MATEMATIČKI FAKULTET
UNIVERZITET U BEOGRADU

Milena Vujošević–Janičić

Automatsko otkrivanje prekoračenja bafera u programskom jeziku C

Magistarski rad

Mentor: prof. dr Dušan Tošić

Beograd
2008.

Vanji,
koji me je oduvek učio
tome kako je nauka lepa

Predgovor

Prekoračenje bafera jedan je od najkritičnijih i najčešćih propusta koji se javljaju u programima napisanim na programskom jeziku C. Ovi propusti imaju za posledicu dalje neispravno ponašanje programa i predstavljaju najpogodniju metu za bezbednosne napade. Ovaj problem je postao veoma aktuelan tokom prethodnih desetak godina. Od 2000-te godine nastali su prvi specijalizovani alati koji analizom koda pokušavaju da otkriju potencijalna prekoračenja bafera. Mnogi od njih uspešno detektuju brojna prekoračenja bafera, ali problem nije konačno rešen. Naime, neki od ovih alata rade presporo, neki imaju previše uzan domen, neki daju previše lažnih upozorenja, itd. Prema tome, još uvek postoji velika potreba za novim idejama, metodama i alatama koji bi se odnosili na ovaj problem.

U okviru rada na ovoj tezi, pokušala sam da razvijem novi metod, kao i prateći alat za otkrivanje prekoračenja bafera. Metod koji je razvijan koristi neke od postojećih ideja i uvodi razne novine koje omogućavaju pronalaženje većeg broja grešaka u kodu u odnosu na neke metode i manji broj lažnih upozorenja u odnosu na neke druge metode. Prototip implementacija ovog metoda, alat FADO, koji sam takođe razvila u okviru rada na ovoj tezi, pokazuje bolje performanse od nekoliko postojećih alata. Naravno, bilo bi dobro da se ovaj metod i alat dalje unapređuju i planiram da se time bavim u daljem radu. Tekuća verzija metoda i alata izložena je u radu [76] koji je prihvaćen za predstavljanje na značajnoj međunarodnoj konferenciji koja se bavi programskim jezicima — ICSOFT koja se ove godine, u julu, održava u Portu, u Portugaliji. Pored toga, delovi ove teze sadržani su u radovima [75, 77].

Magistarska teza sastoji se od sedam glava. U uvodnoj glavi su opisani problem prekarčenja bafera, njegovi najčešći uzroci i posledce. U drugoj glavi dat je pregled osnovnih pojmova i metoda koji su neophodni za razumevanje daljeg teksta. U trećoj glavi prikazani su mehanizami napada zloupotrebom prekoračenja bafera. U četvrtoj glavi dat je detaljan pregled postojećih tehnika za otkrivanje prekoračenja bafera. U naredne dve glave, koje su ujedno i centralni deo rada, opisani su razvijeni metod i njena implementacija. Poslednja glava sumira postojeći rad i ukazuje na moguće pravce u daljem radu.

U radu na ovoj tezi, izuzetnu pomoć pružio mi je mentor, profesor Dušan Tošić i na tome mu se najtoplije zahvaljujem. Profesor Tošić me je ohrabrivao razmatrajući ideje koje sam imala, ukazujući mi na moguće probleme ili dajući nove predloge. Više puta je detaljno čitao različite verzije magistarske teze i svo-

jim primedbama značajno doprineo njenom kvalitetu. Pored dragocene stručne pomoći, profesoru Tošiću sam neizmerno zahvalna i na razumevanju, podršci i poverenju koje mi je ukazao u toku rada na tezi. Zahvalna sam i članovima komisije za pregled i ocenu rada — profesoru Miodragu Živkoviću i profesoru Dušanu Starčeviću na podršci i korisnim sugestijama, zahvaljujući kojima je ova teza bolja nego u prethodnim verzijama. Veliku pomoć pružio mi je kolega Filip Marić, čiji dokazivač ArgoLib koristim u okviru alata FADO — Filip mi je u puno navrata pomagao u tome da na što bolji način koristim njegov dokazivač, implementirao dodatke dokazivača koji su važni za rad alata FADO, ali i davao veoma korisne sugestije za samu analizu programa. Zahvalna sam Jergu Šenu (Jörg Schön) iz Nemačke, autoru parsera JSCPP za jezik C koji se koristi u okviru alata FADO — Jerg mi je pružio veliku pomoć u razumevanju tehničkih detalja vezanih za njegov parser, kao i pravljenjem varijanti koje su meni bile potrebne. Zahvaljujem Dejvidu Vagneru (David Wagner) iz Sjedinjenih Američkih Država, autoru jednog od prvih alata za otkrivanje prekoračenja bafera, na korisnim sugestijama i referencama. Zahvaljujem i Kendri Kratkievič (Kendra Kratkiewitcz) iz Sjedinjenih Američkih Država i Džonu Vilanderu (John Wilander) iz Švedske na korisnim informacijama i ustupanju test primera koje sam koristila. Zahvalna sam svojim kolegama, Jeleni, Angelini, Sani, Vesni i Mladenu, saradnicima u nastavi Matematičkog fakulteta, na podršci i vremenu koje smo proveli razmenjujući iskustva iz zadataka na kojima smo radili. Najveću zahvalnost dugujem svojim roditeljima, Sonji i Mirku, kao i bratu Dušanu, na bezgraničnoj podršci, razumevanju i vremenu, posebno na brojnim vikendima i odmorima koje su utrošili da bih mogla da radim na ovoj tezi. Zahvaljujem se i svojoj svekrvi Stani što mi je često omogućavala da se posvetim radu na tezi ali i da se odmorim od tog rada. Svom dedi Vanji dugujem veliko hvala na tome što me je od mojih prvih godina učio tome kako je nauka lepa i kako treba uživati u njoj. Zahvalna sam ćerki Ružici i suprugu Peđi na podršci i razumevanju za vreme koje sam provodila radeći a ne sa njima. Pored toga, Peđi sam zahvalna i na vremenu koje je proveo slušajući moje ideje i dajući korisna zapažanja. Zahvalnost dugujem i svojim studentima čija su me pitanja u značajnoj meri motivisala da se bavim ovom problematikom, kao i svim svojim prijateljima koji su me razumeli kada sam, umesto da se vidim sa njima, morala da se posvetim svom radu. Svima koje sam nabrojala, zahvalna sam što su mi, na jedan ili drugi način, pomogli da privedem kraju rad na ovoj tezi. Svi oni su mi pomogli da teza bude bolja nego što bi bila inače.

