SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 3053

Automatizirano testiranje uz pomoć alata AFL

Ivanković, Ivan

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

DIPLOMSKI ZADATAK br. 3053

Pristupnik:	**Ivan Ivanković (0036510560)**
Studij:	Računarstvo
Profil:	Programsko inženjerstvo i informacijski sustavi
Mentor:	doc. dr. sc. Stjepan Groš

Zadatak: **Automatizirano testiranje uz pomoć alata AFL**

Opis zadatka:

Ranjivosti u programskoj podršci su značajan vektor napada preko kojega se kompromitiraju sustavi. Iz tog razloga vrlo je bitno otkriti ranjivosti i ukloniti ih. Međutim, to je vrlo težak zadatak budući da u iole složenijoj programskoj podršci nije moguće provjeriti sve moguće puteve kroz kod, što ovisi o ulaznim podacima, ali i o njihovom sadržaju. Često korištena metoda za otkrivanje ranjivosti je neizrazito testiranje (engl. fuzzing). U osnovi, radi se o nasumičnom generiranju podataka koji se potom predaju sustavu koji se testira te se promatra njegov odziv na temelju čega se zaključuje je li došlo do pogreške. Dodatno se takva testiranja za složenije sustave odvijaju u virtualiziranim okolinama, kakav je primjerice QEMU. U sklopu ovog diplomskog rada potrebno je istražiti mogućnosti alata American fuzzy lop (AFL) koji se koristi za neizrazito testiranje i njegovu integraciju s alatom QEMU. Ispitati sustav traženja pogrešaka na nekim primjerima te složiti metodologiju testiranja cijelog operacijskog sustava koji se izvršava unutar QEMU.

Rok za predaju rada: 27. lipnja 2022.

SADRŽAJ

1. Uvod

Kontinuirani razvoj računalnih sustava i programskih rješenja je s godinama rezultirao povećanjem kompleksnosti. Kompleksnost direktno utječe na sigurnost zbog povećavanja površine napada, a razvojnim inženjerima je sve teže ostati informiranima o sigurnosnim rizicima velikih sustava. Trend rasta korištenja gotovih rješenja bila ona otvorenog ili zatvorenog koda doprinosi smanjenju sigurnosti sustava, jer se gotova rješenja nerijetko uzimaju olako bez da su inženjeri upućeni u sigurnosne rizike koje ta rješenja donose.

Sigurnost je izrazito bitna za sustave koje je teško testirati i analizirati kao na primjer ugradbena računala i sustave zatvorenog koda. Ugradbena računala je teško adekvatno testirati radi manjka podrške na samim ugradbenim računalima. Problem je također cijena ugradbenih računala i riskiranje kvarova ako se sustav optereti zahtjevnim testovima. Sustavi zatvorenog koda su zahtjevni za testiranje, jer se kod ne može pregledati ručno, a reverzno inženjerstvo je mukotrpan proces pogotovo za sustave koji su izrazito kompleksni.

Kako bi se riješio dio problema testiranja ugradbenih računala može se koristiti emulacija u te svrhe. *Quick Emulator* (QEMU) [1] je emulator koji podržava brojne arhitekture, ali postoji i mnoštvo inačica koje proširuju funkcionalnosti i dodaju druge arhitekture koje nisu podržane od alata QEMU. Otkrivanje ranjivosti se može provesti pomoću neizrazitog testiranja (engl. *fuzzing*). Ova metoda testiranja funkcionira na jednostavnim principima, no kako bi se brzo i sa što manje pogrešaka otkrile greške potrebno je primijeniti različite tehnike optimizacije testiranja. Jedan od popularnijih alata za tu svrhu je *American fuzzy lop* (AFL) [2] i zadnjih nekoliko godina njegova inačica AFL++ [3] koja više prati smjer razvoja novih tehnologija.

Cilj ovog rada je istražiti i implementirati sustav za testiranje kojim se mogu testirati različiti emulirani sustavi. Sustavi će se izvršavati unutar QEMU alata, a AFL++ alat se koristi u svrhe testiranja i otkrivanja grešaka unutar emuliranog sustava. Iako se sustavi značajno mogu razlikovati, potrebno je napraviti arhitekturu za testiranje koju je moguće mijenjati ovisno o cilju i računalnoj arhitekturi koja se testira.

Rad je strukturiran tako da drugo poglavlje služi kao uvod u QEMU alat i objašnjava njegove specifičnosti, mogućnosti i bitne dijelove za implementaciju praćenja. Treće poglavlje objašnjava vrste neizrazitog testiranja, AFL alat, njegovu inačicu AFL++ i biblioteku

LibAFL. Četvrto poglavlje objašnjava promjene koje su trebale biti napravljene na QEMU alatu i programski kod potreban za povezivanje QEMU alata s AFL++ alatom. Peto poglavlje postavlja testni slučaj i provodi se eksperiment. Na kraju rada se nalaze zaključak i literatura.

2. QEMU

Quick Emulator (QEMU) je emulator otvorenog koda [4]. Zato što je alat otvorenog koda moguće je raditi promjene u izvornom kodu kako bi se ostvarile funkcionalnosti koje nisu implementirane originalnom verzijom alata. U sklopu ovog diplomskog rada nadograđuju se dijelovi izvornog koda koji su zaduženi za prevođenje osnovnih blokova koda, te se zato obrađuju osnove funkcioniranja alata QEMU i njegovog dinamičkog prevodioca.

2.1. Vrste simulacije

QEMU ima dva osnovna načina emulacije, korisničku (engl. *user*) i sistemsku (engl. *system*). U sklopu ovog rada je bitnija sistemska emulacija, no bitno je naznačiti razlike između obje emulacije zato što se obje pojavljuju više puta u narednim poglavljima.

2.1.1. Korisnička emulacija

QEMU implementira jednostavniju vrstu emulacije koja se naziva korisničkom emulacijom [5]. Ova vrsta emulacije omogućuje korisniku da pokreće procese namijenjene određenoj arhitekturi na bilo kojoj drugoj arhitekturi koju QEMU sustav podržava. Korisnička emulacija je brža od sistemske emulacije zato što koristi dio implementacije sistemske emulacije i ne treba emulirati sve dijelove.

Podržana je emulacija programa namijenjenim Linux i BSD operacijskim sustavima. Osigurava prevođenje sistemskih poziva neovisno o razlikama poslužiteljskog računala i emulacije te podržava prevođenje signala ulazno/izlaznih uređaja. Oslanja se na poslužiteljsko računalo koje je zaduženo da reagira na signale koje šalje emulirano okruženje. Također, dretve pokrenute unutar emuliranog okruženja se pokreću u jednakom broju na poslužiteljskom računalu.

Ova vrsta emulacije je efektivna kada se trebaju pokretati jednostavni alati koji su prevedeni za drugu arhitekturu. Radi jednostavnosti ove emulacije, alati AFL i AFL++ podržavaju praćenje izvršavanja zato što se emulira samo program, a ne cijela okolina sustava.

2.1.2. Sistemska emulacija

Glavni fokus ovog rada je na testiranju sistemskih emulacija unutar QEMU alata. Sistemska emulacija se razlikuje od korisničke emulacije po tome što emulira cijeli sustav zajedno s potrebnim brojem procesora i svim okolnim uređajima tog sustava [6].
Postoje brojne funkcionalnosti alata, ali za istraživanje o stanjima i upravljanje njima navedene su najznačajnije funkcionalnosti sistemske emulacije: [7]:

Monitor QEMU sistemska emulacija nudi opciju pokretanja QEMU monitora koji je izrazito interesantan zato što omogućuje korisniku različite napredne opcije upravljanja sustavom. Korisnik može upravljati uređajima koje koristi emulacija, upravljanje preslikama stanja emulacije te uvid u unutarnje stanje emulacije bez da je potrebno implementirati kompleksne funkcionalnosti. Moguće je implementirati u izvornom kodu dodatne funkcionalnosti ako su korisniku potrebne.

GDB udaljeni pristup QEMU sistemska emulacija omogućava korištenje *GNU Debugger alata* (GDB) u svrhe istraživanja unutarnjih stanja tijekom izvođenja sistemske emulacije. Kako bi se omogućio pristup alatu GDB potrebno je tijekom pozivanja emulacije dodati "-s" argument kako bi se otvorila vrata na lokaciji 1234 za pristup alatu GDB. Dodatno, opcija "-S" omogućava zaustavljanje okruženja sve dok ne dobije naredbu od korisnika preko GDB alata da nastavi s normalnim radom. GDB alatom se može detaljno provjeravati izvođenje sistemske emulacije i svih stanja koja taj alat može poprimiti.

Virtualizacija uređaja Podržana je virtualizacija uređaja koje korisnik želi povezati s emuliranim sustavom. Podržava brojne vrste mrežnih i USB uređaja.

2.2. Tiny Code Generator

QEMU je dinamički prevoditelj [8]. Dinamički prevoditelji uzimaju kratak dio koda koji je obično osnovni blok (engl. *basic block*) koji se prevodi i sprema za kasnije korištenje. Dinamički prevodioci se oslanjaju na činjenicu da se brojni osnovni blokovi izvode puno puta te da će spremanje prethodno prevedenih operacija ubrzati tok izvođenja.

Osnovni blok je definiran kao jedinica koda maksimalne dužine koja nema grananja [9]. Osnovni blok počinje s početnom operacijom, a završava s grananjem, skokom ili nekom drugom operacijom koja mu mijenja tok. Svojstvo osnovnog bloka je da ako se jedna operacija izvrši, tada se izvršavaju ostale operacije u određenom poretku dokle god nema iznimaka tijekom izvršavanja koda.

Prvi put kada se QEMU susretne s dijelom koda koji prethodno nije preveden, prevodi instrukcije s virtualiziranog računala na instrukcije poslužiteljskog računala. Kako bi to postigao, u pozadini koristi *Tiny Code Generator* (TCG) alat [10].

