0. Kako čitati ovaj rad?

Za stjecanje dojma o samoj metodi preskočite uvod, pročitajte potpoglavlja 1.1. i 1.2. – bez potpotpoglavlja 1.2.1 i 1.2.2, samo 1.2.3. Za bolji uvid pročitati sva potpotpoglavlja od 1.2. Za najbolji uvid pogledati i opis programskog ostvarenja (poglavlje 1.3.).

Za dodatna pitanja javite mi se na bilo koji servis na kojem me pronađete (github, mail, linkedin, facebook, ...).

1. Praktični rad

Mirai je zloćudni program koji napada uređaje interneta stvari s Linux operacijskim sustavom. Ranjive uređaje skuplja u botnet mrežu te koordinira raspodijeljeni napad uskraćivanjem usluga (engl., *Distributed Denial of Service*, skr., *DDoS*). Na ranjive uređaje prenosi program *bot* koji ima ulogu klijenta u *Command And Control* arhitekturi. Bot na zapovijed napadača izvodi DDoS napad, skenira mrežu u potrazi za novim žrtvama te ima mogućnost brisanja svoje izvršne datoteke zbog čega ostavlja minimalan trag u sustavu. Uređaje napada pogađajući njihovu IP adresu te pogađajući korisničko ime i lozinku za servise *SSH* i *telnet*. Kombinacije koje isprobava su one koje su najčešće korištene ili one koje su postavljene kao zadane na popularnijim uređajima interneta stvari – poput najprodavanijih IP kamera i usmjeritelja (engl., *router*). Napad na tvrtku DYN 2016. godine prouzročio je pad većine popularnih servisa poput GitHuba, Twittera, Reddita, Netflixa i AirBnBa, a napad na tvrtku OVH 2016. godine je generirao promet veličine 1 Tbps [5]. Procijenjeno je da je najviše zaraženih uređaja u jednom trenutku bilo oko 600 000 [5].

Autor je izvorni kôd programa javno objavio što je uzrokovalo pojavu varijanti zloćudnih programa koji danas tvore familiju Mirai zloćudnih programa. Varijante napadaju širi spektar uređaja i arhitektura, pa primjerice varijante *SORA* i *UNSTABLE* napadaju sustave za pohranu snimki nadzornih kamera [8], a *Okiru* napada posebno procesore s ARC arhitekturom.

U ovom radu opisan je postupak generalizacije i klasifikacije zloćudnih programa familije *Mirai* korištenjem rudarenja podataka. Opisana je teorija postupka te je opisana napravljena implementacija, odnosno pronalaženje zajedničkih svojstava svim programima iz familije.

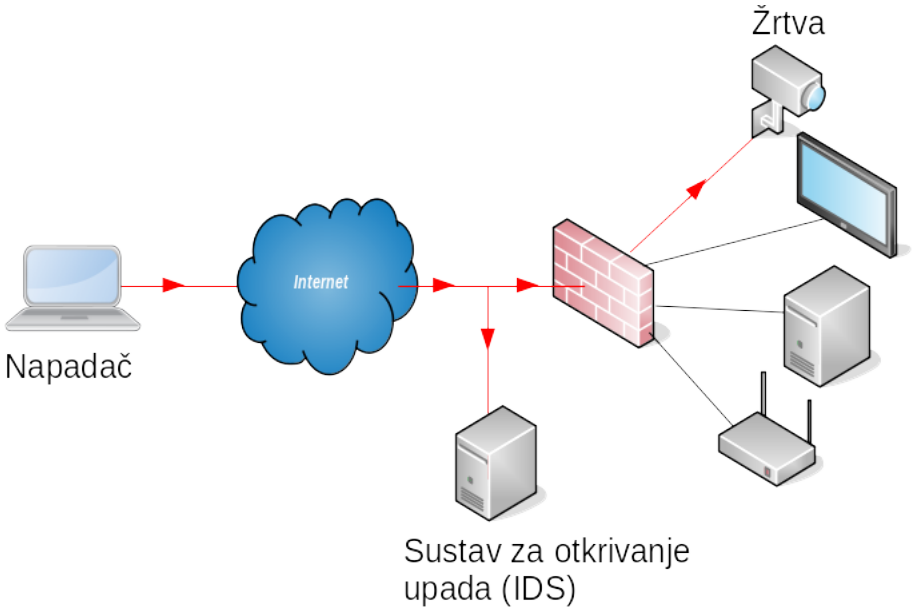
1.1. Definicija problema otkrivanja familije

Zbog formiranja familije, odnosno skupa zloćudnih programa s istim svojstvima i nastalim iz istog izvornog koda, postavljen je problem generalizacije familije.

Promatrajući općenita svojstva familije Mirai, postavljena su temeljna ograničenja rješenja problema:

1. zbog brisanja izvršne datoteke programa, teško je pronaći virus skeniranjem trajne memorije (tvrdog diska);
2. familija napada razne arhitekture, ali specifično operacijski sustav Linux;
3. zloćudni program na žrtvu prenosi ili napadač ili druge, već zaražene, žrtve;
4. sve varijante nastale su iz, ili po uzoru na, isti izvorni kod,
5. varijante mogu biti *packirane*;
6. varijante mogu mijenjati svoj *payload*;
7. u programe je moguće dodavati lažne funkcije i funkcionalnosti.

U ovom radu, problem je sveden na analizu (generaliziranje) izvršnih datoteka bota koji se prenose na žrtve u svrhu klasifikacije pojedinog izvršnog koda u kojem se pokušava odrediti pripada li neki izvršni kod Mirai familiji ili ne. Tim pristupom je zaobiđeno prvo ograničenje jer se skenira mreža, a ne trajna memorija.

