ispit valjan, tj. ne sadrži neispravne konstrukcije, ne čudi što je osvojio najveću dobrotu.

Vidljivo je kako različiti testni primjeri pogađaju različit broj funkcija što je proporcionalno ostvarenoj dobroti. Ipak, pretpostavka da su neke funkcije zanimljivije od drugih rezultira slučajem kada testni primjer koji je pogodio 120 funkcija ima dobrotu 3793, dok testni primjer koji je pogodio 121 funkciju ima dobrotu 3402. Razlika ovih dvaju testnih primjera je u putevima kroz program. Put koji je ostvario veću dobrotu posjeduje veći potencijal testiranja, te će biti preferiran od strane genetskog algoritma. U slučaju da ne postoji evaluacija puteva, sustav za ispitivanje ne bi mogao razlikovati puteve, pa bi duži putevi uvijek dominirali evolucijom.

Nekoliko ispitnih primjera prikazano je na slici 4.9. Polje „*Mutation fitness*" predstavlja dobrotu mutirane jedinke, dok „*Fitness*" predstavlja ukupnu dobrotu predloška. Najčešće primjenjivani operatori mutacije bili su *zamjeni bajt* i *umnoži graničnike.*

Nakon slanja 8 ispitnih primjera došlo je do rušenja sustava za praćenje, *Immunity Debuggera,* prikazano slikom 4.10. Uz ovaj, pronađeno je još nekoliko nedostataka koji onemogućavaju provođenje ispitivanja željenog poslužitelja.



Slika 4.8 – Ocjena dobrote u sustavu za praćenje

```
C:\WINDOWS\system32\cmd.exe - GAzzy.py
Sending...  'DA\xff\xff\xff\xff'
Sending...  'asdasd\xff\xff\xff\xff\xff\xff'
Sending...  'asdasda\xff\xff'
Sending...  '\r\n'
Sending...  '.\r\n'
Sending...  '\r\xff'
Sending...  '\xff\xff\xffP\r\n'
Sending...  'QUIT\r\xff'
Read >> SUCCESS
Mutation fitness:  3402.75
--Fitness>  10273.3625
Read >> SUCCESS
127.0.0.1 25
Sending...  'HELO mailse\xff\xff\xffr\r\n'
Sending...  'MAIL FROM: dabar@dab\xff\xff\xff\xff\xff\xff.com\r\n'
Sending...  'RCPT T\xff\xff\xff\xff\xff\xff\xff@noone.com\r\n'
Sending...  '\xff\xff\xff\xff\xff\n'
Sending...  'asda\xff\xff\xff\xff\xff\xff\xff\xff'
Sending...  'asdas\xff\xff\xff\xff'
Sending...  '\r\xff'
Sending...  '.\xff\xff'
Sending...  '\r\n'
Sending...  '\xffOOP\r\n'
Sending...  'QU\xff\xff\xff\xff'
Read >> SUCCESS
Mutation fitness:  4101.75
--Fitness>  10478.45
Read >> SUCCESS
127.0.0.1 25
Sending...  'HELO m!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!
A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!A!
A!A!A!A!A!A!A!A!A!A!A!A!A!Aailserver\r\n'
Sending...  'MAIL FROM: dabar@dabrovin\xff\xff\xff\xff\xff\xff\n'
Sending...  'RCPT TO: none@noone.com\r\n'
Sending...  'DA\xff\xff\xff\xff'
Sending...  'asda\xff\xff\xff\xff\xff\xff\xff\xff'
```

Slika 4.9 – Ispitni primjeri

Immunity Debugger, 32-bit analysing debugger

Immunity Debugger, 32-bit analysing debugger has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to us. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Send Error Report | Don't Send

Slika 4.10 – Propust Immunity Debuggera

4.4 Nedostatci i buduća poboljšanja

Neki od uočenih nedostataka i prijedlozi rješenja su:

- Ograničenost na ispitivanje samo naredbi koje se nalaze u snimljenim predlošcima protokola.
    - Moguće rješenje ovog problema krije se u pretpostavci da svaka naredba protokola može sadržavati maksimalno jedan korisnički definiran argument. Naime, u tom slučaju moguće je heurističkim pretraživanjem programskog koda izdvojiti sve konstantne nizove znakova koji predstavljaju potencijalnu naredbu i iskoristiti genetski algoritam da otkrije koji su nizovi zaista naredbe.

- Nisu poznati graničnici protokola.
  - Velik broj graničnika moguće preuzeti iz javno dostupne dokumentacije otvorenih protokola. Ipak, prilikom ispitivanja je moguće naići na protokol čiji se graničnici razlikuju od unaprijed pretpostavljenih graničnika. U tom slučaju se vrši ispitivanje na puno višoj razini apstrakcije, što onemogućava kvalitetno ispitivanje. Uvođenjem automatskog prepoznavanja potencijalnih graničnika može se smanjiti opasnost od pre visoke apstrakcije sintakse protokola.
- Još uvijek velika generalizacija funkcija.
  - Iako značajno smanjena, generalizacija funkcija još uvijek je visoka. Uvođenjem dobrote svake funkcije uvedena je prva razina sintaksne diskriminacije funkcija. Na ovoj razini funkcije se razlikuju obzirom na sadržaj procesorskih instrukcija što daje naslutiti način obrade podataka, ali ne i semantiku tj. cilj obrade. Tako na primjer nije moguće razlikovati funkcije za obradu protokola i iscrtavanje korisničkog sučelja. Uvođenje semantičke diskriminacije je moguće asocijacijom funkcijskih poziva. Ukoliko se pretpostavi da funkcije za obradu protokola moraju pozvati *read()* na mrežni priključak (eng. *socket*), takve funkcije se mogu smatrati mrežno orijentirane. Asocijacijskim nasljeđivanjem svojstava, sve funkcije određene udaljenosti od mrežno orijentirane funkcije dobivaju koeficijent pripadnosti tom skupu. Izgradnjom ovakve mreže moguće je bolje diskriminirati funkcije.
- Proširenje mogućnosti na ispitivanje obrade datoteka.
  - Velik broj ranjivosti pojavljuje se prilikom obrade različitih datotečnih formata npr. mp3, pdf, swf, doc. Zbog zajedničkih osobina koje datotečni format dijeli s mrežnim protokolom, promjenom genetskih operatora moguće je proširiti primjenjivost genetskog sustava za ispitivanje.