Milena Vujošević-Janičić

Beograd, 12.06.2008.

Sadržaj

1

Uvod

Prihvatnik ili *bafer* (eng. buffer) je blok memorije rezervisan za privremeno skladištenje podataka.[1] U programskom jeziku `C`, baferom se smatra i niz i dinamički rezervisan blok memorije pridružen pokazivaču. *Prekoračenje* ili *prepunjenje bafera* (eng. buffer overflow ili buffer overrun) je upisivanje sadržaja van granica datog bafera, to jest, to je situacija u kojoj se ukazuje na adresu nekog raspoloživog bafera ali se koriste lokacije koje ne predstavljaju deo odgovarajućeg rezervisanog prostora (što je moguće u jezicima kao što je `C`). U principu, prekoračenje bafera je moguće samo ako postoji propust u programu. Postoji veliki broj načina zloupotrebe programa u kojima može doći do prekoračenja bafera [74]. Pored toga, prekoračenje bafera može da dovede i do neočekivanog toka izvršavanja programa. Ovaj problem je posebno rasprostranjen u programima napisanim na programskom jeziku `C`.

1.1 Najčešći uzroci prekoračenja bafera

Osnovni koncept programskog jezika `C` je da se dâ što veća sloboda programeru, uključujući i direktan pristup memoriji, čime se omogućava pisanje veoma efikasnih programa. Međutim, to otežava pisanje bezbednog koda jer sva odgovornost za bezbednost aplikacije leži isključivo na programeru. Naime, ne postoje automatske provere granica rezervisane memorije dodeljene nizovima ili pridružene pokazivačima. Dodatno, mnoge funkcije za rad sa niskama koje su podržane standardnom `C` bibliotekom su nebezbedne (kao što su, na primer, funkcije `strcpy`, `strcat`, `sprintf` i `gets`).

Do prekoračenja bafera obično dolazi kada programer prilikom korišćenja bafera zaboravi da proveri njegove granice ili uradi pogrešnu proveru granica.

[1] Pod *prihvatnikom* ili *baferom* se često misli na uži pojam — na blok memorije rezervisan za privremeni smeštaj podataka dok čekaju na prenos između područja podataka aplikacije i ulazno/izlaznog uređaja (pošto su ulazno/izlazni uređaji obično spori u poređenju sa centralnim procesorom, nije praktično pristupati im samo radi jednog ili dva bajta podataka). Termin *prihvatnik* je veoma dobar prevod za originalni termin, ali kako još nije zaživeo u našem jeziku, i u nastavku ovog teksta uglavnom će se koristiti opšte prihvaćen engleski termin *bafer*.

Provere su posebno važne prilikom korišćenja funkcija standardne biblioteke za rad sa niskama. Programeri često (naročito početnici) podrazumevaju da su pozivi ovih funkcija bezbedni ili rade pogrešne provere. Čak i iskusni programeri često koriste nebezbedne funkcije, nekada sa nekim proverama ili samo oslanjajući se na analizu koda koji se razvija. Mnoge (ili skoro sve) aplikacije napisane na programskom jeziku C sadrže nebezbedne pozive funkcija za rad sa niskama [79].

Razmotrimo sledeći kôd:

```
char src[200];
char dst[100];
fgets(src,200,stdin);
strcpy(dst,src);
```

Poziv funkcije `fgets` u nevedenom kodu je bezbedan jer ograničava unos sa standardnog ulaza na najviše `200` karaktera, a toliko prostora je i rezervisano za bafer `src`. Međutim, poziv funkcije `strcpy` nije bezbedan i može da dovede do prekoračenja bafera `dst` ukoliko je dužina niske koja se čuva u baferu `src` veća od `100`.

Neke funkcije standardne biblioteke se mogu koristiti bezbedno ukoliko se pre njihovih poziva sprovedu odgovajuće provere, dok su neke uvek nebezbedne. Na primer, poziv funkcije `strcpy(dst,src)` je bezbedan ako se prethodno proveri da je za `dst` rezervisano dovoljno prostora. Međutim, programeri obično ne urade ovakvu proveru ili je urade pogrešno, najčešće zaboravljajući da u proveru uključe i bajt neophodan za znak za kraj niske u `C`-u. Na primer, provera

```
if (sizeof(dst) < strlen(src))
    ...
else
    strcpy(dst, src);
```

je pogrešna jer omogućava prekoračenje bafera za jedan bajt i upisivanje znaka za kraj niske u prekoračeni bajt (pravilna provera bi bila `if (sizeof(dst) <= strlen(src))`). Zloupotreba ovakvih grešaka i prekoračenja je registrovana i opisana u literaturi [63, 44]. U nešto bezbednije funkcije spadaju verzije funkcija za rad sa niskama koje imaju ograničenja (kao što je funkcija `strncpy`), ali i ove funkcije imaju svoje nedostatke [79]. Na primer, funkcija `strncpy` može da ostavi bafer u koji prepisuje sadržaj bez terminirajuće nule, dok funkcije `strncat` i `snprintf` uvek dodaju terminirajuću nulu na kraj upisanog sadržaja. Ovakve nekonzistentnosti u efektima pomenutih funkcija dovode do grešaka u njihovom korišćenju.