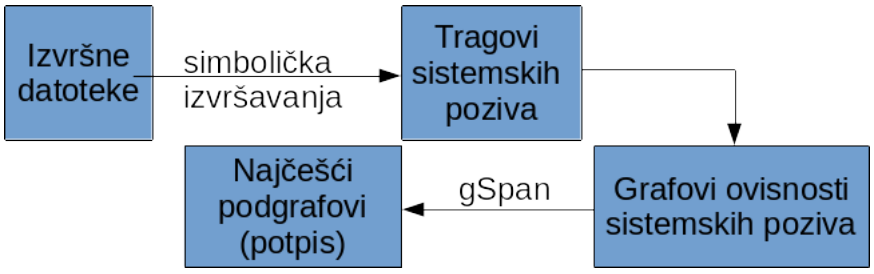


Slika 1.1: Prikaz prijenosa izvršnog koda programa s napadača na žrtvu. Crvena linija prikazuje tok podataka.

Ovakva definicija problema omogućuje implementaciju rješenja u sustav u kojem se program za klasifikaciju nalazi na sustavima za otkrivanje upada (IDS). Primjer takvog napada i sustava prikazan je na slici 1.1. IDS konstantno prisluškuje dolazeći promet, izdvaja ELF izvršne datoteke i zaključuje pripada li datoteka Mirai familiji.

1.2. Opis pristupa

Pristup korišten u ovom radu izdvaja semantičko ponašanje programa. Semantičko ponašanje opisano je grafovima ovisnosti sustavskih poziva (engl., *System Call Dependency Graph*, skr., *SCDG*), odnosno svaki program predstavljen je sustavskim pozivima koje poziva. Programi su pokrenuti u simuliranom okruženju i tehnikom simboličkom izvršavanja. Simboličko izvršavanje prolazi više mogućih puteva izvršavanja programa, na način da varijablama ne dodjeljuje konkretne vrijednosti, već pri svakom uvjetu razdvaja simulaciju na više mogućih stanja. Nad SCDG grafovima svih izvršnih datoteka u bazi, primijenjen je algoritam rudarenja grafova koji kao rezultat daje najčešće podgrafove u ulaznim grafovima – pretpostavka je da takvi grafovi najbolje opisuju ponašanje familije te takvi podgrafovi postaju svojevrstan potpis familije. Koraci izdvajanja potpisa prikazani su dijagramom 1.2. U sljedećim potpoglavljima opisan je svaki prethodno navedeni korak.



Slika 1.2: Dijagram izvlačenja potpisa iz skupa izvršnih datoteka.

1.2.1. Simboličko izvršavanje i tragovi sustavskih poziva

Simboličko izvršavanje je metoda statičke analize koda u kojoj se izvođenje programa simulira, a svaka nepoznata varijabla opisana je simbolima te se naziva simbolička varijabla. Simbolička varijabla nema konkretnu vrijednost već sadrži skup ograničenja (uvjeta) koje ispunjava. Svaki novi uvjet nad simboličkom varijablom razdvaja trenutno stanje simulacije na dva nova stanja – jedno u kojem uvjet vrijedi, a drugo u kojem uvjet ne vrijedi. Pojedino stanje simulacije sadrži informacije o svim varijablama korištenim u izvođenju. Na taj način izvršavanjem se obilaze sve moguće grane izvršavanja.

Na dijagramu 1.3 prikazano je simboličko izvršavanje programa prikazanom u programskom kodu 1.1. Na dijagramu 1.3 pravokutnicima su prikazana stanja simulacije. Gornji dio pravokutnika sadrži trenutnu naredbu na kojoj se simulacija nalazi, a donji dio sadrži skupove uvjeta korištenih simboličkih varijabli. Strelice prikazuju koja stanja nastaju iz kojeg. Početno stanje je prikazano na vrhu, a završna stanja su ona iz kojih više nema novih stanja i nalaze se pri dnu. Početno stanje ne sadrži niti jednu simboličku varijablu – varijabla a postavljena je na vrijednost 0. Nakon poziva naredbe scanf varijabla a postaje simbolička. Zbog uvjeta u programskom kodu na linijama 4-10 varijabla a može poprimiti 3 različita skupa ograničenja te se stanje razdvaja na tri sljedeća moguća stanja. Skup uvjeta simboličke varijable u svakom trenutku mora biti zadovoljiv pa ako je postavljen uvjet simboličke varijable a == 3, uvjet a < 0 na liniji 13 neće biti ispunjen.

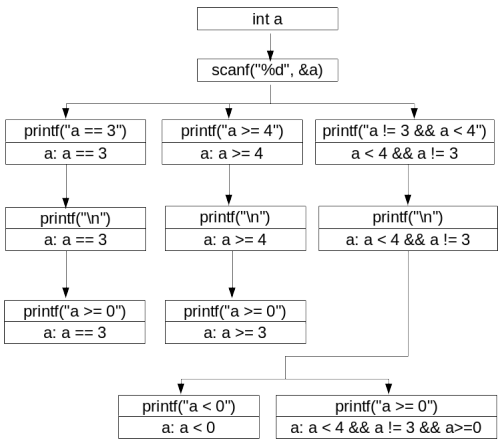
Prilikom simboličkog izvršavanja ne pozivaju se stvarni sustavski pozivi, već se oni simuliraju. Osim simuliranja sustavskih poziva, moguće je simulirati i funkcije vanjskih programskih knjižnica koje program koristi. Zbog simuliranja sustavskih poziva, preciznost simulacije (podudarnost simulacije sa stvarnim izvršavanjem) ovisi o kvaliteti implementacije simulacije.