TCG radi tako da prevodi instrukcije emuliranog sustava, na niz naredbi poslužiteljskog sustava koje se zatim izvršavaju na poslužiteljskom računalu. TCG je naravno sporiji od izvođenja samih operacija na izvornoj arhitekturi za koju je kod namijenjen, jer mora napraviti prevođenje ili traženje prevedenih instrukcija, ali zato su osmišljene brojne optimizacije. Značajnije optimizacije su navedene u sljedećem dijelu:

Optimizacije stanja procesora Tijekom izvođenja procesor se može nalaziti različitim unutarnjim stanjima koja u konačnici utječu na rezultat izvođenja naredbi. Kako bi emulacija bila brža, QEMU zapisuje stanja procesora i ta stanja zajedno s rezultatom prevođenja zapisuje u spremnik za blokove prevođenja (engl. *Translation Blocks*, TB). Tijekom svakog prevođenja bloka, prije prevođenja se provjerava nalazi li se blok u memoriji , ako je ranije preveden. Ako se utvrdi da je trenutačni blok jednak nekom bloku koji je ranije preveden, tada se izvršava prethodno prevedeni blok. Ako se neki dio stanja procesora promijenio, tada se stvara novi TB koji se sprema u memoriju za potencijalno daljnje korištenje. Zato što je stanje procesora zapisano u TB, promjene u stanjima procesora utječu da se blok treba ponovno prevesti. Zato što su stanja brojna ne prate se sve informacije o stanju procesora zato što su neke bitnije od drugih. Tim procesom se značajno dobiva na učinkovitosti izvršavanja zato što nema potrebe za prevođenjem svih instrukcija više puta, jer je ta operacija vremenski skupa. Također, pazi se na konzistentnost i ispravnost rezultata operacije tako što se prati stanje procesora tijekom prijevoda svake instrukcije.

MMU emulacija Sistemske emulacije unutar QEMU alata koriste programsku izvedbu jedinice za upravljanje memorijom (engl. *Memory Management Unit*, MMU). QEMU koristi spremnik prijevoda adresa (eng. Translation Lookaside Buffer, TLB) kako bi

ubrzao prevođenje te MMU optimizacija svaki put kada se pristupa memoriji radi prevođenje virtualne adrese u fizičku adresu. Taj način upravljanja memorijom sprječava probleme koji mogu nastati kada se mijenja raspored memorije, čime bi (ukoliko bi se čuvale virtualne adrese) spremnik prevedenih blokova postao nevaljan. Zbog MMU komponente, prevedeni blokovi u sebi čuvaju fizičke adrese i ostaju valjani, neovisno o memorijskim događajima.

Spajanje povezanih blokova Dodatne performanse se ostvaruju povezivanjem više prevedenih blokova u niz. Kako bi TCG mogao odrediti koji osnovni blok je sljedeći potrebne su mu informacije simuliranog programskog brojača (engl. *Program Counter*, PC) i ostale informacije o stanju procesora.

Kada se ne koristi ova optimizacija TCG prevodioca izlaskom iz TB prolazi kroz epilog tog TB kako bi se mogao vratiti u glavnu petlju. Povratkom u glavnu petlju QEMU uzima sljedeći TB i izvodi ga ili prevodi, ovisno o tome postoji li TB u memoriji. Zatim QEMU mora izvršiti prolog i instrukcije sljedećeg TB.

Epilog je izraz koji se koristi kao skup operacija koje se trebaju izvesti kako bi se procesor vratio u početno stanje prije ulaska u blok, tj. prebacivanje na drugi programski brojač, vraćanje na stanja koja se nalaze na stogu i postavljanje registara procesora [11]. Prolog je epilog, ali samo obrnuto definiran. Označava postavljanje varijabli i postavki prije nego se uđe u kod koji određuje programski brojač te on postavlja trenutačne vrijednosti u stogu kako bi se kasnije mogle rekreirati.

Kod epiloga je moguće napraviti optimizacije, jer se može utvrditi koje je sljedeće stanje koje je potrebno postaviti i isto tako koje su sljedeće operacije koje je potrebno izvršiti. Kako bi to brže funkcioniralo, QEMU radi optimizacije pomoću kojih spaja više TB kako bi se izvršili jedan za drugim bez da se treba vraćati u glavnu petlju gdje se provjerava sljedeći TB i je li on preveden.

Kako bi se postigla dodatna učinkovitost tijekom izvođenja QEMU-a, potrebno je isključiti ASLR (engl. *Address space layout randomization*) koji služi kao funkcionalnost koja postavlja sistemske naredbe na nasumične lokacije u memoriji kako bi se zaštitilo protiv napada preljevom memorije [12]. Ova postavka jezgre računala smanjuje učinkovitost TCG spremnika prevedenih blokova te rezultira time da se pojedini blokovi moraju više puta izvršavati što je nepotrebno.

3. Neizrazito testiranje

Neizrazito testiranje je metoda kojom se nastoje otkriti greške pomoću automatiziranog testiranja. Ovom metodom testiranja šalju se ulazni podaci koji se mogu razlikovati od programa do programa koji se testira.

Kako bi se razumjela problematika neizrazitog testiranja potrebno je objasniti koje vrste testiranja postoje te koje su razlike među njima. Iako postoje brojni alati za neizrazito testiranje, ovaj rad koristi AFL++ alat zbog brojnih prednosti koje alat omogućava.

3.1. Vrste neizrazitog testiranja

Pristupi testiranju programa se razlikuju primarno po količini informacija s kojima raspolaže inženjer i alat kojim se provodi testiranje. Alati kojima se provodi testiranje se mogu rasporediti u 3 grupe: *black-box*, *white-box* i *gray-box* [13] [14].

3.1.1. Black-box testiranje

Neizrazito testiranje metodom crne kutije (engl. *black box fuzzing*) je najjednostavniji način testiranja. Kako bi se provodila ova vrsta testiranja, od programa koji se testira je potrebno samo dobiti informaciju o grešci, ako se ona dogodila. Ne zahtijeva kompleksnije mehanizme koje su potrebne ostalim vrstama neizrazitog testiranja.

Potrebno je pažljivo odabrati i postaviti korijensku datoteku koja će se koristiti za generiranje promjena. Svakim generiranjem nove ulazne testne datoteke se primjenjuje određeni skup pravila kako bi se promijenila korijenska datoteka. Prvi alati su koristili jednostavne promjene kao što su okretanje određenog broja bitova, kopiranje ili brisanje odabranog dijela datoteke.

Noviji alati za neizrazito testiranje metodom crne kutije koriste generiranje ulaznih podataka pomoću gramatike ili informacije o formatu koji se šalje programu. Iako je ovaj način bolji od jednostavnih metoda, zahtijeva određeno znanje o alatu kojeg se testira i potrebno je dosta vremena kako bi se razvili ti mehanizmi.

Kod izrade novih ulaznih podataka je potrebno paziti nad kojom količinom podataka se rade promjene. Nije optimalno mijenjati prevelike skupove podataka korijenske datoteke jer su velike šanse da novi podaci neće zadovoljavati format koji je potreban alatu i vjerovano neće otkriti ništa novo. Ako se mijenja izrazito mali dio, tada će trebati puno iteracija prije nego se otkrije ranjivost. Proces određivanja pravog omjera koliki dio korijenske datoteke se želi mutirati može biti dugotrajan.

Prednosti:

- jednostavnost korištenja i postavljanja testiranja
- efektivan kod programa koji primaju binarne podatke kao ulaz. Ako program prima binarne podatke koji imaju strogo definirane lokacije koje određuju izvođenje tada se malim promjenama može značajno utjecati na izvođenje programa. Primjer toga je JPEG format slike. Potrebno je implementirati podršku za svaki novi format koji se želi podržati za testiranje.

Nedostatci:

- potrebno je puno računalnih resursa i vremena kako bi se otkrile greške
- ne pokriva se dovoljna količina koda

– efektivnost uvelike ovisi o kvaliteti korijenske datoteke (ili ulaza) koja se koristi kao uzorak za mutacije

– ako nisu dobro postavljeni početni uvjeti tada će biti puno lažnih dojava i teže će se otkriti prave ranjivosti koje se mogu nalaziti među njima.

3.1.2. White-box testiranje

Neizrazito testiranje metodom bijele kutije (engl. *White Box*) radi na principu da je izvorni kod mete koja se testira prethodno poznat. Cilj testiranja metodom bijele kutije je da se prođu sve putanje pomoću analize izvornog koda i njegovog grananja.

Prilikom pokretanja testiranja, prvo je potrebno prikupiti sve uvjetne naredbe koje određuju putanju izvođenja programa. Pokretanjem prvog testnog slučaja alat razlaže sva grananja na operacije Booleove algebre i zapisuje putanju pomoću nje. Zatim mijenja jedan po jedan od tih uvjeta putanje kako bi prošao što veći dio koda. Sljedeći ulazni podatak će se razlikovati u dijelu grananja kako bi se obišle prethodno nepoznate grane. Sve se može prikazati pomoću binarnog stabla koje se grana u dubinu.

Algoritam obilaska putanja u stablu će postavljati veći prioritet na putanje s najviše grananja kako bi se pokrilo što više neistraženih grana u što manje testova. Teoretski, testiranje metodom bijele kutije može dosegnuti potpunu pokrivenost koda, ali često radi kompleksnosti koda i velikog broja putanja koje postoje to nije slučaj.

3.1.3. Gray-box testiranje

Testiranje metodom sive kutije (engl. *Gray Box*) kombinira funkcionalnosti testiranja bijele i crne kutije. Ne zahtijeva cijeli izvorni kod, ali je potrebno nešto više informacije osim toga je li program završio greškom ili nije.

Jedna od korištenih informacija o izvođenju programa kojeg se testira je pokrivenost koda. Pokrivenost koda predstavlja informaciju o obiđenim putanjama i koliki dio od ukupnih putanja je otkriven. Pokrivenost se prikuplja tijekom izvođenja i temeljem te informacije se prilagođava sljedeći ulazni podatak kako bi otkrio što više neistraženih putanja.

Analiza mrlja (engl. *taint analysis*) je još jedna metoda koja se može koristiti. Koristi se kako bi se utvrdilo koji dijelovi ulaznih podataka mogu utjecati na grananja i to tako da se prati ulazni niz tijekom izvođenja programa. Metoda je efektivna u svrhe otkrivanja podataka koji su osjetljivi, a došli su u dio koda koji ne bi smio imati pristup tim podacima.

3.2. AFL

Alat AFL, kojeg je napravila kompanija Google, funkcionira na principu sive kutije koja prati tok izvođenja programa kojeg se testira. Prate se grane koda koje su se izvršile i broj puta koliko su te grane bile posjećene [15]. Podaci o granama koje su posjećene se čuvaju u polju veličine 64kB što znači da se kolizije mogu događati, no nisu toliko učestale. Kolizije se češće događaju na programima koji su veliki, te imaju puno grananja.