Za razliku od prethodno pomenutih funkcija, funkciju `gets` nije moguće koristiti bezbedno. Ova funkcija učitava sadržaj koji se unosi sa standardnog ulaza sve do pojave znaka za novi red i smešta ga u bafer na koji ukazuje njen jedini argument. Broj znakova koji se unose sa standardnog ulaza, pre pojave znaka za novi red, može da bude proizvoljan, dok je veličina datog bafera fiksirana pre poziva funkcije. Iako je očigledno da je upotreba ove funkcije uvek

nebezbedna, ona se i dalje široko koristi, kao i druge nebezbedne funkcije (što se obično opravdava potrebom za efikasnim programima).

1.2 Posledice prekoračenja bafera

Upisivanje van granica bloka rezervisane memorije može da izazove razna nepoželjna ponašanja programa kao što su korišćenje pogrešnih podataka, neočekivan krah programa ili omogućavanje izvršavanja zlonamernog koda. Prekoračenje bafera je čest uzrok bezbednosne slabosti (ranjivosti) softvera[2] (eng. software vulnerability). Prema nekim procenama, prekoračenja bafera predstavljaju 50% uzroka slabosti programa, a ovaj procenat se dalje povećava vremenom [79]. Dakle, prekoračenje bafera je najrasprostranjenija forma bezbednosne slabosti softvera.

Zlonamerni programeri i korisnici programa, u daljem tekstu *napadači*, su uspeli da otkriju i zloupotrebe prekoračenja bafera u velikom broju programa [74]. Neke od najpoznatijih zloupotreba prekoračenja bafera su:

- *Morris worm* — crv koji se 1988. samostalno proširio po mreži računara ARPANET koristeći prekoračenje bafera u programima *Unix sendmail* i *Finger*.
- *Code Red worm* — crv koji je 2001. godine iskoristio prekoračenje bafera u Majkrosoftovom proizvodu *Internet Information Services 5.0*.
- *SQL Slammer worm* — crv koji je 2003. godine napao računare koji su koristili Majkorsoftov *SQL Server 2000*.

Prekoračenje bafera može da se desi u različitim delovima memorije pridružene programu.Najčešća i najlakša eksploatacija prekoračenja bafera odnosi se na prekoračenja koja su na steku. Detaljniji opis mehanizama bezbednosnih napada zasnovanih na prekoračenju bafera dat je u poglavlju 3.

1.3 Otkrivanje prekoračenja bafera

Za pronalaženje kritičnih delova programa dugo je korišćen program `grep` [72]. `Grep` je program opštije namene koji omogućava pretragu tekstualnih datoteka i pronalaženje onih linija teksta koje sadrže instance zadatog regularnog izraza. Kako je `grep` standardni deo operativnog sistema Unix, a takođe je i veoma jednostavan za upotrebu, on je godinama upotrebljavan kao jedina pomoć ručnom pregledanju koda. Međutim, kako ovaj pristup ne daje zadovoljavajuće rezultate (`grep` je detaljnije opisan u poglavlju 4.1.1), u poslednjih petnaestak godina razvijene su brojne naprednije tehnike za rešavanje problema automatskog

[2]Bezbednosna slabost softvera u ovom kontekstu odnosi se na svojstvo sistema koje dozvoljava napadaču da naruši pristup kontroli sistema, mehanizme nadgledanja sistema i sistemskih podataka, pouzdanost, integritet, dostupnost ili konzistentnost sistema.

pronalaženja mogućeg prekoračenja bafera. Ove tehnike dele se na statičke i dinamičke.

Statičke tehnike analiziraju izvorni kôd i imaju za cilj otkrivanje mogućih prekoračenja bafera pre nego što se program pusti u rad. Ove metode su najčešće neprecizne i nepotpune jer koriste različite aproksimacije i heuristike za određivanje mesta na kojima do prekoračenja može da dođe. To znači da pomoću njih neće moći da se uoče sve greške, kao i da će biti prjavljena i ona mesta na kojima do prekoračenja zapravo ne može da dođe. Statičke tehnike dele se na tehnike zasnovane na leksičkoj analizi i tehnike zasnovane na semantičkoj analizi. Tehnike zasnovane na leksičkoj analizi vrše jednostavnu analizu koda zasnovanu na poređenju komandi programa sa bazom podataka koja sadrži obrazce rizičnih upotreba određenih funkcija, veoma su efikasne, ali imaju visok nivo lažnih upozorenja. Tehnike zasnovane na semantičkoj analizi vrše dublju analizu koda i veoma se razlikuju po metodologijama i pristupu problemu prekoračenja bafera. Više o statičkim tehnikama i pomenutim pristupima biće rečeno u poglavlju 4.1.