Trag sustavskih poziva (engl., *system call trace*) čine svi sustavski pozivi pozvani u jednoj grani izvođenja programa kod simboličkog izvršavanja. sustavski pozivi su poredani slijedno po trenutku poziva te su navedene vrijednosti parametara i vrijednost koju su vratili (ako postoji). Za program 1.1 prikazan je trag 1.2 sustavskih poziva za granu izvođenja kad varijabla a ima vrijednost 4. Svaki sustavski poziv zapisan je u obliku ime_poziva(vrijednosti_parametara) = povratna_vrijednost . Trag je dobiven programom strace [4], a unesena vrijednost za varijablu a je 4.

```
1 int main(void) {
2     int a;
```

```
3       scanf("%d", &a);
4       if(a == 3) {
5           printf("a == 3");
6       } else if (a >= 4) {
7           printf("a >= 4");
8       } else {
9           printf("a != 3 && a < 4");
10      }
11      printf("\n");
12
13      if(a < 0) {
14          printf("a < 0");
15      } else {
16          printf("a >= 0");
17      }
18      printf("\n");
19
20      return 0;
21  }
```

Programski kod 1.1: Primjer programa nad kojim je izvršena simboličko pokretanje



Slika 1.3: Stanja simboličkog pokretanja za program 1.1

```
1  execve("./main", ["./main"], 0x7ffc358ee790 /* 57 vars */) = 0
2  brk(NULL)                                = 0x55ea270a7000
3  arch_prctl(0x3001 /* ARCH_??? */, 0x7fffbff26b00) = -1 EINVAL (Invalid argument)
4  access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or directory)
5  openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
6  fstat(3, {st_mode=S_IFREG|0644, st_size=126534, ...}) = 0
7  mmap(NULL, 126534, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f6624bb4000
8  close(3)                                 = 0
9  openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
10 read(3, "177ELF21130000000030>01000360q200000"..., 832) = 832
11 pread64(3, "60004000@0000000@0000000@0000000"..., 784, 64) = 784
12 pread64(3, "4000200005000GNU0200300400030000000", 32, 848) = 32
13 pread64(3, "4000240003000GNU0\t233222%27426032031331326102042 76X>263"..., 68, 880)
       = 68
14 fstat(3, {st_mode=S_IFREG|0755, st_size=2029224, ...}) = 0
15 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
       x7f6624bb2000
16 pread64(3, "60004000@0000000@0000000@0000000"..., 784, 64) = 784
17 pread64(3, "4000200005000GNU0200300400030000000", 32, 848) = 32
18 pread64(3, "4000240003000GNU0\t233222%27426032031331326102042 76X>263"..., 68, 880)
       = 68
19 mmap(NULL, 2036952, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f66249c0000
20 mprotect(0x7f66249e5000, 1847296, PROT_NONE) = 0
21 mmap(0x7f66249e5000, 1540096, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|
       MAP_DENYWRITE, 3, 0x25000) = 0x7f66249e5000
22 mmap(0x7f6624b5d000, 303104, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0
       x19d000) = 0x7f6624b5d000
23 mmap(0x7f6624ba8000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
       MAP_DENYWRITE, 3, 0x1e7000) = 0x7f6624ba8000
24 mmap(0x7f6624bae000, 13528, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
       MAP_ANONYMOUS, -1, 0) = 0x7f6624bae000
25 close(3)                                 = 0
26 arch_prctl(ARCH_SET_FS, 0x7f6624bb3540) = 0
27 mprotect(0x7f6624ba8000, 12288, PROT_READ) = 0
28 mprotect(0x55ea26f21000, 4096, PROT_READ) = 0
29 mprotect(0x7f6624c00000, 4096, PROT_READ) = 0
30 munmap(0x7f6624bb4000, 126534)            = 0
31 fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
32 brk(NULL)                                = 0x55ea270a7000
33 brk(0x55ea270c8000)                      = 0x55ea270c8000
34 read(0, "2\n", 1024)                     = 2
35 fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x1), ...}) = 0
36 write(1, "a != 3 && a < 4\n", 16)        = 16
37 write(1, "a >= 0\n", 7)                  = 7
38 lseek(0, -1, SEEK_CUR)                   = -1 ESPIPE (Illegal seek)
39 exit_group(0)                            = ?
```

Programski kod 1.2: Trag sustavskih poziva programa 1.1 dobiven programom strace, uz unos znaka "4" na standardni ulaz

Packeri su programski alati koji služe za obfuskaciju binarnog koda izvršne datoteke na način da njihove podatke sažmu ili enkriptiraju. Služe za otežavanje statičke analize zloćudnog programa. Korištenjem simboličkog izvršavanja zaobilazi se problem otkrivanja *packiranog* zloćudnog programa jer se simboličkim izvršavanjem program u početku sam otpakira ili dekriptira.

1.2.2. Usmjereni graf ovisnosti sustavskih poziva

Trag sustavskih poziva opisuje kako program komunicira s operacijskim sustavom, no ne prikazuje direktno veze između pojedinih pozvanih sustavskih poziva. Zbog toga se definira usmjereni graf ovisnosti sustavskih poziva (*engl.*, System Call Dependency Graph, skr., *SCDG*). Čvorovi grafa predstavaljaju pozvane sustavske pozive, a lukovi (usmjereni bridovi) opisuju tok informacija. Smjer luka određen je redoslijedom izvršavanja sustavskih poziva – luk ima smjer od starijeg prema novijem, a dva čvora su spojena ako dijele istu vrijednost argumenata ili ako se povratna vrijednost starijeg poziva nalazi kao argument u nekom novijem.

Prikaz stvaranja SCDG grafa prikazan je za program 1.3. Program pročita najviše 512 znakova iz datoteke dat1 i zapiše ih u datoteku dat2. Za dani primjer generiran je trag sustavskih poziva 1.4, ali su u tragu navedeni samo sustavski pozivi bitnu za samu srž aplikacije – kopiranje znakova iz jedne datoteke u drugu. Zbog ograničenja alata za crtanje grafova slika ne sadržava višestruke bridove ni čvorove s istim oznakama. Ako su dva čvora povezana s više bridova, tada je prikazan samo jedan brid, a oznake stvarnih bridova su spojene u jednu koristeći zarez – primjer je brid s oznakom 1−>3,3−>3 između čvorova read_3 i read_4. Oznakama čvorova nadodan je i broj koji označava poredak pozivanja sustavskog poziva, a primjeri su čvorovi read_3 i read_4.

```c
int main(void) {
    const char *pathname = "dat1";
    const char *pathname2 = "dat2";

    char buf[512];

    FILE * f = fopen(pathname, "r");
    FILE * f2 = fopen(pathname2, "w");
    int n = fread(buf, 1, 512, f);
    fwrite(buf, 1, n, f2);
    fclose(f);
    fclose(f2);
}
```