Zapisivanje u polje koje prati grananja se odvija principom prikazanim u isječku 3.1

Isječak 3.1: Aritmetika za praćenje izvođenja koda

```
1  cur_location = <COMPILE_TIME_RANDOM>;
2  shared_mem[cur_location ^ prev_location]++;
3  prev_location = cur_location >> 1;
```

U isječku 3.1 varijabla *shared_mem* predstavlja polje zapisa i povećava se vrijednost broja na indeksu koji odgovara rezultatu XOR operacije Booleove algebre između trenutačne lokacije osnovnog bloka i lokacije prethodnog bloka. Takvim određivanjem indeksa polja se omogućuje praćenje tranzicije iz nekog bloka A u B i u budućnosti kada se ponovi ta tranzicija, isti brojač će se povećati.

Posmicanje trenutačne lokacije za jedan u desno i spremanje rezultata kao prethodne lokacije (*prev_location*) služi kako bi se moglo razaznati između tranzicije A -> B i B -> A.

Algoritam praćenja stanja treba imati načine na koje otkriva koja stanja su značajna i doprinose pokrivenosti koda. Kako bi se odredilo koja izvođenja programa su interesantna, koriste se 2 tehnike.

Prva tehnika prati tranzicije između stanja. Ako putanja izvođenja sadrži neko stanje koje nije prethodno otkriveno, tada se ta putanja smatra interesantnom te posljedično i njezin ulazni niz koji je proizveo tu putanju. Druga tehnika koristi brojače unutar polja putanja. Interesantni slučajevi su kada brojač poprimi po prvi puta poprima vrijednost 1 za određeni indeks polja i to očito znači da je otkrivena nova putanja. Problem je razlikovati koje vrijednosti brojača znače bitnu promjenu za izvršavanje jer brojač može poprimati vrijednosti između 0 i 255. Ako se promijeni vrijednost brojača sa 150 na 151 to vjerojatno nije bitna promjena i nije otkriven novi bitan put. Kako bi se riješio taj problem, AFL koristi grupe koje su potencije broja 2 i ako se dogodi tranzicija iz jedne grupe u drugu tada se ta promjena smatra interesantnom. Primjer je kada se prijeđe s broja 5 (spada u grupu koja je veća ili jednaka od 2^2, a manja od 2^3) na broj 9 (spada u grupu koja je veća ili jednaka 2^3, a manja od 2^4). Testni slučaj se smatra zanimljivim ako zadovolji jedan od ova dva slučaja, tj. otkriva novu tranziciju ili brojač prelazi iz jedne grupe u drugu.

Ulazni parametri koji su otkrili novu tranziciju se pohranjuju u red čekanja kako bi se

mogli koristiti za kreiranje budućih mutacija. Novo dodani ulazni parametri će povećati skup podataka pomoću kojih se generiraju novi podaci i neće zamijeniti prethodno otkrivene podatke kako se ne bi gubile prethodno otkrivene informacije.

Algoritam koji je zadužen za odabir testnih slučajeva koji će se koristiti za generiranje sljedećeg ulaznog podatka svim članovima reda čekanja dodjeljuje ocjenu. Ocjena testa se temelji na trajanju izvođenja testa i veličini ulaznog podatka. Dodjeljivanje ocjena testnim slučajevima je napravljeno kako bi se mogli prioritizirati brži testovi.

AFL alat koristi brojne strategije prilikom generiranja novih ulaznih podataka. Determinističke strategije mijenjaju bitove na različitim lokacijama te mijenjaju brojeve u vrijednosti koje često izazivaju grešku. Osim determinističkih strategija se koriste i rječnici. Rječnici se koriste kako bi se prepoznala gramatika ulaznih podataka i na temelju nje generiralo što više validnih ulaznih podataka. Podržava najučestalije rječnike kao na primjer JavaScript, SQL ili XML. Može i automatski graditi rječnike dok promatra koje promjene bitova rezultiraju nevaljanim ulaznim podacima te na temelju promjena stvara jednostavan rječnik.

Određivanje grešaka i razlučivanje koje su greške jedinstvene, a koje nisu je bitan aspekt neizrazitog testiranja. Ako se gleda da su duplicirani padovi programa svi oni koji su se dogodili na istoj adresi, tada se može izgubiti puno bitnih informacija. Primjer gubljenja informacija bi bio kada se poziva neka biblioteka koja uzrokuje grešku. Biblioteka je mogla biti pozvana iz različitih dijelova koda i pad je mogao biti uzrokovan raznim ulaznim podacima. Ako bi se smatralo da je greška kada se sažeci stoga poklapaju može se dogoditi da program prati više grešaka nego što bi trebao. Primjer zašto ni taj način određivanja duplikata grešaka nije dobar je ako postoji funkcija koja rezultira greškom. Moguće je doći do te funkcije na puno različitih načina i sažetak stoga je različit, ali je uzrok greške jednak. AFL smatra da je greška u programu jedinstvena ako je neka nova grana otkrivena u padu ili jedna od grana nedostaje naspram prijašnjih padova.

AFL omogućuje testiranje programa kojima izvorni kod nije dostupan i to se može izvoditi pomoću QEMU alata. Podržan je rad s QEMU korisničkom emulacijom koja omogućuje pokretanje programa koji nisu nužno prevedeni za poslužiteljsko računalo.

3.3. AFL++

AFL++ je nastao kao projekt kojem je cilj unaprijediti rad AFL alata [16]. Autori AFL++ alata su smatrali da se ne razvija dovoljno na alatu AFL i htjeli su ga dodatno nadograditi s naprednijim funkcionalnostima. Tijekom pisanja ovog rada, AFL repozitorij nije imao promjene u glavnoj grani od lipnja 2021. godine.

Značajne nadogradnje:

- AFL++ dodaje funkcionalnosti predložene u AFLFAST repozitoriju. Promjene se odnose na to da se pokušavaju više testirati putanje koje se rjeđe ponavljaju.
- AFL++ podržava par različitih načina za praćenje i izvođenje koda: LLVM, GCC, QEMU, Unicorn i QBDI.
- Pomoću premosne komponente se može slati pokrivenost koda s bilo kojeg izvora prema alatu. Bitno je naravno da je pokrivenost koda (engl. *code coverage*) u formatu kojeg AFL++ očekuje, ali omogućuje fleksibilnost oko načina na koji se prikuplja.
- Kao nadogradnja na AFL uvodi dva nova načina kako bolje kontrolirati polje s brojačima koje predstavlja koji dijelovi koda su posjećeni. Problem u AFL-u je taj da kada je blok je posjećen 256 puta (ili neki broj koji je potencija broja 256) se dogodi preljev bita te se vrijednost brojača vraća na nulu što stvara probleme s praćenjem izvršavanja koda. Uvedena su dva rješenja: *NeverZero* i *Saturated Counters*. *NeverZero* služi kao mehanizam koji osigurava da ako je u bilo kojem trenutku vrijednost postavljana s 0 na 1 tada prati da ona nikada tijekom tog izvođenja ne može otići natrag na nulu. *Saturated Counters* služi kako bi se zamrznula vrijednost brojača kada dosegne vrijednost od 255 zato da se ne dogodi preljev. *NeverZero* se pokazao efikasniji jer manje utječe na brzinu za razliku od *Saturated Counters* rješenja pa je zato postavljen kao početni način rada, a *Saturated Counters* se može koristiti ako korisnik želi.
- AFL++ je implementirao QEMU podršku na novijoj verziji QEMU alata kojeg alat AFL ne podržava. Dodana je podrška za QAsana [17] koji prati greške koje se dogode u memoriji QEMU korisničke emulacije.

3.4. LibAFL

LibAFL je biblioteka koju je napravila ista zajednica koja radi na AFL++ alatu [18]. Prednosti korištenja ove biblioteke su da se može koristiti na svim sustavima koji podržavaju jezik Rust. Osim toga,pruža puno više mogućnosti od AFL++ alata jer se može programski prilagođavati meti testiranja.

Cilj projekta je da postoje brojne prethodno implementirane komponente koje se zatim mogu nadograđivati i povezivati međusobno, autori to opisuju kao izgradnju pomoću "Lego kockica".

Jedna od prednosti ove biblioteke je što se koristi programski jezik Rust. Rust je programski jezik s brojnim upotrebama, ali zadnjih godina se koristi sve više u programiranju ugradbenih računala radi njegovih performansi. Pomoću LibAFL biblioteke se može napisati *fuzzer* koji radi direktno na mikrokontroleru, PLC uređaju ili sličnome.

Rust podržava kompajliranje za druge arhitekture pomoću svojeg *cargo package managera* kao što je prikazano u isječku 3.2.

Isječak 3.2: Primjer prevođenja za drugu arhitekturu

```
cargo build --no-default-features --target aarch64-unknown-none
```

Iako je to značajna prednost, ovisi dosta o uređajima na kojima se pokreće jer je potrebno implementirati i pronaći način kako dodati praćenje koda i praćenje grešaka.

4. Implementacija

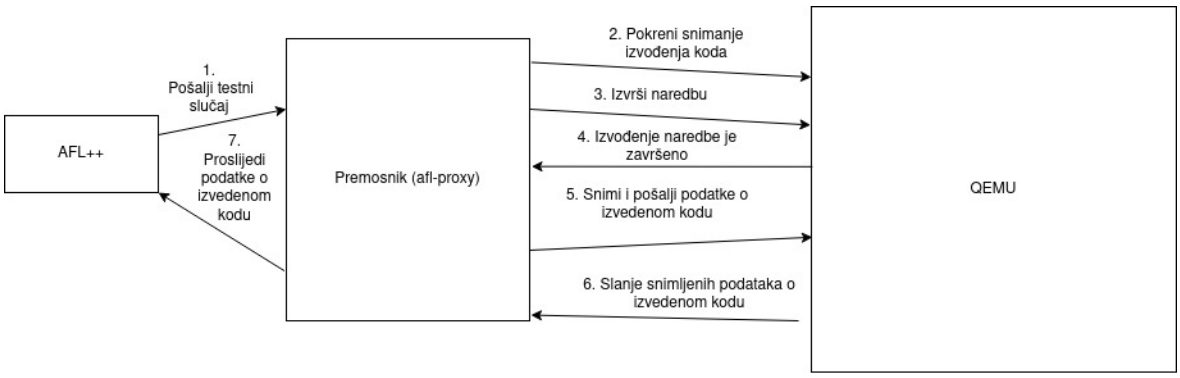
Cilj rada je postaviti testiranje programa unutar QEMU sustava pomoću AFL++ alata, ali to testiranje mora raditi na sistemskoj emulaciji. Trenutačno, AFL++ alat ne podržava neizrazito testiranje QEMU sistemske emulacije, već samo korisničke emulacije. Kako bi se zaobišlo to ograničenje, potrebno je napraviti premosnik između QEMU i AFL++ alata. Taj premosnik je zadužen za primanje ulaznih podataka od AFL++ alata, izvršavanje naredbi na sistemskoj emulaciji i predaja informacija o izvršavanju naredbi natrag AFL++ alatu. Pošto je AFL++ alat za neizrazito testiranje pomoću pokrivenosti koda potrebno je osmisliti mehanizam prikupljanja pokrivenosti koda unutar alata QEMU. Prvi dio poglavlja objašnjava promjene koje je potrebno napraviti u alatu QEMU kako bi se skupljale informacije o pokrivenosti koda, a drugi dio objašnjava načine na koje povezati modificirani QEMU s alatom AFL++.