Pomoću dinamičkih tehnika analizira se program u fazi njegovog izvršavanja. Neki alati zasnovani na dinamičkoj analizi programa mogu da se koriste prilikom razvoja programa i njegovog testiranja, dok drugi imaju za cilj sprečavanje prekoračenja bafera za vreme izvršavanja programa (što značajno utiče na njegove performanse). Sistemi bazirani na dinamičkoj analizi, koji se koriste u fazi razvoja i testiranja programa, mogu da budu veoma uspešni u otkrivanju onih prekoračenja koja se dese za vreme probnog izvršavanja programa. Međutim, to i dalje ne znači da u drugim granama programa ne postoje greške i da do prekoračenja ne može da dođe u nekim drugim situacijama. Tehnike, koje vrše prevencuju prekoračenja bafera, obično mogu da spreče samo neke vrste napada, najčešće samo one koje imaju za cilj prepisivanje adrese povratka funkcije. Ove tehnike dele se na tehnike zasnovane na kompajleru, tehnike zasnovane na korišćenju biblioteka funkcija i tehnike zasnovane na operativnom sistemu. Više o dinamičkim tehnikama biće rečeno u poglavlju 4.2.

U daljem tekstu, pod *otkrivanjem prekoračenja bafera* (eng. buffer overflow detection) misliće se i na otkrivanje prekoračenja bafera do kojeg je zaista došlo (u fazi izvršavanja programa) i na otkrivanje *mogućeg* prekoračenja bafera (analizom koda, pre izvršavanja programa).

2

Osnovni pojmovi i metode

U ovoj glavi ukratko su opisani pojmovi i tehnike čije je poznavanje neophodno za razumevanje ostatka teze. Opisana je organizacija memorije koja se dodeljuje programima, zajedno sa karakteristikama neophodnim za razumevanje prekoračenja bafera. Pored toga, uvedeni su pojmovi relevantni za verifikaciju (proveravanje ispravnosti) programa, uključujući, na primer, jednosmerno uparivanje i Horove trojke, kao i pojmovi i tehnike relevantni za automatsko dokazivanje teorema.

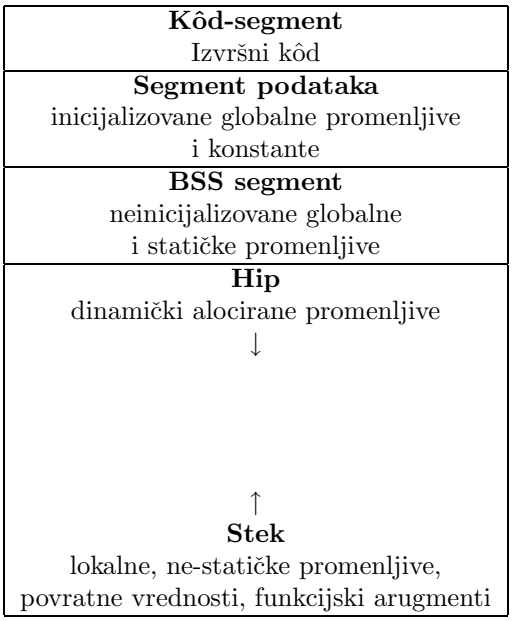
2.1 Organizacija memorije dodeljene programu

U fazi izvršavanja, program se od strane operativnog sistema tretira kao jedan od tekućih procesa. U okviru memorije, koju operativni sistem dodeljuje pojedinačnom procesu, postoji pet razdvojenih oblasti u kojima se čuvaju odgovarajući specifični podaci (ovo je opšti, delimično pojednostavljen model, zajednički za većinu modernih operativnih sistema). Ove oblasti memorije se nazivaju segmenti i to su:

- kôd-segment ili kodni segment (eng. Code segment ili Text segment);
- segment podataka (eng. Data segment);
- BSS[1] segment (eng. BSS segment);
- hip ili slobodna memorija (eng. heap segment);
- programski stek (eng. stack segment).

Za svaki podatak iz programa kompajler određuje u kom segmentu će se čuvati. Pojednostavljena ilustracija osnovne organizacije memorije data je na slici 2.1. Primer koda koji ilustruje promenljive, čije se vrednosti čuvaju u

[1] BSS je engleska skraćenica za "Block Started by Symbol". Ovaj deo memorije često se ne razdvaja od segmenta podataka.



Slika 2.1: Pojednostavljena ilustracija osnovne organizacije memorije

različitim segmentima memorije, dat je na slici 2.2. Segmenti memorije dodeljeni programu objašnjeni su detaljnije u nastavku teksta.

Kôd-segment

U okviru kôd-segmenta čuva se izvršna verzija programa tj. mašinski kôd programa, uključujući sistemske i korisnički definisane funkcije koje ga sačinjavaju. Kôd-segment se često naziva i tekstualni segment (eng. text segment). Operativni sistemi Linux i Unix obično uređuju memoriju tako da, ukoliko je to moguće, više instaci istog programa deli jedan ovaj segment, to jest, bez obzira na to koliko se procesa u jednom trenutku izvršava, u memoriji postoji samo jedan primerak istrukcija istog programa. Po ovom delu memorije nije moguće pisati u fazi izvršavanja programa. Ukoliko programer želi da modifikuje ili proizvede novi kôd u fazi izvršavanja programa, neophodno je da za to koristi programski stek ili hip (kao i da ovi segmenti takođe imaju mogućnost izvršavanja).