Programski kod 1.3: Tekst programa za programski jezik C koji prvih 512 okteta jedne datoteke upisuje u drugu

```
openat(AT_FDCWD, "file.txt", O_RDONLY)  = 3
openat(AT_FDCWD, "file2.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 4
fstat(3, {st_mode=S_IFREG|0664, st_size=16, ...}) = 0
read(3, "Ovo je primjer.\n", 4096)      = 16
read(3, "", 4096)                       = 0
fstat(4, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
close(3)                                = 0
write(4, "Ovo je primjer.\n", 16)       = 16
close(4)                                = 0
```

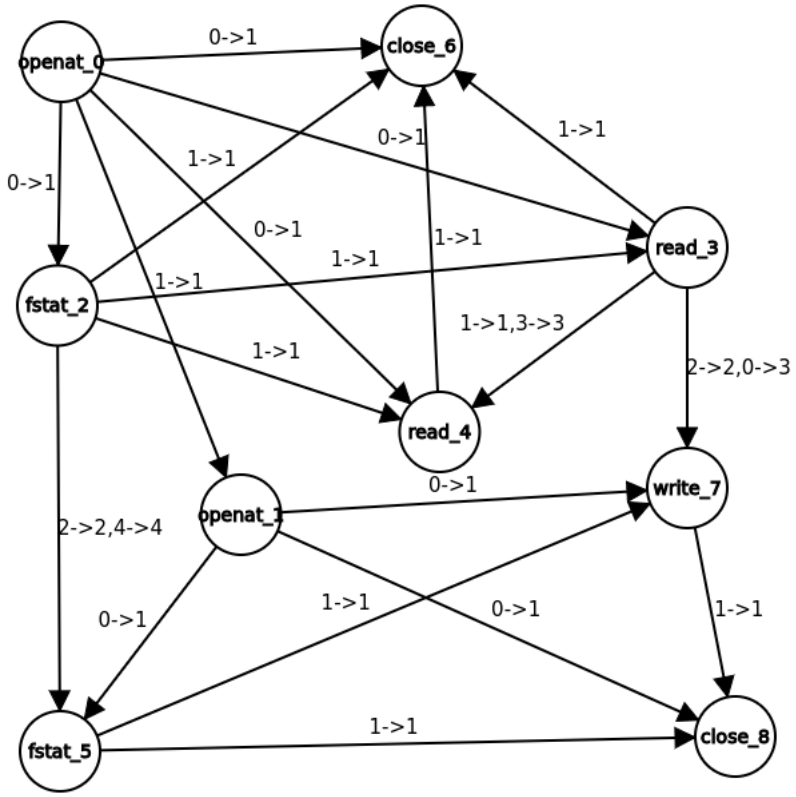
Programski kod 1.4: Dio traga sustavskih poziva dobiven programom strace

Prednosti SCDG grafova

Izmjenu originalnog zloćudnog programa moguće je ostvariti ubacivanjem lažnih ili redundantnih funkcija u programski kod te izmjenom redoslijeda nepovezanih funkcionalnosti programa [6]. Dodavanjem redundantnih funkcija u program, SCDG graf dobiva nove čvorove i bridove, ali se ne gube čvorovi i bridovi izvornog grafa. Izmjena redoslijeda pojedinih sustavskih poziva može izmijeniti SCDG graf, ali samo na način da artefaktni bridovi (iz poglavlja 2.2.3) promijene smjer, tako da su bridovi koji zapravo opisuju ponašanje netaknuti.

Nedostaci SCDG grafova

Prilikom izgradnje SCDG grafa korištenjem prethodno definiranih pravila, vidljivi su poneki artefakti. Primjer je veza između čvorova openat_0 i openat_1 koja nastaje zbog



Slika 1.4: Graf ovisnosti sustavskih poziva za trag 1.3

toga što dijele isti parametar prvi parametar. Prvi parametar je zapravo konstanta AT_FDCWD koja označava da je predana putanja relativna i često ju koristi svaki openat sustavski poziv. Bridovi koji nastaju zbog takvih konstanti nemaju semantičku vrijednost promatrajući izvođenje programa, ali unose šum koji usporava rudarenje takvih grafova (zbog povećanog broja bridova).

1.2.3. Izvlačenje znanja i klasifikacija

Jedan od načina izvlačenja znanja iz skupa grafova je rudarenje učestalih podgrafova (engl., *Frequent Subgraph Mining*, skr., *FSM*). U kontekstu SCDG grafova, gdje pojedini graf predstavlja ponašanje programa, učestali podgrafovi predstavljaju učestala ponašanja skupine programa. Algoritmi za FSM razlikuju se po tome daju li potpuno rješenje (poput algoritama gSpan i AGM) ili aproksimaciju (algoritam SLEUTH). Algoritmi kao hiperparametre primaju broj učestalih podgrafova koji trebaju vratiti te broj koji govori u koliko se grafova mora nalaziti podgraf da bi se smatrao učestalim. Rezultat FSMa nad SCDG grafovima je svojevrstan potpis koji se kasnije koristi u klasifikaciji.

U ovom radu je korišten gSpan, odnosno njegova implementacija zvana quickSpan. Kod izvlačenja potpisa algoritmu *gSpan* predan je hiperparametar *support* te su rezultantni podgrafovi poredani po veličini te se kao krajnji rezultat smatra *n* najvećih. Pseudokod je prikazan u 1. Kod klasifikacije, provjerava se sadrži li graf podgrafove iz potpisa. Prolazeći kroz svaki podgraf iz potpisa, algoritmom *gSpan* izdvoje se svi podgrafovi koje sadrži i podgraf iz potpisa i graf kojeg klasificiramo. Nakon toga ispituje se sličnost takvih podgrafova i podgrafa potpisa i ako su slični, tada je graf klasificiran kao da pripada traženom razredu. Mjera sličnosti grafova definirana je kao omjer broja bridova grafova. Jednadžba je prikazana slikom 1.5.