Sve promjene koje su spomenute u ovom poglavlju se mogu pronaći unutar repozitorija *qemu_ivankovic* [19] i *afl-proxy* [20]

4.1. Arhitektura za testiranje

Slika 4.1 opisuje u najjednostavnijim crtama kako bi implementacija trebala izgledati. Potrebno je implementirati premosnik i napraviti izmjene u QEMU alatu. AFL++ alat treba cijelo vrijeme komunicirati s premosnikom gdje AFL++ alat "misli" kako provodi testiranje premosnika. Premosnik od primljenih testnih podataka stvara format u kojem želi poslati taj podatak u emulaciju. Prije nego pošalje testni podatak kao naredbu, treba pokrenuti stanje u kojem će se pratiti izvršavanje unutar QEMU emulacije. Nakon što je pokrenuo praćenje izvršavanja koda, šalje se naredba emulaciji. Emulirani sustav javlja kada je naredba izvršena, nakon čega premosnik treba zatražiti snimljene podatke od QEMU emulacije kako bi ih mogao proslijediti AFL++ alatu.

Premosnik u te svrhe mora imati osigurana 3 preduvjeta. Prvi je da može poslati naredbe za testiranje preko nekog sučelja. Drugi je da postoji mehanizam upravljanja QEMU alatom pomoću vanjskog sučelja. Treća komponenta je mehanizam pomoću kojeg će primati informacije o izvršenom kodu.



Slika 4.1: Skica arhitekture za testiranje sistemske emulacije

4.2. QEMU izmjene

Inspiracija za praćenje pokrivenosti koda u QEMU alatu je potekla iz članka "QEMU and U: Whole-system tracing with QEMU customization" [21] i modifikacija koje su napravljene na QEMU alatu u svrhe AFL++ projekta [22]. U članku su objašnjene osnove načina na koji alat AFL++ implementira praćenje pokrivenosti koda u QEMU-u, te dodaju svoje dodatne mehanizme. Članak je napravljen sa svrhom da se prati pokrivenost koda koja se dalje analizira u alatima poput IDA, Binary Ninja ili Ghidra, ali postavlja stabilne temelje sustava koji se može koristiti za praćenje koda koji se šalje AFL++ alatu. Također je korišten skup bilješki koje je napravila *Airbus security labs* grupa istraživača [23] u svrhu istraživanja unutrašnjosti QEMU alata.

QEMU je dinamički prevoditelj koji koristi TCG u pozadini kako bi prevodio kod s arhitekture koja se emulira na arhitekturu na kojoj se izvršava emulacija. Iako pojedini dijelovi TCG prevoditelja moraju biti implementirani specifično za arhitekturu koju on prevodi, esencijalni dijelovi za upravljanje prevođenjem su generični i moguće je na jednom dijelu koda dodati upravljanje TCG prijevodom koje će vrijediti za sve arhitekture. *qemuafl* je inačica QEMU alata koju je implementirala AFL++ zajednice te je ideja pratiti pokrivenost koda bilježeći programski brojač prilikom svakog prevođenja translacijskog bloka [22] [24].

Glavna petlja QEMU alata se nalazi u datoteci "accel/tcg/cpu-exec.c" te se petlja nalazi unutar funkcije *cpu_exec*. Funkcija postavlja temeljne vrijednosti emuliranog procesora, usklađuje satove s emuliranim sustavom i poslužiteljskim računalom te zatim ulazi u petlju koja se izvršava dokle god se ne prekine izvođenje ili se naiđe na pogrešku na koju se ne može prikladno reagirati. Unutar petlje se izvršavaju brojne operacije, ali zanimljive operacije u sklopu praćenja izvršavanja koda su *tb_lookup* i *tb_gen_code* koje su prikazane u isječku 4.1.

Isječak 4.1: Dio koda *cpu_exec* funkcije

```
1  tb = tb_lookup(cpu, pc, cs_base, flags, cflags);
2  if (tb == NULL) {
3    mmap_lock();
4    tb = tb_gen_code(cpu, pc, cs_base, flags, cflags);
5    mmap_unlock();
6    /*
7     * We add the TB in the virtual pc hash table
8     * for the fast lookup
9     */
10   qatomic_set(&cpu->tb_jmp_cache[tb_jmp_cache_hash_func(pc)], tb);
11 }
```

Funkcija *tb_lookup* pretražuje spremnik prethodno prevedenih blokova i ako utvrdi da je takav blok već preveden, skraćuje vrijeme izvedbe tako što preskače korak prevođenja i koristi prevedeni blok iz spremnika. Ako nije pronađen blok u spremniku, pokreće se funkcija *tb_gen_code* koja je zadužena da odradi prevođenje bloka instrukcija.

Bitno je da se u *tb_gen_code* funkciji napravi praćenje programskog brojača, jer kada bi se to napravilo u glavnoj petlji tada bi se pratile sve operacije koje događaju čak i one koje su sistemske i nebitne za svrhe testiranja. Cilj je da se sistemska emulacija nakon nekog vremena dovede u stanje gdje je većina sistemskih poziva i operacija prevedeno, kako ti podaci ne bi stvarali šum u podacima koji se koriste za praćenje dijela koji se testira.

Implementirane su pomoćne funkcije i stanja emulatora kako bi se definiralo kada treba pokrenuti praćenje prevođenja blokova i kada prenijeti snimljene podatke na drugu lokaciju. Datoteka bb-enter-helper.c sadrži definiciju stanja, funkcije koje postavljaju stanja i funkciju za dohvaćanje stanja. Vrijednosti stanja ukazuju na sljedeće događaje:

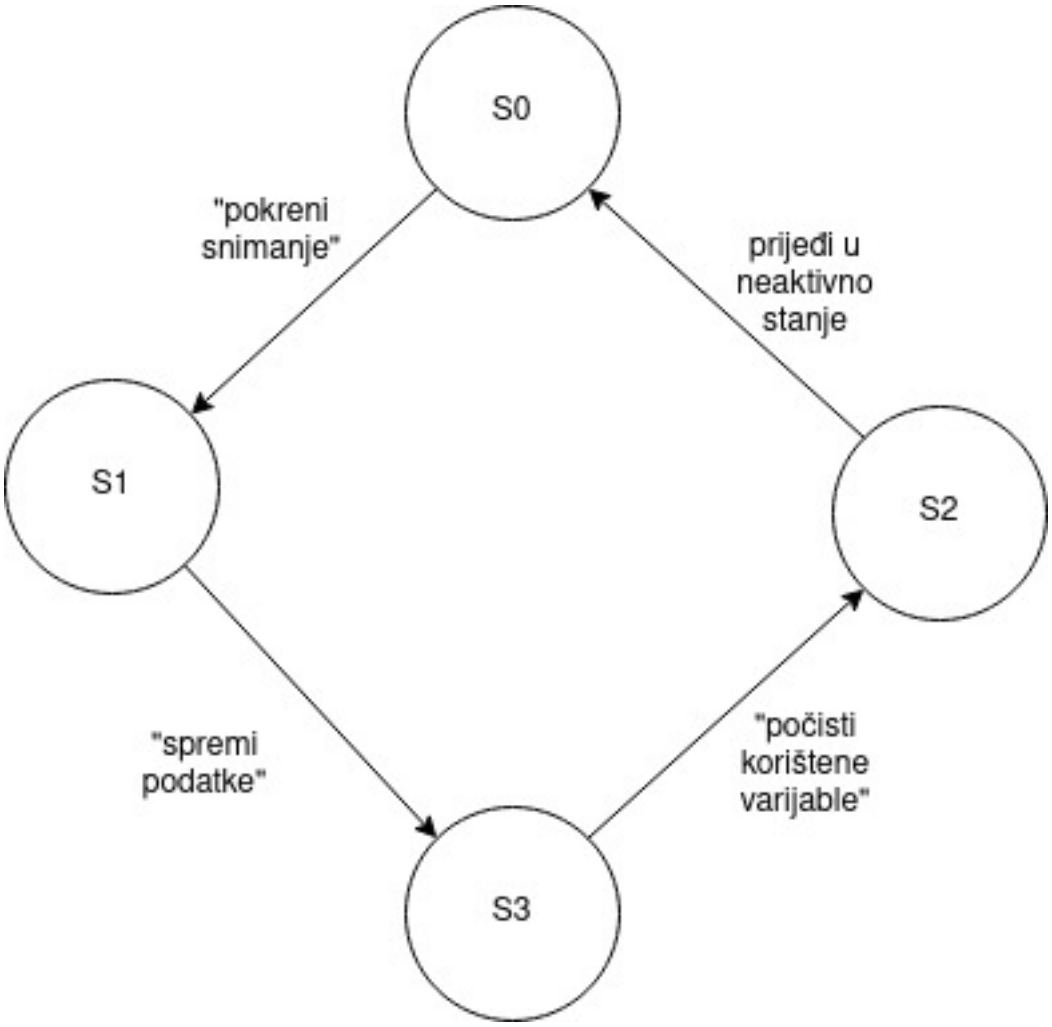
stanje = 0 : neaktivno stanje i ne bilježe se podaci o prevedenim blokovima

stanje = 1 : stanje snimanja te se bilježi sve što se prevodi

stanje = 2 : stanje zaustavljanja. Signalizira da se trebaju sve korištene varijable postaviti na početne vrijednosti i zatim prijeći u neaktivno stanje (stanje 0)

stanje = 3 : stanje snimanja. Potrebno je spremiti podatke i prenijeti ih kako bi se nastavilo s daljnjim radom. Nakon što su podaci o prevedenim blokovima preneseni i uspješno pohranjeni, prelazi se u stanje zaustavljanja.

Ovi prijelazi su prikazani na dijagramu stanja na slici 4.2



Slika 4.2: Dijagram stanja (broj uz slovo S predstavlja vrijednost varijable stanja)

Funkcija koja se poziva tijekom svakog prevođenja bloka instrukcija je *bb_enter()* koja prima programski brojač kao argument. Funkcija prati stanja i radi operacije koje su potrebne za to stanje. Funkcija je prikazana u isječku 4.2.