Segment podataka i BSS segment

Segment podataka sadrži sve inicijalizovane globalne podatke, kao što su globalne promenljive i konstante. Rezervisana veličina segmenta, njegova sadržina, kao i relativna pozicija u memoriji, određuju se u vreme prevođenja programa. BSS segment sadrži neinicijalizovane globalne i statičke podatke. Rezervisana veličina segmenta, kao i relativna pozicija u memoriji, određuju se, takođe u

```
static int global_constant = 1;     /* Inicijalizovana globalna promenljiva,
                                       cuva se u segmentu podataka */
static int global_variable;         /* Neinicijalizovana globalna promenljiva,
                                       cuva se u BSS segmentu */

void main(int argc, char *argv[])   /* Kopije argumenata argc i
                                       argv se cuvaju na steku */
{
int local_dynamic_variable;          /* Lokalna promenljiva,
                                        cuva se na steku */
static int local_static_variable;    /* Lokalna staticka promenljiva,
                                        cuva se u BSS segmentu */
int *buf_pointer = (int *)malloc(32);   /* Memorija na koju pokazuje
                                           buf_pointer, nalazi se na hipu,
                                           dok se buf_pointer cuva na
                                           steku kao lokalna promenljiva */
...
}
```

Slika 2.2: Promenljive čije se vrednosti čuvaju u različitim segmentima memorije

vreme prevođenja programa, dok se određivanje sadržine vrši u vreme izvršavanja programa. Svaki proces (pa i svaka instanca istog programa) sadrži sopstveni segment podataka i BSS segment.

Hip

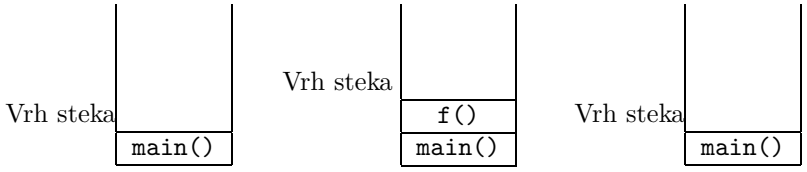
Hip, ili slobodna memorija, je prostor raspoložive memorije koju proces može da koristi za vreme izvršavanja. Alokacija memorije u fazi izvršavanja programa naziva se dinamička alokacija memorije. Da bi se omogućila dinamička alokacija memorije, u okviru hipa se čuva lista pokazivača koji pokazuju na slobodne (neupotrebljene) delove memorije i čiji se sadržaj menja tokom rada programa. Objekat koji je alociran u slobodnom memorijskom prostoru nije imenovan i njime se manipuliše uz pomoć njegove adrese. Objekat živi u memoriji sve do poziva funkcije koja oslobađa zauzetu memoriju ili do završetka rada programa. Često rezervisanje i oslobađanje memorije u toku procesa dovodi do fragmentacije hipa (i time sporijeg rada relevantnih funkcija).

Stek

Na stek segmentu se čuvaju lokalne (automatske) promenljive (lokalne promenljive su sve promenljive deklarisane u okviru bloka ili funkcije za koje se ne navede ključna reč static). Kod većine operativnih sistema, na programskom steku se čuvaju i kopije parametara funkcije, kao i informacije koje generiše kompajler — povratna vrednost funkcije, adresa povratka funkcije i osnovni pokazivač[2] (eng. base pointer) pozivaoca funkcije (na nekim platformama ove informacije se čuvaju u registrima). Svakoj funkciji se prilikom početka izvršavanja

[2]Osnovni pokazivač funkcije je adresa od koje počinju da se čuvaju lokalne promenljive funkcije.

dodeljuje jedan stek okvir (eng. stack frame) koji sadrži sve navedene podatke i oslobađa se kada se završi rad funkcije. Tada njenim lokalnim promenljive prestaje životni vek, a mesto u memoriji (stek okvir), koje je koristila funkcija, se oslobađa i može se koristiti za poziv neke druge funkcije. Programski stek karakteriše LIFO (eng. Last In First Out) struktura, što znači da se novi stek okvir alocira uvek na vrhu steka i da se stek okvir može osloboditi samo ako je na vrhu steka.



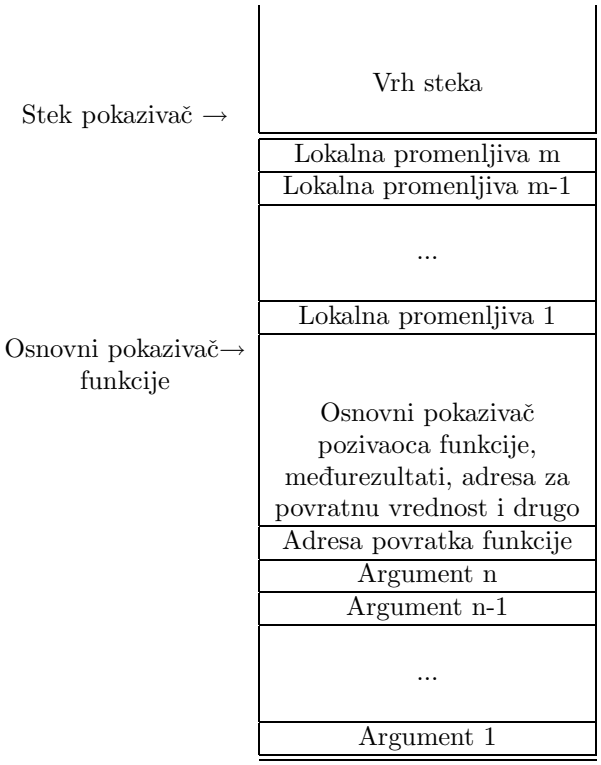
Slika 2.3: Ilustracija stanja programskog steka pre, tokom i nakon izvršavanja neke funkcije