Algorithm 1: Izvlačenje potpisa iz skupa grafova koristeći gSpan algoritam

Input: G: skup SCDG grafova
Input: support: hiperparametar gSpan algoritma
Output: sign: skup podgrafova koji čine potpis
sign $\leftarrow \{\emptyset\}$;
CSG $\leftarrow$ `gSpan(`G, support`)` ;
maxCSG $\leftarrow$ `sortiraj(`CSG, n`)` ; `// Dohvati n najvećih učestalih podgrafova`
foreach g $\in$ maxCSG **do** sign $\leftarrow$ sign $\cup$ {g } ;

Algorithm 2: Klasifikacija SCDG grafa

Input: sign: skup podgrafova koji čine potpis
Input: support: hiperparametar gSpan algoritma
Input: threshold: :
prag sličnosti dva grafa
Output: Booleova vrijednost spada li graf g unutar klase ili ne
foreach g_1 $\in$ sign **do**
    `// Support hiperparametar je 1`
    subsubG $\leftarrow$ `gSpan(`{g , g_1}, 1`)` ;
    foreach g_2 $\in$ subsubG **do**
        if `slicnost_grafova(`g_2 , g_1`)` > **threshold:**
        then
            return *True*
return *False*

$$M(G_1, G_2) = |E(G_1)|/|E(G_2)|$$

Slika 1.5: Mjera sličnosti grafova koja uzima u obzir omjer broja bridova

1.3. Programsko ostvarenje

U ovom potpoglavlju opisani su implementacijski detalji, korišteni alati i načini pokretanja napravljenog programa.

1.3.1. Radni okvir Angr

Okvir angr je alat za analizu izvršnih datoteka, a napisan je za programski jezik Python. Pruža mogućnost simboličkog izvršavanja programa, ali i automatizirano otkrivanje sigurnosnih ranjivosti te nudi alate za analizu ponašanja programa, poput grafa ovisnosti podataka (engl., *Data Dependency Graph*) i grafa kontrole toka (engl., *Control-Flow Graph*). Sposoban je analizirati programe za razne procesorske arhitekture (ARM, x86, x64) te razne formate izvršnih datoteka (ELF i PE/EXE).

Prije početka simulacije, potrebno je učitati izvršnu datoteku i pružiti Angru sve potrebne vanjske knjižnice, a procesorsku arhitekturu, format izvršne datoteke i procesorsku arhitekturu alat odredi sam. Simulacija se provodi koračanjem kroz blokove izvršnog programa, odnosno stanja simulacije. Blok je dio učitanog izvršnog programa koji sadrži ili predefinirani najveći broj instrukcija ili završava uvjetovanom instrukcijom ili instrukcijom skoka. Stanje simulacije sadrži blok koda te sadržaj registara i memorije, a njezinim izvršavanjem dobivaju se sljedeća stanja, odnosno sljedbenici. Simulacija započinje sadržavajući samo jedno početno stanje kojoj blok počinje s instrukcijama na početku programa. Angr sprema nekoliko različitih listi stanja, poput listi aktivnih stanja, mrtvih stanja i uspješnih stanja. Aktivna stanja su ona koja će se izvesti u sljedećem koraku simulacije, a nakon koraka simulacije sljedbenici tih stanja postaju aktivna stanja. Mrtva stanja su ona čijim se izvođenjem dogodila greška, a kod uspješnih stanja je izvođenje simulacije uspješno završeno. Dio programskog koda 1.5 prikazuje korištenje navedenih mehanizama i pokretanje simulacije u Angru.

U ovom radu korišteni su Angrovi mehanizmi prijelomnih točaka (engl., *breakpoint*) i dodataka (engl., *plugin*). Korištenjem prijelomne točke moguće je izvršiti korisnički definiranu funkciju u trenutku kad se dogodi neki događaj, poput pozivanja sustavskog poziva, zapisivanja i čitanja iz memorije, ili poziva do tad neviđene ins-

trukcije. U dodacima se spremaju dodatne informacije vezane uz stanje, a korisnik određuje što se događa u trenutku kad pojedino stanje nastaje. U ovom radu navedeni mehanizmi iskorišteni su za pamćenje tragova sustavskih poziva.

```
1   so_dirs = [x[0] for x in os.walk('./arm_libs/armel')]
2
3   proj = angr.Project(os.path.join(BIN_DIR, FILE_NAME),
4                       load_options={
5                           'auto_load_libs': True, 'except_missing_libs': True, "
                                use_system_libs": False,
6                           'ld_path': so_dirs
7                       },
8                       use_sim_procedures=False)
9
10  start_state = proj.factory.entry_state()
11
12  root_nodes = list()
13  start_state.register_plugin('syscall_tree', SyscallTreePlugin())
14  start_state.inspect.b('syscall', when=angr.BP_AFTER, action=
        syscall_tree_action_builder(root_nodes))
15
16  simgr = proj.factory.simgr(start_state)
17
18  print("Exploring binary...")
19  simgr.explore(n=400)
20  print("Exploring binary finished.")
```

Programski kod 1.5: Dio koda zaslužan za simboličku analizu u angru i učitavanje potrebnih dodataka i vanjskih knjižnica

1.3.2. Ekstrakcija SCDG grafova koristeći okvir Angr

Dodavanje prevedenih sustavskih knjižnica

U ovom radu ciljana arhitektura je ARM, odnosno njegove inačice ARMLE i ARMHF, a ciljani operacijski sustav je Linux. Svaki dinamički preveden koristi standardne knjižnice (poput *stdlib.c*) za ispravan rad pa tako i Angru treba putanja standardnih knjižnica prevedenih za ciljani operacijski sustavi ciljanu arhitekturu. ARMHF preuzet je iz operacijskog sustava RaspbianOS za arhitekturu ARM [2]. ARMEL je preuzet iz operacijskog sustava Debian kompajliranog za ARMEL arhitekturu [7].