Isječak 4.2: *bb_enter()* funkcija

```
void bb_enter(target_ulong cur_loc)
{
    int state = get_started_bb();
    // state 0 means that logging is stopped and we don't need to
       call anything
    if (state == 0) {
       return;
    // state 2 means that we want to clear all data stored
       currently and then step to state 0
    } else if (state == 2){
       afl_prev_loc = 0;
       memset(afl_area_ptr, 0, MAP_SIZE);
       stop_bb_enter();
       return;
    } else if (state == 3) {
       client_send_bitmap();
       // clear_bb_enter will set state value to 2 and in next
          iteration it will clear
       clear_bb_enter();
    }
    cur_loc = (uintptr_t)(afl_hash_ip((uint64_t)cur_loc));
    cur_loc &= (MAP_SIZE - 1);

    TCGv cur_loc_v = tcg_const_tl(cur_loc);
    gen_helper_afl_maybe_log(cur_loc_v);
    tcg_temp_free(cur_loc_v);
}
```

Varijable *afl_area_ptr* je referenca na polje koje se koristi kako bi se pratili blokovi koji su prevedeni i svi brojevi unutar tog polja su brojači koji predstavljaju koliko puta je izvršen određeni blok. Nakon što stanje *bb-enter* postane vrijednost 2, tada je potrebno postaviti prethodni blok na vrijednost 0 i također postaviti polje brojača blokova na početnu vrijednost. Ako je vrijednost stanja snimanja jednaka broju 3 tada QEMU treba poslati polje brojača preko pristupne točke koja se nalazi na vratima 5001, a s te pristupne točke će kasnije čitati podatke premosnik između AFL++ alata i QEMU alata. Ako je stanje 1, potrebno je odrediti indeks člana polja kojem će se povećati brojač, ali tako da je taj indeks povezan s vrijednošću programskog brojača. Alat AFL koristi jednostavne operacije kojima se postiže veća uniformna raspoređenost u polju kako bi povećavao brojače po cijelom polju, a ne

samo nekim njegovim dijelovima. Mehanizam na koji se povećava uniformnost je prikazan isječkom 4.3.

Isječak 4.3: AFL mehanizam za povećavanje vrijednosti brojača

```
cur_location = (block_address >> 4) ^ (block_address << 8);
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

To postiže tako da posmiče adresu bloka za 4 mjesta udesno i 8 mjesta u lijevo te nad time vrši XOR operaciju i to pohranjuje u varijablu *cur_location*. Nakon toga radi XOR operaciju s prethodnom lokacijom i na tom indeksu povećava brojač unutar polja. AFL++ alat koristi dodatne mehanizme koji su preuzeti iz drugog projekta pod nazivom xxhash/XXH3 [25] koji implementira brz algoritam za sažimanje podataka. Za implementaciju korištenu u ovom projektu preuzete su *afl_hash_ip* i povezane funkcije iz AFL++ implementacije kako bi se postigla što veća uniformnost podataka.

Unutar *bb_enter* funkcije se koristi *HELPER* funkcija koja prima cijeli broj bez predznaka od 32 bita i povećava vrijednost na određenom indeksu na temelju adrese. Također postavlja varijablu prethodnog bloka koja je jednaka trenutačnom bloku posmaknutim za 1 u desno. *HELPER* funkcija je prikazana u isječku 4.4.

Isječak 4.4: HELPER funkcija koja inkrementira indeks u polju brojača

```
void HELPER(afl_maybe_log)(target_ulong cur_loc) {
   register uintptr_t afl_idx = cur_loc ^ afl_prev_loc;
   INC_AFL_AREA(afl_idx);
   afl_prev_loc = cur_loc >> 1;
}
```

4.2.1. TCG izmjene

Nakon što je implementirana funkcionalnost snimanja obrađenih blokova, potrebno je razmotriti tu funkcionalnost u kontekstu TCG dinamičkog prevodioca. Funkcija za snimanje blokova je smještena u dio koda koji je zadužen za prevođenje bloka. Ako je neki od blokova prethodno preveden, tada će se preskočiti poziv funkcije *bb_enter* što nije dobro za praćenje izvršavanja blokova. Samo tijekom prvog prevođenja će se povećati brojač i nakon tog prevođenja translacijski blok će se nalaziti u spremniku prevedenih blokova. To narušava pouzdanost praćenja prevedenih blokova, te je potrebno implementirati način na koji će se zaobići to ograničenje. Također, ako je sustav dovoljno dugo pokrenut, svi blokovi koji se izvršavaju će biti prevedeni i implementirano praćenje blokova će se vrlo rijetko pozivati zato što će rijetko trebati prevoditi nove blokove.

Najprimitivnije rješenje je ukloniti sve prethodno prevedene blokove i to se treba napraviti prije svakog početka snimanja prevođenja blokova. Takva funkcionalnost se može implementirati pozivom funkcije *tb_flush()* koja kao parametar prima varijablu stanja procesora. Time se može osigurati da će se snimanje blokova sigurno pokrenuti i neće zastati neko duže vrijeme jer će imati mnoštvo blokova za prevesti. Postoje brojne mane takvog pristupa, ali tri su najbitnije:

- izrazito se usporava sustav jer su pobrisani svi prethodno prevedeni blokovi, čak i oni koji su prevedeni radi pozadinskih procesa
- blokovi pozadinskih procesa će se trebati ponovo prevesti što će dodati šum u snimljenim blokovima
- ako se neki blok jednom prevede, a bitan je za program koji se testira, tada će se samo jednom zabilježiti u polje koje prati prevedene blokove i sva ostala pojavljivanja tog bloka će se izvršavati bez poziva za povećanje brojača

Rješenje koje je zadovoljavajuće je da se dozvoli sistemskoj emulaciji par minuta da se većina sistemskih poziva i procesa koji se izvršavaju u pozadini izvrši i njihovi prevedeni blokovi pohrane u memoriju. Izvršavanje pozadinskih procesa može uzrokovati smetnje u snimljenom prometu tijekom snimanja izvođenja koda. Nakon što su nebitni osnovni blokovi prevedeni, prihvatljivo je da ostaju u spremniku zato što nisu od značaja za testiranje, a zato što su prethodno prevedeni povećaju brzinu izvođenja operacija u emulaciji. Tijekom praćenja koda koji se izvodi, potrebno je onesposobiti spremanje prevedenih translacijskih blokova kako bi se riješilo svih prethodno navedenih mana.

Jednostavnim dodavanjem provjere stanja snimanja u funkciju *tb_link_page()* se može zaustaviti snimanje te je potrebno dodati provjeru uvjeta stanja. Spremanje u spremnik prevedenih blokova želimo onesposobiti ako je QEMU u stanju 1, tj. pokrenuto je snimanje izvršavanja koda. U tu svrhu dodana je provjera na početak funkcije *tb_link_page()* kao što

je prikazano u isječku 4.5.

Isječak 4.5: Provjera spremanja translacijskog bloka u spremnik

```
1 static TranslationBlock *tb_link_page(TranslationBlock *tb,
     tb_page_addr_t phys_pc, tb_page_addr_t phys_page2)
2 {
3       if (get_started_bb() == 1) return tb;
4       ...
5 }
```

Funkcija *tb_link_page* kao izlaz vraća strukturu *TranslationBlock* i radi toga je ovaj uvjet na vrhu i ne radi nikakve operacije nad translacijskim blokom koji je argument već ga samo vraća.

4.2.2. QMP i HMP naredbe

Nakon implementacije snimanja izvođenja koda, potrebno je napraviti mehanizam koji će sluziti da vanjske komponente mogu pokrenuti promjene stanja unutar QEMU alata. QEMU ima svoj protokol koji služi da se pomoću JSON paketa, upravlja stanjem emulacije te se taj protokol naziva QMP (*QEMU Machine Protokol*). Usko povezano s QMP protokolom je HMP (*Human Monitor Interface*) koji omogućava pozivanje definiranih naredbi preko monitor sučelja. Kako bi se pozvala funkcija koja početno nije definirana u QMP i HMP sučeljima, potrebno je dodati definicije tih funkcija u sam izvorni kod i implementirati njihovo ponašanje. Srećom, funkcije koje su potrebne u sklopu ovog rada su izrazito jednostavne i potrebno je implementirati samo 4 naredbe.

start-bb-enter: Postavlja stanje na vrijednost 1, te će snimanje početi prvim pozivom prevođenja osnovnog bloka

stop-bb-enter: Postavlja stanje na vrijednost 0 i snimanje će se zaustaviti

clear-bb-enter: Postavlja stanje na vrijednost 2 i u tijekom sljedećeg prevođenja osnovnog bloka će se varijable koje su korištene postaviti na početne vrijednosti

save-bb-enter: Postavlja stanje na vrijednost 3 te će se pokrenut proces spremanja snimljenih podataka o izvođenju.

Kako bi se otvorila vrata za komunikaciju pomoću QMP protokola potrebno je prilikom pokretanja emulacije dodati argumente navedene u isječku 4.6.

Isječak 4.6: Argument za otvaranje QMP protokola i pripadnih vrata

```
-qmp tcp:localhost:4444,server,wait=off
```

Zatim se pozivom na vrata 4444 na lokalnoj mreži može poslati zahtjev za izvršavanjem naredbi. Potrebno je u tijelo zahtjeva dodati naredbu forme prikazane u isječku 4.7.

Isječak 4.7: Primjer izgleda JSON objekta u zahtjevu

```
{
    "execute" : "ime-naredbe"
}
```

4.3. AFL++ premosnik

Za implementaciju premosnika u ovom radu se koristi predložak koji se može pronaći u službenom repozitoriju AFL++ alata na putanji "utils/alf_proxy". Značajno olakšava pisanje premosnika, jer postavlja sve stvari koje su potrebne za rad s AFL++ alatom i zato je potrebno samo implementirati funkcije koje se izvršavaju nakon što se prime testni podaci od AFL++ alata.

afl-proxy u glavnoj funkciji postavlja okruženje prije nego što krene dohvaćati testne podatke. Početak glavne funkcije je prikazan u isječku 4.8.

Isječak 4.8: Dio glavne funkcije unutar *afl-proxy* datoteke

```
int main(int argc, char *argv[]) {
  /* This is were the testcase data is written into */
  u8 buf[1024]; // this is the maximum size for a test case! set
     it!
  s32 len;
  init_bitmap_socket();
  init_qmp_communication();
  init_bind_shell_connection();
  /* here you specify the map size you need that you are reporting
     to
    afl-fuzz. Any value is fine as long as it can be divided by
       32. */
  __afl_map_size = MAP_SIZE; // default is 65536