Kao ilustraciju rada programskog steka, razmotrimo sledeći primer. Kada program počne izvršavanje funkcije `main()`, rezerviše se stek okvir za ovu funkciju (tj. prostor na steku za sve promenljive koje se deklarišu unutar ove funkcije i za druge relevantne informacije.) Ako se u okviru funkcije `main()` poziva funkcija `f()`, dodatni prostor na vrhu steka se rezerviše za potrebe ove funkcije, kao što je to prikazano na slici 2.3. Na steku se, kao što je ilustrovano na slici 2.4, čuvaju parametri koje prosleđuje funkcija `main()` funkciji `f`, povratna adresa funkcije, osnovni pokazivač pozivaoca funkcije `f()` i lokalne promenljive funkcije `f()`. Takođe, stek okvir funkcije čuva pokazivač steka (eng. stack pointer) koji pamti trenutnu vrednost vrha steka. Ako bi funkcija `f()` pozivala neku funkciju, prostor za tu funkciju bi bio alociran na poziciji novog vrha steka. Kada `f()` završi sa radom, njen stek okvir se oslobađa i vrh steka se vraća na poziciju na kojoj je bio pre poziva funkcije `f()`. Novi poziv neke funkcije onda koristi isti memorijski prostor koji je koristila funkcija `f()`.

2.2 Verifikacija programa i Horove trojke

Centralno pitanje u razvoju programa je ispitivanje njegove ispravnosti (korektnosti). Postupak dokazivanja da je program ispravan naziva se verifikacija. Formalna verifikacija programa se zasniva na matematičkim dokazima koji se obično izvode u terminima logike prvog reda. U razvijanju tehnika verifikacije programa, potrebno je najpre precizno formulisati pojam ispravnosti programa. Ispravnost programa počiva na pojmu specifikacije. Specifikacija je, neformalno, opis ponašanja programa koji treba napisati. Formalno, specifikacija se može zadati u terminima Horovih trojki [38].

Horova trojka opisuje kako se stanje programa menja izvršavanjem određenog koda. Horova trojka ima oblik (ϕ, P, ψ) pri čemu su ϕ i ψ tvrđenja logike prvog reda, a P je fragment koda. Tvrđenje ϕ se naziva preduslov (eng. precondition)



Slika 2.4: Organizacija stek okvira za pojedinačnu funkciju

a ψ se naziva postuslov, posleuslov ili pauslov (eng. postcondition) i zajedno čine specifikaciju za P. Značenje Horove trojke (ϕ, P, ψ) je da ako važi uslov ϕ i izvrši se program P, tada će nakon izvršavanja koda P važiti uslov ψ. Horova logika je formalni sistem koji sadrži pravila izvođenja kojima se utvrđuje korektnost programa u skladu sa zadatom specifikacijom.

2.3 Izraz, supstitucija, jednosmerno uparivanje

Pojmovi izraza, supstitucije i jednosmernog uparivanja važni su za formalno formulisanje uslova ispravnosti programa i pojedinačnih komandi.

Definicija 2.1 (Izraz) *Za skup promenljivih V i skup funkcijskih simbola Σ:*

- *promenljiva v iz skupa V je izraz;*
- *funkcijski simbol c iz Σ arnosti 0 je izraz;*
- *ako su e_1, e_2, ... e_n izraz i f funkcijski simbol iz Σ arnosti n tada je $f(e_1, ..., e_n)$ izraz;*

- *izrazi se formiraju samo primenom prethodna tri pravila.*

Na primer, za skup promenljivih $V = \{x, y, z\}$ i skup funkcijskih simbola $\Sigma = \{(a, 0), (+, 2), (-, 1)\}$ izrazi su: x, y, a, $(a + y)$, $-(x)$, $(x + y)$, $-(x + a)$ (pri čemu je funkcijski simbol + zapisan infiksno).

Nad skupom svih izraza definiše se operacija zamene promenljive izrazom. Ovom operacijom dobija se novi izraz tako što se svako pojavljivanje određene promenljive zameni datim izrazom.

Definicija 2.2 (Zamena promenljive izrazom) *Za zamenu promenljive v izrazom e u izrazu E, u oznaci $E[v \to e]$ važi:*

- $v[v \to e] = e$*;*
- $u[v \to e] = u$ *za promenljivu u iz V razlicitu od v;*
- $c[v \to e] = c$ *za funkcijski simbol c iz Σ arnosti* 0*;*
- $f(e_1, ...e_n)[v \to e] = f(e_1[v \to e], ..., e_n[v \to e])$.

Na primer, zamenom promenljive x izrazom $a + y$ u izrazu E

- za $E = x$ dobija se izraz $x[x \to (a + y)] = a + y$;
- za $E = y$ dobija se izraz $y[x \to (a + y)] = y$;
- za $E = -(x)$ dobija se izraz $-(x)[x \to (a + y)] = -(a + y)$;
- za $E = -(x + a)$ dobija se izraz $-(x + a)[x \to (a + y)] = -((a + y) + a)$.

Kompozicija zamena promenljivih izrazom je složena zamena.

Definicija 2.3 (Složena zamena) *Za različite promenljive v_1, v_2, ..., v_n iz V složena zamena $E[v_1 \to e_1, v_2 \to e_2, ..., v_n \to e_n]$ je zamena $E[v_1 \to e_1]...[v_n \to e_n]$, pri čemu ni jedan od izraza e_i ne sadrži ni jednu od promenljivih v_j.*

Na primer, složenom zamenom $\sigma = [x \to -(a)][y \to (a + a)]$ primenjenom na izraz $(x + y)$ dobija se izraz $(x + y)\sigma = (-(a) + (a + a))$.

Koristeći operaciju složene zamene nad izrazima se definišu relacija unifikabilnosti i relacija jednosmernog uparivanja.