Implementacija nedostajućih sustavskih poziva

Pokušajem pokretanja analize nad skupom podataka utvrđeno je da postoje sustavski pozivi koji nisu implementirani. Primjeri takvih sustavskih poziva su access, connect,

getsockname, ioctl, readlink, rt_sigprocmask, set_robust_list i set_thread_area. U nastavku je prikazana implementacija sustavskog poziva readlink.

sustavski poziv readlink učitava vrijednost simboličkog linka [1]. Funkcija kao parametar prima simbolički link kao niz znakova, spremnik u koji zapisuje rezultat kao niz znakova i veličinu tog spremnika. Kao rezultat vraća broj zapisanih znakova u spremniku. Deklaracija funkcije prikazana je kodom 1.6. Ako je kao simbolički link predana vrijednost /proc/ self /exe, onda se u spremnik zapisuje puna putanja do izvršne datoteke pokrenutog programa. Programi u promatranom skupu podataka pozivaju funkciju isključivo s tom vrijednošću simboličkog linka te zbog toga implementacija funkcije nije potpuna, već sadrži samo korišten slučaj. Okolina u kojoj Angr simbolički izvršava program je simulirana, no u ovoj implementaciji vraćena je puna putanja do simbolički izvršavane izvršne datoteke. Pri definiranju novog sustavskog poziva u Angru potrebno je stvoriti razred koji nasljeđuje SimProcedure i implementirati metodu run koja se poziva pri pozivanju sustavskog poziva. Klasa je spremljena u datoteku procedures/ linux_kernel / readlink .py. Nakon toga sustavski poziv je unešen u sustav, no unos nije opisan jer je postupak predugačak i, trenutno, nije službeno dokumentiran.

```
1    ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

Programski kod 1.6: Deklaracija readlink funkcije

```
1  class readlink(SimProcedure):
2      def run(self, pathname, buf, bufsiz):
3          elf_path = (os.getcwd() + "/" + self.state.project.filename).encode('utf-8
               ')
4          buffsize_int = self.state.solver.eval(bufsiz)
5          elf_path_bvv = claripy.BVV(value=elf_path, size=len(elf_path) * 8)
6          store_size = min(buffsize_int, len(elf_path))
7          self.state.memory.store(addr=buf, data=elf_path_bvv, size=store_size)
8          l.debug('readlink: {} : {} : {} | {}'.format(pathname, buf, buffsize_int,
               buffsize_int))
9          return len(elf_path)
```

Programski kod 1.7: Implementacija readlink sustavskog poziva za angr

Implementacija nedostajućih instrukcija

Interno, Angr simboličku analizu izvodi nad međukodom kojeg dobije alatom Vex, kojeg koristi i popularan alat za analizu programa Valgrind. Prilikom pokušaja sim-

boličke analize nad nekim izvršnim datotekama, uočeno je da trenutna verzija Vexa koju koristi Angr nema implementiranu instrukciju rdsspq. Instrukcija je dodana u okviru Intelovog sklopovskog proširenja za osiguravanje kontrole toka (engl., *Control-flow Enforcement Technology*, skr. CET), a onemogućuje promjenu adrese povratka iz funkcije koristeći sklopovsku podršku. Promjena adrese je onemogućena jer se povratne adrese čuvaju na rezervnom, tzv., stogu u sjeni (engl., *Shadow stack*) kojem nije moguće pristupiti. Prilikom povratka iz funkcije učitava se povratna adresa te ako učitana povratna adresa ne odgovara onoj sa stoga u sjeni, događa se iznimka koja signalizira grešku.

U ovom radu sve naredbe za manipulaciju stoga u sjeni su implementirane kao prazna instrukcija (eng., *No operation Code*, skr., *NOP*). Promjene su napravljene u izvornom kodu alata Vex.

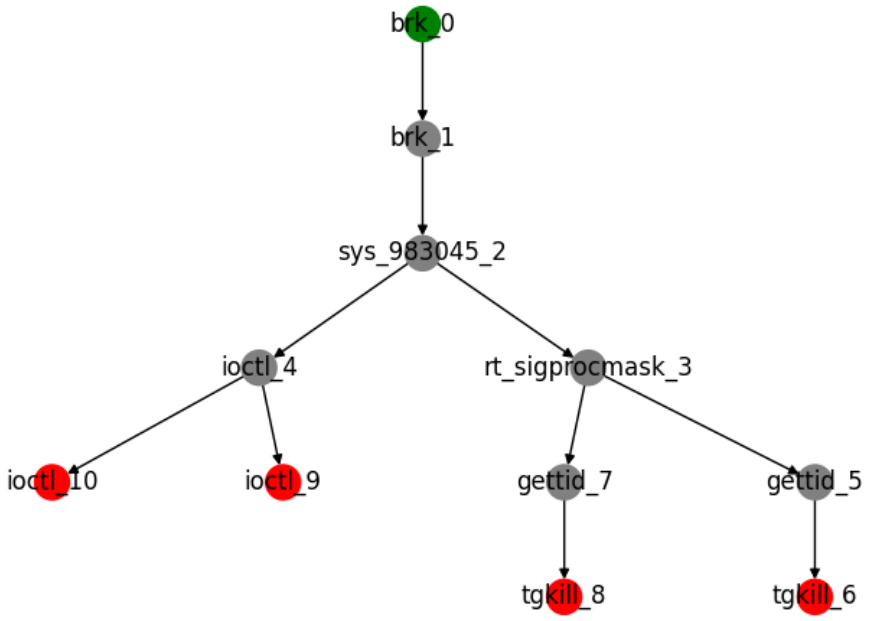
Stvaranje tragova sustavskih poziva

Stvaranje tragova sustavskih poziva nije nativno podržano u Angru, već je izvedeno pomoću proširenja stanja i prijelomnih točaka. Pomoću prijelomnih točaka moguće je izvršiti neku funkciju u trenutku kada se dogodi događaj u simulaciji izvođenja programa – poput ulaska i izlaska iz sustavskog poziva, čitanja ili zapisivanja u određeni segment memorije, čitanja ili zapisivanja u određeni registar ili poziv procesorske naredbe. Pomoću proširenja stanja Angr omogućuje zapisivanje dodatnih informacija u pojedino stanje. Informacije se prenose (odnosno kopiraju) s izvornog stanja na stanja koja iza njega slijede, te ih je moguće dinamički mijenjati.

Kod stvaranja tragova sustavskih poziva u svako stanje simulacije upisan je zadnji, do tog trenutka pozvani, sustavski poziv i svi njegovi sljedbenici, a prekidne točke postavljene su na poziv sustavskog poziva te u zadano stanje upisuju informacije o pozvanom sustavskom pozivu te se nadodaje u listu sljedbenika svog prethodnika. Na taj način je niz tragova sustavskih poziva zapisan kao skup stabala. Primjer jednog takvog stabla prikazan je slikom 1.6. Zbog ograničenja programa za crtanje grafa, uz ime sustavskog poziva, oznaci čvora nadodan je i redni broj otkrivanja sustavskog poziva prilikom simulacije.

Stvaranje SCDG grafova iz tragova sustavskih poziva

Tragovi sustavskih poziva mogu se zamisliti kao usmjerena stabla u kojima su čvorovi pozvani sustavski pozivi, a bridovi opisuju kronologiju pozivanja sustavskih poziva – brid je usmjeren od prethodnika prema sljedbeniku. U ovom radu, SCDG graf dobi-



Slika 1.6: Prikaz grafa sustavskih tragova

ven je obilaženjem tragova sustavskih poziva u dubinu (engl., *Depth-first Search*, skr., *DFS*) te pamćenjem vrijednosti koje sustavski pozivi primaju ili vraćaju do trenutno promatranog čvora. Programski kod 1.8 prikazuje funkciju koja prima korijen stabla i, do tog trenutka, poznate čvorove čiji sustavski pozivi primaju ili vraćaju određene vrijednosti. Parametar funkcije parametri je rječnik koji mapira vrijednost u listu čvorova sustavskih poziva koji ga primaju, dok je parametar povratne_vrijednosti također rječnik koji mapira vrijednost u listu čvorova sustavskih poziva koji ju vraćaju. Prvi dio funkcije (linije 3 - 15) upisuju trenutni čvor (zapisan u varijabli cvor) u prethodno navedene rječnike u sve liste povezane s vrijednostima njegovih argumenata koje je primio i vratio. Drugi dio funkcije (linije 17 i 18) izvode korak rekurzije pozivajući funkciju nad svim nasljednicima čvora. Treći dio funkcije (linije 20 do 24) brišu informacije nadodane u drugom koraku. Zadnji dio funkcije (linije 26 do 33) zapisuje bridove grafa na način da trenutni čvor povezuje s prethodnicima kod kojih povratna vrijednost ili vrijednost argumenata njihovih sustavskih poziva postoji u argumentima sustavskog poziva trenutnog čvora. Smjer brida je od prethodnika prema sljedbeniku, a oznaka brida je uređeni par dva prirodna broja koji odgovaraju poziciji na kojoj se određena vrijednost nalazi u argumentima prethodnog i sljedećeg poziva, uz iznimku da je indeks povratne vrijednosti prethodnika označena s nulom, odnosno pozicije argumenata broje se od jedinice.