5. Zaključak

Ispitivanje sigurnosti uporabom genetskih algoritama pokazalo se prominentnim. Osobine genetskih algoritama pogodne su za rješavanje problema generiranja sjednica nepoznatih protokola, što je jedan od glavnih problema primjenjivosti raznih sustava za ispitivanje. Dobrota heurističkom procjenom pokrivenosti funkcija pokazala se ne samo primjenjivom na velike aplikacije, već kao precizniji način ocjene dobrote testnih primjera uz manje zahtjeve. Funkcija dobrote važan je čimbenik genetskog algoritma koji usmjerava proces evolucije, stoga poboljšanje ove funkcije ima povoljnu posljedicu, bolje ispitivanje.

Prednost *black-box* sustava za ispitivanje bila je jednostavnost i brzina implementacije. Genetski algoritmi uklonili su velik dio programskog koda prisutnog kod tradicionalnih gray-box sustava zaduženog za generiranje testnih primjera, razumijevanje protokola i usmjeravanje procesa ispitivanja. Predstavljen *gray-box* sustav neznatno je programski zahtjevniji od black-box sustava, zadržavajući sve dobrobiti i prednosti *gray-box* sustava.

Iako *white-box* sustavi pružaju mogućnosti za najbolje ispitivanje, ukoliko se nastavi s evolucijom ovakvih sustava njihov značaj će se smanjiti. Mogućnost praćenja dinamike programa i prilagođavanje nepoznatim aplikacijama uklanja sve negativne osobine *white-box* sustava, dok u isto vrijeme zadržava sve pozitivne.

6. Literatura

[1] K. Gopalratnam, S. Basu, J. Dunagan, H. J. Wang. *Automatically Extracting Fields from Unknown Network Protocols*

[2] W. Cui, J. Kannan, H. J. Wang. Discoverer: *Automatic Protocol Reverse Engineering from Network Traces*

[3] N. Borisov, D. J. Brumley, H. J. Wang. *Generic Application-Level Protocol Analyzer and its Language*

[4] W. Cui, V. Paxson, N. C. Weaver, R. H. Katz. *Protocol-Independent Adaptive Replay of Application Dialog*

[5] S. Sparks, S. Embleton, R. Cunningham, C. Zou. *Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting*

[6] J. D. DeMott, R. J. Enbody, W. F. Punch. *Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing*

[7] P. McMinn, M. Holcombe. *Evolutionary Testing of State-Based Programs*

[8] P. McMinn, M. Holcombe. *Evolutionary Testing Using an Extended Chaining Approach*

[9] A. Baresel, H. Sthamer. *Evolutionary Testing of Flag Conditions*

[10] X. Liu, M. Zhang, Z. Bai, L. Wang, W. Du, Y. Wang. *Function Call Flow based Fitness Function Design in Evolutionary Testing*

[11] Y. Cheon, M. Kim. *A Fitness Function for Modular Evolutionary Testing of Object-Oriented Programs*

[12] M. Sutton, A. Greene, P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[13] *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, Intel Press, 2003

[14] Fuzz testing, http://en.wikipedia.org/wiki/Fuzz_testing

[15] Immunity Debugger, http://www.immunitysec.com/products-immdbg.shtml

[16] IDA Pro, http://www.hex-rays.com/idapro/

[17] Python, http://www.python.org/

Sažetak

Primjena genetskih algoritama u postupku otkrivanja propusta protokola

Kombinacijom pažljivo osmišljene heuristike i genetskih algoritama za sakupljanje znanja, moguće je konstruirati efikasne sustave za ispitivanje sigurnosti. Dodatna mogućnost koju pružaju ovakvi sustavi su ispitivanje nepoznatih protokola korištenjem malog broja predložaka sjednica za učenje. Razlika u odnosu na dosadašnje implementacije genetskih algoritama je bodovanje dobrote potencijalom funkcija. Heurističko bodovanje sintakse funkcije na strojnoj razini koristi se kao diskriminirajući faktor prilikom ocjene značaja ostvarene pokrivenosti koda testnim primjerom.

Ključne riječi: Genetski algoritmi, ispitivanje sigurnosti, mrežni protokoli, pokrivenost koda

Application of genetic algorithms in protocol vulnerability detection

By carefully combining heuristics and genetic algorithms used for knowledge mining it is possible to construct efficient systems for security testing. An additional capability of these systems is the ability to test unknown protocols using a small number of session templates for learning. The difference between this and current implementations of genetic algorithms is the scoring of fitness based on function potential. Heuristic scoring of function fitness on an assembly level is used as a discriminatory factor while evaluating the significance of the achieved test case code coverage.

Key words: Genetic algorithms, security testing, network protocols, code coverage