Definicija 2.4 (Unifikabilni izrazi) *Izrazi e_1 i e_2 su unifikabilni ako postoji zamena σ takva da važi $e_1\sigma = e_2\sigma$.*

Na primer, izrazi $(x + y)$ i $(z + a)$ su unifikabilni, pri čemu je odgovarajuća zamena $\sigma = [x \to z, y \to a]$, odnosno

$$(x + y)[x \to z, y \to a] = (x + y)[x \to z][y \to a] = (z + y)[y \to a] = (z + a)$$

$$(z + a)[x \to z, y \to a] = (z + a)[x \to z][y \to a] = (z + a)[y \to a] = (z + a)$$

odakle važi da je $(x + y)\sigma = (z + a)\sigma$.

Definicija 2.5 (Jednosmerno uparivanje) *Izraz e se jednosmerno uparuje sa izrazom E ako postoji zamena σ takva da važi $e = E\sigma$.*

Svojstvo jednosmernog uparivanja je slabije od svojstva unifikabilnosti. Izrazi e_1 i e_2 su unifikabilni ako se e_1 jednosmerno uparuje sa e_2 i e_2 se jednosmerno uparuje sa e_1. Na primer, izraz $((x+a)+a)$ se jednosmerno uparuje sa izrazom $(z+y)$, odgovarajuća zamena je $[z \to x+a, y \to a]V$. Izraz $(z+y)$ se ne uparuje jednosmerno sa izrazom $((x+a)+a)$.

2.4 Zadovoljivost u odnosu na teoriju i SMT rešavači

Zadovoljivost u odnosu na teoriju je problem odlučivanja da li je data formula zadovoljiva u odnosu na osnovnu teoriju T opisanu u klasičnoj logici prvog reda sa jednakošću. Rešavači za ovaj problem, SMT (eng. Satisfiability Modulo Theory) rešavači, imaju mnoge primene, posebno u verifikaciji softvera i hardvera. Neke od, u primenama, najkorišćenijih osnovnih teorija su teorija neinterpretiranih funkcija, linearna aritmetika i teorije koje opisuju programske strukture, kao što su nizovi i liste. Moderni SMT rešavači u velikoj meri koriste iskazno rezonovanje i tehnike uvedene modernim SAT rešavačima [66]. Većina trenutno vodećih SMT rešavača (zasnovanih na DPLL(T) arhitekturi [60]), ima podršku za linearnu aritmetiku i može da obradi izuzetno složena tvrđenja koja dolaze iz industrije.

SMT-lib inicijativa[3] ima za cilj stvaranje biblioteke SMT test primera i svih pratećih stadarda i notacijskih konvencija [65]. U SMT-lib standardu, osnovna logika je klasična logika prvog reda sa jednakošću.

2.5 Linearno programiranje i linearna aritmetika

Linerno programiranje (poznato i kao linearna optimizacija) je problem maksimiziranja ili minimiziranja linearne funkcije (koja se naziva kriterijumska ili ciljna funkcija) na skupu koji je određen linearnim ograničenjima i uslovima nenegativnosti. Na primer, problem linearnog programiranja može da bude maksimiziranje kriterijumske funkcije

$$f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$$

pri ograničenjima

$$A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0$$

pri čemu je

$$\mathbf{x} = (x_1, x_2, ..., x_n),$$

$$\mathbf{c} = (c_1, c_2, ...c_n) \in \mathbf{R}^n, \mathbf{b} = (b_1, b_2, ...b_n) \in \mathbf{R}^n,$$

[3] http://www.smt-lib.org/

$$A = [a_{ij}]_{i=1\ldots m}^{j=1\ldots n}, a_{ij} \in \mathbf{R}, n, m \in \mathbf{N}.$$

Linearna aritmetika, nad poljem relanih brojeva, je deo aritmetike koji uključuje sabiranje i množenje racionalnim konstantama. Formula bez kvantifikatora, koja pripada linearnoj aritmetici, je formula logike prvog reda čiji su atomičke formule oblika

$$a_1 x_1 + \ldots + a_n x_n \bowtie b,$$

gde su $a_1, \ldots, a_n$ i b realni brojevi, a $x_1, \ldots, x_n$ su realne, racionalne ili celobrojne promenljive, a $\bowtie$ je jedna od relacija $=, \leq, <, >, \geq, \neq$. Linearna aritmetika je odlučiva, tj. postoji algoritam, procedura odlučivanja, takva da za svaku ulaznu rečenicu linearne aritmetike F procedura vraća *tačno* ako i samo ako je F teorema linearne aritmetike, i vraća *netačno* u suprotnom. Takođe, postoji algoritam koji za svaki ulazni skup formula linearne aritmetike bez kvantifikatora, vraća *tačno* ako i samo ako je ulazni skup zadovoljiv (a *netačno* inače). Zadovoljivost skupova formula linearne aritmetike bez kvatnifikatora može da se razmatra kao varijanta problema linearnog programiranja. Na primer, skup linearnih ograničenja $x + y > 0$, $x + 2y < 3$ je zadovoljiv (na primer, zadovoljiv je za vrednosti $x = 1, y = 0.5$).

Linearno programiranje i linearna aritmetika se koriste u verifikaciji softvera i hardvera zato što mogu da modeluju mnoge procese i izračunavanja. Na primer, aritmetika pokazivača se pogodno modeluje linearnom aritmetikom. Dodatno, procedure odlučivanja za linearnu aritmetiku su mnogo efikasnije nego procedure odlučivanja za celokupnu aritmetiku nad realnim brojevima ili procedure odlučivanja za linearnu aritmetiku nad celim brojevima.