```
fun stvori_scdg_iz_cvora(cvor, parametri, povratne_vrijednosti):

    # Nadodaj sebe u rjecnike "parametri" i "povratna_vrijednost"
    #  da se nasljednici mogu povezati s tobom
    for argument in cvor.sustavski_poziv.argumenti:
      if argument not in parametri:
        parametri[argument] = lista()
      parametri[argument].dodaj(cvor.id)

    povratna_vrijednost_poziva = cvor.sustavski_poziv.povratna_vrijednost

    if povratna_vrijednost_poziva:
     if povratna_vrijednost_poziva not in povratne_vrijednosti:
       povratne_vrijednosti[povratna_vrijednost_poziva].append(cvor.id)

    # Korak rekurzije
    for nasljednik in cvor.nasljednici:
      stvori_scdg_iz_cvora(nasljednik)

    # Izbrisi se iz rjecnika "parametri" i "povratna_vrijednost"
    for argument in cvor.sustavski_poziv.argumenti:
      parametri[argument].pop()

    povratne_vrijednosti[povratna_vrijednost_poziva].pop()

    # Provjeri s kim se trebas povezati od prethodnika
    for argument in cvor.sustavski_poziv.argumenti:
      if argument in parametri:
         for prethodnik in parametri[argument]:
           dodaj_brid(prethodnik, cvor)
      if argument in povratne_vrijednosti:
      for prethodnik in povratne_vrijednosti[argument]:
           dodaj_brid(prethodnik, cvor)
```

Programski kod 1.8: Rekurzivna funkcija za stvaranje SCDG grafa iz čvora sustavskog traga

1.3.3. Treniranje i klasifikacija algoritmom quickSpan

U ovom radu korištena je quickSpan implementacija gSpan algoritma. Implementacija je otvorenog koda te, za razliku od originalne implementacije, omogućuje višedretveno izvođenje te je memorijski manje zahtjevna i brža [3]. Programsko rješenje ima niz opcija pokretanja programa, no u ovom radu korišteni su sljedeći parametri:

- support – *support* hiperparametar gSpan algoritma,
- input_file – putanja do tekstualne datoteke koja sadrži ulazne grafove,

– output_file – putanja do tekstualne datoteke u koju će program upisati rezultat,

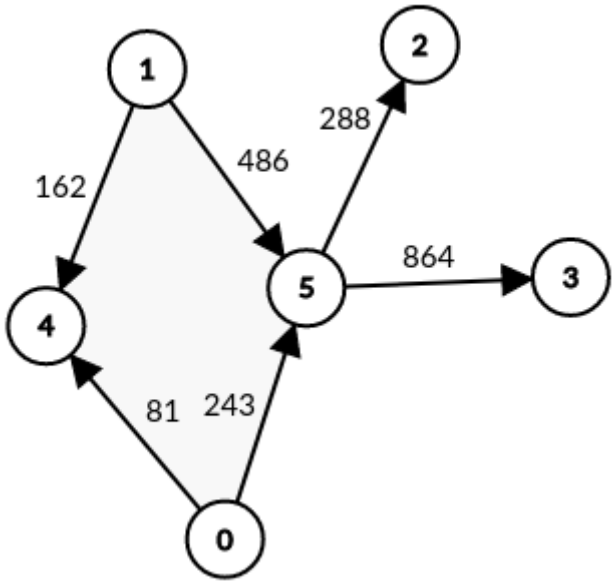
– threads – broj dretvi koje program treba koristiti,

– directed_graph – zastavica koja određuje je li graf usmjeren,

– pattern – zastavica koja određuje treba li zapisati rezultat,

– biggest_subgraphs – broj koji određuje koliko će prvih najvećih podgrafova biti zapisano kao rezultat.

Datoteka koja sadrži ulazne grafove mora biti kodirana u UTF-8 formatu, a graf zapisan na način da se u prvoj liniji navede jedinstveni broj grafa, a nakon njega definiraju svi njegovi čvorovi i bridovi, zajedno s njihovim oznakama. Početak grafa definira se linijom oblika `t # id_grafa`, čvor linijom `v id_cvora oznaka_cvora`, a brid `e id_izlaznog_cvora id_ulaznog_cvora oznaka_brida`. Oznake vrhova i bridova u quickSpan implementaciji moraju biti pozitivni cijeli brojevi, ali su bridovi u SCDG grafu označeni s uređenim parom. Zbog toga su bridovi preimenovani na način da je uređeni par `(x, y)` postaje `2^x 2^y`. Primjer ulazne datoteke 1.9 sadrži jedan graf prikazan na slici 1.7. Datoteka s rješenjem također je kodirana u UTF-8 formatu te koristi isti oblik za početak grafa, čvorove i bridove, ali je nadodana i informacija o tome koliko i koji grafovi sadrže promatrani podgraf.



Slika 1.7: Prikaz grafa za datoteku 1.9