  /* then we initialize the shared memory map and start the
     forkserver */
  __afl_map_shm();
  __afl_start_forkserver();
  /* ostatak koda se prolazi kasnije */
}
```

Prvo se postavlja polje u koje će se spremati testni podaci preuzeti od AFL++ alata. Sljedeće 3 funkcije su implementirane u svrhu razmjene informacija između dvaju sustava. *init_bitmap_socket* funkcija postavlja potrebne varijable i inicijalizira vezu koja će se koristiti za primanje snimljenih podataka o izvođenju programa. Za primanje podataka se koristi obična TCP pristupna točka koja se otvara na vratima 5001.

init_qmp_communication je funkcija koja je zadužena za spajanje na vrata 4444 koja se koriste za komunikaciju QMP protokolom. Kako bi se ostvarile dodatne performanse,

također se postavljaju varijable koje čuvaju JSON objekte koji će se koristiti za slanje naredbi preko ovog sučelja, a to su objekti za pokretanje snimanja i slanje snimanja prometa.

init_bind_shell_communication postavlja TCP pristupnu točku preko koje će se slati naredbe koje se izvode unutar sistemske emulacije. Ova komponenta se treba mijenjati u ovisnosni o tome što se testira. Ako se želi testirati neki alat unutar korisničke ljuske, tada se može koristiti ova funkcija, ali ako se testira primjerice web poslužitelj tada se ne treba koristiti ova funkcija već implementirati funkcije za komunikaciju s web poslužitelj.

Zatim se postavlja varijabla *__afl_map_size* koja je dijeljena između *afl-proxy* alata i AFL++ alata. Tom varijablom je određena veličina polja u koje se bilježe podaci o izvoženju. Osnovna vrijednost je 65536 što je jednako kao 2^{16} što odgovara veličina polja koja se koristi za snimanje i u alatu QEMU.

Funkcija *__afl_map_shm* služi kako bi se postavila dijeljena memorija koja se koristi za spremanje podataka o izvođenju koda. *__afl_start_forkserver* inicijalizira *forkserver* koji je optimizacija koju AFL++ koristi. Služi za pokretanje roditeljskog procesa koji je stalno aktivan, a djeca pokreću testne slučajeve. Time se ubrzava izvršavanje jer AFL++ alat ne treba svaki put inicijalizirati sve varijable koje su potrebne za izvršavanje i pozivati *alf-proxy* pomoću *execl* funkcije. Najbitniji dio koda je dio koji se izvršava unutar petlje u glavnoj funkciji i on je prikazan u isječku 4.9.

Isječak 4.9: Petlja unutar glavne funkcije

```
1    while ((len = __afl_next_testcase(buf, sizeof(buf))) > 0) {
2       if (len > 1 && buf[0] != 0) {
3         buf[len] = '\0';
4
5         // first we send command to QEMU to start recording
6         start_qmp_command();
7
8         // send command to bind shell
9         size_t write_result = write(sockfd_bind_shell, buf,
              strlen(buf));
10
11        int n;
12        memset(rcvBuff, 0, BUFF_LEN);
13        while (1) {
14          if ((n = read(sockfd_bind_shell, rcvBuff,
                sizeof(rcvBuff))) > 0) {
15            break;
16          }
17        }
18        status = atoi(rcvBuff);
19
20        save_hmp_command();
21
22        accept_bitmap();
23
24        strncpy(__afl_area_ptr, bitmap, MAP_SIZE);
25
26        memset(bitmap, '0', sizeof(bitmap));
27        memset(buf, '0', sizeof(buf));
28      }
29
30      /* report the test case is done and wait for the next */
31      if (status != 0) {
32        __afl_end_testcase(137);
33      } else {
34        __afl_end_testcase(0);
35      }
36    }
```

Izvodi se petlja dokle god se primaju testni slučajevi od strane AFL++ alata. Kada *afl-proxy* primi sljedeći testni slučaj, sprema ga u polje *buf*.

Program koji se testira unutar emulacije prima nizove znakova kao ulazni parametar. Zbog toga je potrebno provjeriti da se na početku ulaznog niza znakova na nultom indeksu ne nalazi *NULL byte* koji znači kraj niza. Nije korisno dozvoljavati testove koji su kraći od jednog znaka. U varijablu *len* je prilikom dohvaćanja testnih podataka je primljena dužina testnog podatka pa kako bi se ti znakovi koji nužno nisu u formatu niza znakova pretvorili u niz znakova, na indeks *len* postavlja se *NULL byte*.

Nakon toga je potrebno pokrenuti snimanje *start_qmp_command()* funkcijom koja će poslati zahtjev s JSON tijelom kako bi pokrenula stanje snimanja unutar emulacije. Jednom kada je snimanje pokrenuto potrebno je poslati testni niz emulaciji na pristupnu točnu na kojoj očekuje naredbe. Zatim emulacija odgovara na testni slučaj sa statusnim kodom koji je 0 ako je naredba izvršena uspješno ili neka druga vrijednost ako se pojavila pogreška.

Sljedeća se pokreće naredba *save_hmp_command()* koja će poslati zahtjev QEMU-u da prenese podatke o snimljenom prometu preko TCP pristupne točke. Funkcija *accept_bitmap()* služi kako bi slušala promet koji će QEMU poslati na definirana vrata 5001. Nakon toga, *afl-proxy* ima podatke o izvođenju koda spremljene u polje koje se zove *bitmap* te je potrebno izjednačiti polje na koje pokazuje pokazivač *__afl_area_ptr* s poljem *bitmap*. Podaci se pomoću *strncpy* naredbe kopiraju u polje koje je dijeljeno između *afl-proxy* i AFL++ alata.

Nakon izjednačavanje vrijednosti dvaju polja, polja *bitmap* i *buf* su nepotrebna te se postavljaju na početne vrijednosti kako bi se pripremila za korištenje u sljedećem testnom slučaju.

Varijabla *status* služi kako bi se u nju spremila vrijednost o grešci. Vrijednost može biti izlazni kod kao na operacijskim sustavima Unix ili može biti ručno postavljena, ako se na neki drugi način detektira greška. Ukoliko je vrijednost *status* varijable različita broju 0, dogodila se pogreška i potrebno je grešku dojaviti alatu AFL++ pomoću funkcije *__afl_end_testcase()*.

Funkcija *__afl_end_testcase()* služi kako bi se dojavila pogreška i ujedno prešlo na sljedeći testni slučaj. Brojni statusni kodovi se koriste kako bi se roditeljskom procesu dojavile greške koje su se dogodile i od kojih se proces ne može oporaviti i nastaviti s radom. Jedan statusni kod koji ne znači ne oporavljivu grešku, već da je roditeljski proces samo zabilježi i nastavi s daljnjim izvođenjem je kod 137 koji označava *Out-of-memory* na operacijskim sustavima Linux. Mogu se koristiti razni kodovi, ali ovo je prvi kod koji je pronađen da ne rezultira prisilnim zaustavljanjem testiranja.

5. Eksperiment

5.1. Postavljanje sistemske emulacije

Prije eksperimenta je potrebno pokrenuti izgradnju svih datoteka potrebnih za emulaciju. U eksperimentu se koristi ARM procesor pa je potrebno postaviti argumente prije samog prevođenja. Može se koristiti bilo koji drugi procesor, ali ovaj se obrađuje u sklopu rada.

Prvo je potrebno pozicionirati se u build direktorij i zatim pokrenuti configure skriptu. Ako se želi isključiti prevođenje QEMU alata za sve podržane arhitekture, potrebno je eksplicitno definirati koje arhitekture se žele prevesti te zatim pokrenuti naredba za izgradnju svih potrebnih datoteka.

U sljedećem primjeru se koristi -j 4 argument kako bi se pokrenulo višedretveno izvođenje, čime se izgradnja QEMU alata značajno ubrzava jer je alat izrazito velik. Potrebne naredbe su navedene u isječku 5.1.

Isječak 5.1: Prevođenje izmijenjenog QEMU repozitorija

```
1 cd build
2 ../configure --target-list=aarch64-softmmu,arm-softmmu
3 make -j 4
```

Kako bi se prikladno testirao kod koji je objašnjen u prethodnom poglavlju, potrebno je postaviti primjer sistemske emulacije na kojoj će se testirati neki primjer programa.

Primjer emulacije koja će se koristiti u ovom radu je preuzet s repozitorija qemu-images. Slika koja je preuzeta iz repozitorija je slika Raspberry Pi uređaja, koja se koristi sistemsku emulaciju ARM procesora [26].

Potrebno je podesiti programske argumente prije pokretanja slike, kako bi što bolje radila s napravljenom implementacijom praćenja.

Isječak 5.2: Pokretanje sistemske emulacije iz programske ljuske *bash*

```
1 $PUTANJA_DO_QEMU/build/qemu-system-arm -s -M versatilepb -cpu
     arm1176 -d nochain \
2  -accel tcg,tb-size=256 \
3  -hda 2012-07-15-wheezy-raspbian.img -kernel kernel-qemu -m 192
     -append "root=/dev/sda2" \
4  -qmp tcp:localhost:4444,server,wait=off \
5  -nographic \
6  -nic user,hostfwd=tcp:127.0.0.1:1080-:80
```

Objašnjenje argumenata naredbe navedenih u isječku 5.2 [27]:

$PUTANJA_DO_QEMU se odnosi na putanju do direktorija izmijenjenog QEMU alata koji je izgrađen pomoću naredbe *make*

qemu-system-arm se koristi za pokretanje sistemske emulacije s ARM procesorom

-s se koristi za otvaranje GDB sučelja koje se može koristiti za daljnje istraživanje. Vrata koja će se otvoriti su po početnim postavkama na lokalnom poslužitelju vrijednosti 1234. Ovaj argument nije nužan.

-M definira koja matična ploča se emulira

-cpu definira procesor koji se emulira

-d nochain služi kako bi se isključila opcija koja je bila opisana u TCG poglavlju. Ovom opcijom se isključuje optimizacijski mehanizam povezivanja više osnovnih blokova, ali je za svrhe testiranja bolje da je isključeno povezivanje.

-accel tcg,tb-size=256 postavlja veličinu spremnika za čuvanje prevedenih blokova na 256 MB

-hda definira putanju do datoteke sa slikom operacijskog sustava

-kernel definira putanju do datoteke koja sadrži jezgru sustava

-m opcija dodjeljuje emulaciji 192 MB radne memorije

-append dodaje korijensku particiju na "/dev/sda2"

qmp tcp:localhost:4444,server,wait=off otvara vrata na lokaciji 4444 na lokalnom poslužitelju koja će se koristiti za komunikaciju pomoću QMP protokola

-nographic isključuje grafičko sučelje i sve što se izvodi se može vidjeti u programskoj ljusci (engl. *shell*)

-nic user,hostfwd=tcp:127.0.0.1:1080-:80 omogućava da se promet koji se odvija unutar emulacije na vratima 80 prosljeđuje na vrata 1080 poslužiteljskog računala. Ova spojna točka će se koristiti u svrhe slanja naredbi.

5.2. Pristup udaljenoj ljusci

U prijašnjem potpoglavlju je spomenuto da se otvaraju vrata 80 u emulaciji i prosljeđuju na vrata 1080 na poslužitelju. Preusmjeravanje je napravljeno kako bi se mogle slati naredbe u ljusku emulacije, a naredbe može slati bilo koja komponenta koja ima mogućnosti komunikacije preko poslužiteljskih vrata.

Implementirana je vrsta ljuske koja se naziva *bind shell*. *Bind shell* obično koriste napadači kako bi imali udaljeni pristup ljusci drugog računala tako da *bind shell* drži otvorena vrata do kojih napadač može pristupiti [28].

Program koji postavlja bind *shell* treba čitati naredbe koje dolaze na vrata 80 unutar emulacije i *afl-proxy* će slati naredbe na vrata 1080 na poslužiteljskom računalu. Kako bi se implementirao jednostavan način dojavljivanja greške može se koristiti izlazni podatak nakon izvođenja naredbe koji se naziva *exit code*. Operacijski sustavi Linux i Unix koriste izlazne kodove kako bi javili rezultat izvođenja naredbe ili programa. Iako kodovi nisu jako deskriptivni dovoljni su u ovom slučaju da pruže informaciju je li program rezultirao greškom.

Isječak 5.3: Primjer jednostavnog *bind shella*

```
import socket
import os

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('0.0.0.0', 80))
s.listen(10)
conn, addr = s.accept()
print("Connection accepted.\n")
while 1:
    data = conn.recv(1024)
    command_string = data.decode("utf-8", "ignore").strip()
    command_string = command_string.replace("\'", "")
    if (command_string == ""):
        conn.send((str(0).encode()))
    res = os.system("./test1 \'"+ str(command_string) + str("\'
        >/dev/null 2>&1"))
    conn.send(str(res).encode())
conn.close()
```

Isječak 5.3 prikazuje jednostavnu implementaciju *bind shella* koja sluša na vratima 80. Kada primi podatke, pretvara ih u *utf-8* i zanemaruje znakove koji nisu dio *utf-8* standarda,

uklanja prazne znakove s početka i kraja uklanja " ’ " znak. Znak " ’ " se koristi prilikom slanja argumenata kako bi se izbjeglo izvodenje više naredbi kao na primjer kada ulazni niz sadrži točka zarez znak. Zbog toga je potrebno maknuti sve " ’ " iz ulaznog niza kako se ne bi poništila dva koja okružuju ulazne podatke.

Ako je nakon izbacivanja nepotrebnih podataka naredba postala prazan tekst, potrebno je vratiti broj 0 kako bi se prešlo na sljedeći testni podatak.

Zatim se ti podaci koriste kao ulazni argumenti za program *test1* i izlazni kod se vraća pošiljatelju naredbe (to će biti *afl-proxy*).

Isječak 5.4 pokazuje potrebnu varijablu okoline kako bi se koristili samo UTF-8 znakovi, jer se inače pojavljuju problemi s izvođenjem.

Isječak 5.4: Pokretanje bind.py programa

```
1 PYTHONIOENCODING=utf-8 python3 bind.py
```

5.3. Pokretanje testiranja

Prije pokretanja testiranja, potrebno je odrediti metu testiranja. Kao meta testiranja je napisan jednostavan kod koji, ako primi ulazni podatak koji počinje znakom 2 ili 3, rezultira greškom koja se događa kada se pristupa dijelu memorije kojem ne bi smjelo. Kod je prikazan u isječku 5.5

Isječak 5.5: Program *test1.c* koji može uzrokovati *segmentation fault*

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
/* Example for seg_fault taken from:
https://www.geeksforgeeks.org/core-dump-segmentation-fault-c-cpp/
*/
void seg_fault() {
  char *str;

  str = "Something";
  *(str + 1) = '!';
}

int main(int argc, char *argv[]) {
  char buf[100];
  strcpy(buf, argv[1]);
  buf[strlen(buf)] = '\x00';

  if (buf[0] == '2') {
    seg_fault();
  }
  if (buf[0] == '3') {
    seg_fault();
  }

  return 0;
}
```

Kako bi se testiranje preko AFL++ alata moglo pokrenuti prvo se treba pokrenuti sistemska emulacija pomoću ranije navedene naredbe. Zatim je potrebno pokrenuti bind shell unutar emulacije kako bi se mogle slati naredbe.

AFL++ alat mora biti postavljen na računalu. Radi jednostavnosti korištena je lokalno

izgrađena verzija AFL++ alata, a upute se mogu pronaći u službenoj dokumentaciji u INSTALL.md datoteci u repozitoriju.

Zatim je potrebno pozvati AFL++ alat pomoću sljedeće naredbe:

Isječak 5.6: Naredba korištena za pokretanje testiranja

```
AFL_SKIP_CPUFREQ=1 AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
    AFL_DEBUG=1 $PUTANJA_DO_AFLPLUSPLUS/afl-fuzz -i
    $PUTANJA_DO_ULAZNIH_DATOTEKA -o $PUTANJA_DO_IZLAZNIH_DATOTEKA
    -t 5000 -- ./afl-proxy
```

Objašnjenje korištenih opcija u isječku 5.6 [29]:

AFL_SKIP_CPUFREQ varijabla je postavljena na vrijednost jedan zato što AFL++ detektira da postavke procesora na kojem se izvodi testiranje nisu optimalne i daje upute kako ih postaviti, no zahtijeva mijenjanje osjetljivih dijelova sustava, pa se ovim argumentom može preskočiti provjera postavki. Gubi se na učinkovitosti.

AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES varijabla je postavljena zato što se padovi programa neće prijavljivati na očekivani način, već će ih *afl-proxy* ručno javiti

AFL_DEBUG kako bi imali detaljnije informacije o tome što se događa tijekom testiranja

$PUTANJA_DO_AFLPLUSPLUS je put do direktorija u kojem se nalazi prethodno preuzet repozitorij i u njemu se nalazi izgrađena afl-fuzz binarna datoteka

-i $PUTANJA_DO_ULAZNIH_DATOTEKA specificira direktorij u kojem se nalaze početni testni slučajevi od kojih će se krenuti s mutacijom podataka

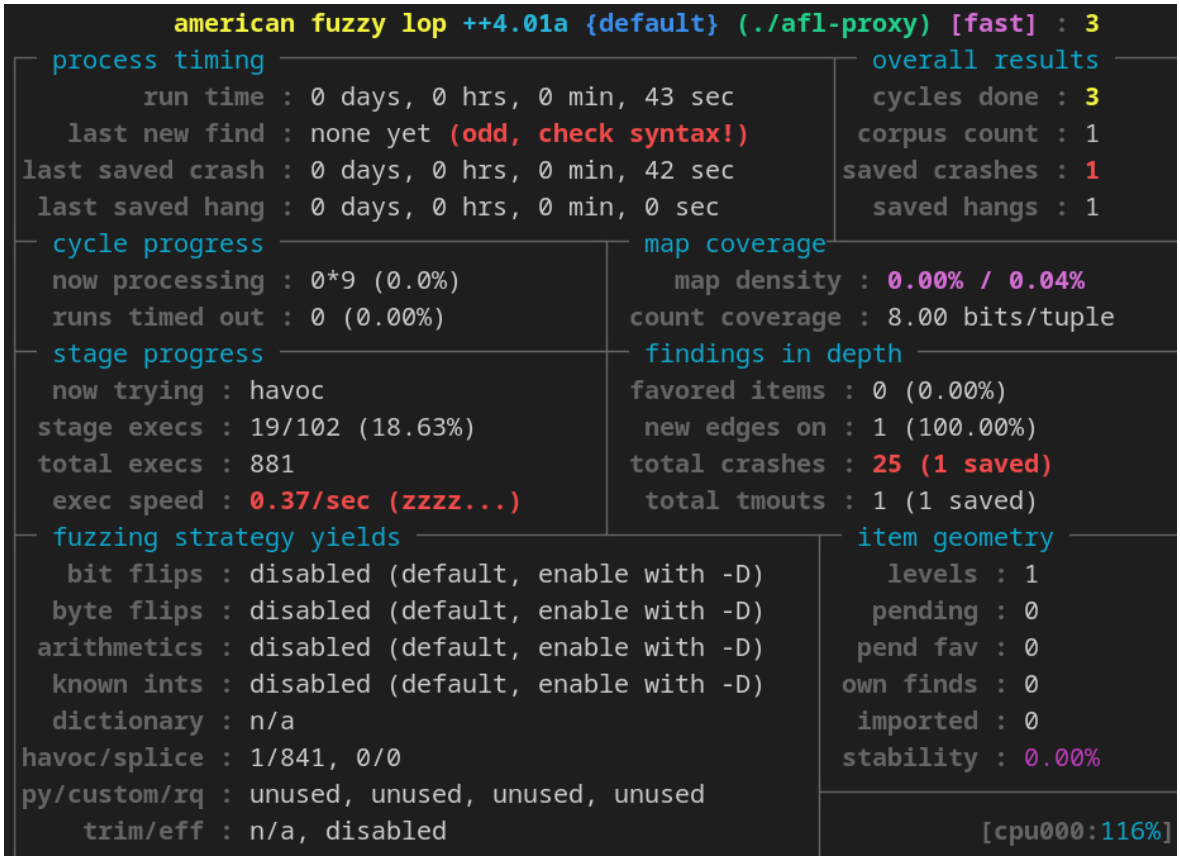
-o $PUTANJA_DO_IZLAZNIH_DATOTEKA određuje direktorij koji AFL++ koristi kao radni direktorij i tamo se na kraju testiranja mogu pronaći pogreške koje su pronađene

-t 5000 postavljanje vrijeme nakon kojeg se test smatra neuspješnim, u ovom slučaju 5 sekundi

-- ./afl-proxy definira koja datoteka se testira, a to je implementirani premosnik. AFL++ zapravo ne zna da testira unutar QEMU sistemske emulacije

Unutar direktorija za ulazne datoteke je potrebno napraviti datoteku s prvim testnim slučajem koji će se koristiti za daljnje generiranje ulaznih podataka. Za ovaj jednostavan program je napravljena datoteka koja u sebi sadrži "1" vrijednost. Takva početna vrijednost neće rezultirati pogreškom tijekom prvog izvođenja, no dobar je temelj kako bi testiranje došlo do vrijednosti 2 ili 3 kao prvog znaka.