```
t # 0
v 0 0
v 1 4
v 2 1
v 3 5
v 4 2
v 5 3
e 0 1 81
e 2 1 162
e 2 3 486
e 0 3 243
e 3 4 288
e 3 5 864
```

Programski kod 1.9: Primjer ulazne datoteke koja predstavlja graf

Program za pokretanje modela pokreće se naredbom 1.10, a kao argumente pri pokretanju prima sljedeće:

- malwares - putanja do direktorija u kojem se nalaze zlođudni programi,
- result - putanja do datoteke u kojoj se zapisuju rezultati

```
$ ./train.py --malwares ./data --result ./result
```

Programski kod 1.10: Pokretanje treniranja

Program za klasifikaciju izvršne datoteke pokreće se naredbom 1.11, na kraju izvršavanja korisniku ispisuje pripada li izvršna datoteka familiji, a kao argumente pri pokretanju prima:

- binary - putanja do datoteke koju se ispituje,
- signature - putanja do datoteke u kojoj se nalazi rezultat treniranja, odnosno potpis familije.

```
$ ./classify.py --binary ./example --signature ./result
```

Programski kod 1.11: Pokretanje klasifikacije

1.4. Rezultati

Istraživanje najboljih hiperparametara je preskočeno zbog prevelike složenosti izvođenja algoritma. Hiperparametar support postavljen je na 0.5, a threshold sličnosti na 0.46, po najbolje dobivenim rezultatima iz originalnog rada [6]. Skup podataka sadržava 407, po njihovom MD5 sažetku različita, zloćudna programa iz familije Mirai, prikupljenih u lipnju 2017. godine. Programi su namijenjeni za arhitekture ARM EABI 4 (196 primjerka), PowerPC ili cisco 4500 (3 primjerka), Intel 8036 (2 primjerka), MIPS (1 primjerak), x86 (1 primjerak), te nepoznate ARM arhitekture. Arhitekture izvršnih datoteka određene su linux naredbom file . Izvlačenje SCDG grafa Angrom uspješno je za samo 131 zloćudnu izvršnu datoteku. Po uzoru na izvorni rad, skup dobroćudnih datoteka sastoji se od 131 nasumično odabrane izvršne datoteke iz /usr/bin direktorija za operacijski sustav Ubuntu 20.04 i x86_64 arhitekturu. Za sve odabrane izvršne datoteke uspješno je izvučen SCDG graf.

Pri generiranju potpisa korišteno je 105 (80%) nasumično odabranih zloćudnih programa. Pri evaluaciji korišteno je 105 nasumično odabranih zloćudnih programa i 105 nasumično odabranih dobroćudnih programa. U eksperimentu dobivena su 3 lažno pozitivna rezultata te 1 lažno negativan. Preciznost je 97.20%, odziv je 99.05%. F0.5 (koja naglasak stavlja na preciznost, a ne na odziv) mjera je 0.9756.

1.5. Manjkavosti i moguća poboljšanja

Smatram da su brojne manjkavosti i imam više ideja kako poboljšati ovu metodu. Slobodno se javite na mail, linkedin ili bilo koji 3. servis na kojem me pronađete.

Hrvoje Spajić

2. Literatura

[1] Linux Foundation. readlink(2) linux user's manual, 2021. URL https://man7.org/linux/man-pages/man2/readlink.2.html.

[2] Raspberry PI Foundation. Operating systems - raspberry pi, 2021. URL https://www.raspberrypi.org/software/operating-systems/.

[3] INFRIA. quickspan, 2017. URL https://gitlab.inria.fr/Quickspan/quickspan.

[4] Paul Kranenburg i Dmitry Levin. strace, 2021. URL https://github.com/strace/strace.

[5] Zhen Ling, Yiling Xu, Yier Jin, Cliff Zou, i Xinwen Fu. *New Variants of Mirai and Analysis*, stranice 1–8. Springer International Publishing, Cham, 2020. ISBN 978-3-319-32903-1. doi: 10.1007/978-3-319-32903-1_174-1. URL https://doi.org/10.1007/978-3-319-32903-1_174-1.

[6] Vesselin Bontchev Najah Ben Said, Fabrizio Biondi. Detection of mirai by syntactic and semantic analysis, 2017.

[7] The Debian Project. Debian – arm ports, 2021. URL https://www.debian.org/ports/arm/.

[8] TrendMicro. Sora and unstable: 2 mirai variants target video surveillance storage systems. techreport, 2020. URL https://www.trendmicro.com/vinfo/us/security/news/internet-of-things/sora-and-unstable-2-mirai-variants-target-video-surveillance-stora