Pozivanjem naredbe se pojavljuje monitor preko kojeg se mogu vidjeti dodatne informacije o toku testiranja.



Slika 5.1: AFL++ monitor za pregled stanja testiranja

Na slici 5.1 se može vidjeti puno podataka. Monitor prikazuje kako je bilo 881 pokretanja testova u 43 sekunde. Bitno je uočiti kako je alat pronašao 25 grešaka, ali samo jednu spremio jer je zaključio da su ostale 24 greške duplikati. Greška se može pronaći unutar datoteke koja se nalazi u *output/default/crashes* direktorija. Vrijednost datoteke unutar tog direktorija je prikazana u isječku 5.7.

Isječak 5.7: Ulazni niz koji je prouzročio grešku

```
1 $ cat id:000000,sig:09,src:000000,time:762,execs:57,op:havoc,rep:4
2 22
3 ++++++++++++++++++++++++++
```

Greška koja je pronađena kao prvi znak ima broj 2 što je razlog da se dogodi pogreška unutar programa koji se pokreće u emulaciji. Pogreška je adekvatno dojavljena premosniku i AFL++ alatu.

Nakon nekog vremena izvođenje u emulaciji se izrazito uspori i pretpostavka je da je radi malih računalnih resursa. Također je problem što se izvodi *bind shell* kod u Pythonu koji bi se mogao dodatno optimizirati ili prepisati u drugi programski jezik koji je prikladniji sustavima s malo resursa.

6. Zaključak

QEMU je alat koji omogućuje emuliranje raznih kompleksnih sustava. Alat AFL++ je alat koji služi za neizrazito testiranje pomoću praćenja izvršavanja koda. Trenutačno ne postoji podrška od strane AFL++ alata za testiranje sistemske emulacije, ali podržava testiranje korisničke emulacije.

U ovom radu su objašnjene osnove načina na koji alat AFL++ testira korisničku QEMU emulaciju. Iz informacija dobivenih iz izvornog koda qemuafl inačice QEMU-a, koriste se osnovni dijelovi koji su potrebne za praćenje izvršavanja koda, te se dodaju u izvorni kod QEMU alata. Izmjenama u QEMU alatu se implementiraju potrebni mehanizmi za testiranje sistemske emulacije. Jednostavnost implementacije praćenja koda u QEMU alatu omogućuje da se izmjene mogu prenijeti u druge QEMU inačice poput Xilinx [30] koje omogućavaju emulaciju još više različitih sustava. Napravljen je premosnik između QEMU i AFL++ alata koji služi kao veza za testiranje programa koji se izvršavaju u sistemskoj emulaciji.

Implementacijom tih mehanizama pokrenuto je testiranje na proizvoljnoj sistemskoj emulaciji. Objašnjeno je na koji način pokrenuti sistemsku emulaciju te koje je parametre nužno navesti navesti tijekom pokretanja kako bi se moglo testirati. Za premosnik je objašnjen način funkcioniranja te je demonstrirana lakoća testiranja na drugim sustavima i kako ga prilagoditi za neki drugi slučaj testiranja.

Sljedeći korak u razvoju ovog rada bi bilo naći učinkovit način dojavljivanja grešaka koje se događaju u sistemskoj emulaciji. Postoje alati koji promatraju memoriju unutar QEMU-a poput QASan alata, ali on također ne podržava sistemsku emulaciju. Trenutačni način prenošenja informacija između QEMU-a i premosnika i zatim premosnika i AFL++ alata je skup jer se prenosi mrežom, tu bi se mogle postići dodatne performanse ukoliko bi se implementirali načini prenošenja informacije drugim medijem poput dijeljenog memorijskog prostora. Testiranje bi se moglo ubrzati kada bi se napravio mehanizam *snapshota* QEMU slike kada se pokrene snimanje te kada bi se nakon svakog testa vraćalo u početno stanje. Time bi se dodatno smanjio šum u podacima, ali bi se isto tako omogućilo paralelno testiranje na više emulacija.

7. Literatura

[1] QEMU Project Developers. *QEMU*, Posjećeno 9.6.2022. URL https://www.qemu.org/.

[2] Michal Zalewski. american fuzzy lop. https://github.com/google/AFL, Pristupljeno 30.5.2022.

[3] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. American fuzzy lop plus plus (afl++), Posjećeno 10.6.2022. URL https://github.com/AFLplusplus/AFLplusplus.

[4] Fabrice Bellard. *QEMU, a fast and portable dynamic translator.* 2005.

[5] QEMU Project Developers. *QEMU User space emulator*, Posjećeno 2.6.2022. URL https://www.qemu.org/docs/master/user/main.html.

[6] Debian Zajednica. *QEMU*, Posjećeno 13.6.2022. URL https://wiki.debian.org/QEMU.

[7] QEMU Project Developers. *System Emulation*, Posjećeno 13.6.2022. URL https://www.qemu.org/docs/master/system/index.html.

[8] QEMU Project Developers. *Translator Internals*, Posjećeno 30.5.2022.. URL https://gitlab.com/qemu-project/qemu/-/blob/master/docs/devel/tcg.rst.

[9] Keith D Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011.

[10] QEMU Project Developers. *TCG README*, Posjećeno 30.5.2022.. URL https://gitlab.com/qemu-project/qemu/-/blob/master/tcg/README.

[11] GCC Documentation. *Function Entry and Exit*, Posjećeno 30.5.2022. URL https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gccint/Function-Entry.html.

[12] IBM. *Address space layout randomization*, Posjećeno 30.5.2022. URL https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization.

[13] Patrice Godefroid. Fuzzing: Hack, art, and science. *Communications of the ACM*, 63 (2):70–76, 2020.

[14] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[15] Michal Zalewsk i drugi. *Technical "whitepaper" for afl-fuzz*, Posjećeno 3.6.2022. URL https://lcamtuf.coredump.cx/afl/technical_details.txt.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[17] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, pages 23–30, 2020. doi: 10.1109/SecDev45635.2020.00019.

[18] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and s1341. *The LibAFL Fuzzing Library*, Posjećeno 6.6.2022. URL https://aflplus.plus/libafl-book/introduction.html.

[19] Ivan Ivanković. qemuafl github repozitorij, . URL https://github.com/ivanivankovic/qemu_ivankovic.

[20] Ivan Ivanković. afl-proxy, . URL https://github.com/ivanivankovic/afl-proxy.

[21] Jordan Whitehead. *QEMU and U: Whole-system tracing with QEMU customization*, 15.4.2021. URL https://www.atredis.com/blog/qemu-and-u-whole-system-tracing-with-qemu-customization.

[22] Nisu imenovani ali je repozitorij od AFLplusplus projekta. qemuafl github repozitorij, Posjećeno 30.5.2022. URL https://github.com/AFLplusplus/qemuafl.

[23] airbus seclab. *QEMU Blog*, Posjećeno 30.5.2022. URL https://github.com/airbus-seclab/qemu_blog.

[24] Andrea Fioraldi. *Improving AFL's QEMU mode performance*, 21.9.2018. URL https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[25] Brojni autori. *xxHash - Extremely fast hash algorithm*, Posjećeno 4.6.2022. URL https://github.com/Cyan4973/xxHash.

[26] Palmer Cluff. *Raspberry Pi running Raspbian Wheezyg*, Posjećeno 9.6.2022. URL https://github.com/palmercluff/qemu-images/tree/master/raspbian-wheezy.

[27] Fabrice Bellard. *QEMU User Documentation*, Posjećeno 9.6.2022. URL https://www.mankier.com/1/qemu.

[28] Anmol Shah. *Reverse Shell vs Bind Shell*, Posjećeno 13.6.2022. URL https://infosecwriteups.com/reverse-shell-vs-bind-shell-d5a1e80b6a6c.

[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. *Environment variables*, Posjećeno 9.6.2022. URL https://aflplus.plus/docs/env_variables/.

[30] Xilinx. *Xilinx's fork of Quick EMUlator (QEMU) with improved support and modelling for the Xilinx platforms.*, Posjećeno 13.6.2022. URL https://github.com/Xilinx/qemu.

Automatizirano testiranje uz pomoć alata AFL

Sažetak

Rastom složenosti se otvara sve više vektora napada na programsku podršku i sustave. Kako bi se uklonili vektori napada potrebno je napraviti prikladna testiranja i ukloniti pronađene sigurnosne propuste. Brojni sustavi nisu jednostavni za testiranje zbog njihove kompleksnosti i u tom slučaju kao koristan alat se može koristiti emulacija sustava. QEMU je alat koji omogućuje emulaciju cijelog sustava.

Načini i alati pomoću kojih se može provesti testiranje su brojni, ali alat prikladan za testiranje emuliranog sustava je AFL++ alat. Zbog nedostatka podrške testiranja cijelog sustava, u ovom radu je obrađeno način na koji se trebaju prilagoditi oba alata kako bi mogli funkcionirati zajedno.

U radu se prolaze izmjene u izvornom kodu QEMU-a kako bi se moglo pratiti izvođenje koda unutar emulacije. Razvija se premosnik izmedu QEMU i AFL++ alata kako bi se omogućila komunikacija među njima.

Ključne riječi: AFL, AFL++, QEMU, TCG, QEMU sistemska emulacija

Automated testing using the AFL tool

Abstract

With the increasing complexity of computer systems, the attack surface is also increasing. To reduce the attack surface, it's necessary to do appropriate testing and remove newly found vulnerabilities. A lot of computer systems are not easy to test because of their complexity and size. In those cases, emulation can be a great tool for making testing of those complex systems easier. QEMU is a tool that can emulate the whole system.

There are many different tools and approaches to testing, but a tool like AFL++ stands out as one of the more popular tools today. Because AFL++ currently doesn't support testing of system emulation, in this thesis it is explained how to modify both tools to achieve testing of system emulation.

This thesis explains how to modify source code of QEMU tool to gather code coverage of emulation. A proxy tool was also developed to orchestrate communication between AFL++ and QEMU tools.

Keywords: AFL, AFL++, QEMU, TCG, QEMU system emulation