Najčešće korišćene metode odlučivanja za ispitivanje zadovoljivosti skupova formula linearne aritmetike su Furije-Mockinova procedura [50] i procedura koja se zasniva na simpleks metodu [25].

2.6 Metod simpleks

Metod simpleks je metod za rešavanje problema linearnog programiranja [17]. Mnogi ga smatraju jednim od najznačajnijih algoritama dvadesetog veka. Metod simpleks ima brojne primene, u različitim optimizacionim problemima, ali i u verifikaciji softvera i hardvera. Metod počiva na činjenici da sistem linearnih ograničenja definiše skup dopustivih rešenja, koji je konveksan poliedar (politop). Ekstremna vrednost kriterijumske funkcije nalazi se u jednoj od ekstremnih tačaka ovog skupa, tj. u temenu politopa. Rešenje koje odgovara temenu politopa naziva se dopustivo bazno rešenje. Metod je iterativan — najpre se rešava problem pronalaženja bilo kojeg dopustivog baznog rešenja. Ispitivanje optimalnosti tekućeg baznog rešenja vrši se tako što se poredi vrednost kriterijumske funkcije sa vrednostima kriterijumske funkcije u susednim baznim rešenjima (susednim temenima). Ako ne postoji susedno rešenje koje daje bolju vrednost kriterijumske funkcije, tekuće rešenje je optimalno, a u suprotnom se tekuće rešenje zamenjuje rešenjem koje daje bolju vrednost kriterijumske funkcije. Teorijski je dokazano da je simpleks algoritam eksponen-

cijalne složenosti [45]. Međutim, on se u praksi često koristi i u velikom broju slučajeva (tj. za posebne klase funkcija) konvergira u polinomijalnom vremenu [61, 31].

Pored osnovne verzija simpleks metoda za problem optimizacije, postoje i mnoge njegove varijante, uključujući varijantu odlučivanja koja ispituje da li je skup linearnih ograničenja zadovoljiv ili nije. Ova varijanta simpleks metoda (koja se koristi u implementaciji dokazivača kojeg koristi ovde predložen sistema za otkrivanje prekoračenja bafera), opisana je u radu [25].

3

Bezbednosni napadi zasnovani na prekoračenju bafera

Napad na bezbednost softvera je zlonamerna promena toka kontrole izvršavanja programa, najčešće sa ciljem ubacivanja i pokretanja koda napadača. Kôd koji napadač pokrene nasleđuje privilegije koje je imao napadnuti program i može da ih zloupotrebi. Ukoliko je napadnuti program imao najviše privilegije, napadač stiče mogućnost kompletnog upravljanja napadnutim sistemom. Ugrožavanje bezbednosti softvera je moguće samo ukoliko postoji propust u dizajnu i/ili implementaciji programa ili u sistemu u kojem se on izvršava. Propusti koji omogućavaju prekoračenje bafera su kritični za bezbednost softvera — zloupotreba prekoračenja bafera je jedan od najčešćih vidova napada na bezbednost programa [79].

Mehanizmi napada nisu relevantni za statičku analizu prekoračenja bafera ali su veoma važni za dinamičku analizu. Napadi i dinamičke tehnike se razvijaju uporedo: otkrivanje nove vrste napada utiče na unapređivanje postojećih dinamičkih tehnika (kako bi one omogućile prevenciju novootkrivene vrste napada) i obratno, razvijanje tehnika za dinamičku prevenciju i otkrivanje prekoračenja bafera, podstiče pronalaženje novih vrsta napada na bezbednost softvera.

U ovoj glavi biće opisani načini na koje napadač može da ubaci sopstveni kôd u program i pokrene ga, kao i različite tehnike napada na bezbednost programa zasnovane na prekoračenju bafera. Postoje tehnike koje su specifične za segment memorije u kojem se desi prekoračenje bafera, dok tehnike opšte prirode mogu da se realizuju u svim segmentima memorije.

3.1 Ubacivanje koda napadača

Ako napadač zloupotrebom prekoračenja bafera ima za cilj pokretanje sopstvenog koda, tada je potrebno da željeni kôd smesti na neko mesto u memoriji. Za smeštanje svog koda, napadač može da koristi isti bafer kojim realizuje prekoračenje i promenu toka kontrole programa, ali može da koristi i neki drugi bafer.

Ukoliko ne postoje nikakve provere granica bafera koji se prekoračuje, onda se nakon sadržaja, izmenjenog sa ciljem promene toka kontrole programa, može upisati i kompletan kôd koji napadač želi da izvrši (to, pored ostalog, ilustruje primer 3.2). U zavisnosti od veličine bafera i veličine koda koji napadač želi da izvrši, kôd je, u nekim situacijama, moguće smestiti i na početak samog bafera (to, pored ostalog, ilustruje primer 3.5). Ukoliko bafer nije dovoljno veliki da može da sadrži kôd koji napadač želi da izvrši i ukoliko postoje (pogrešne) provere granica bafera koje dozvoljavaju prekoračenje za svega nekoliko bajtova, tada je moguće napad izvesti uz pomoć dva bafera. U tom slučaju, jedan bafer omogućava prekoračenje koje obezbeđuje izmenu toka kontrole programa, a drugi bafer sadrži kôd koji treba da se izvrši. Ova dva bafera ne moraju da se nalaze u istim delovima memorije.

Napad često ima za cilj pokretanje komandnog prozora[1] [1]. Pokretanjem komandnog prozora napadač stiče mogućnost izvršavanja sistemskih naredbi sa privilegijama koje je imao napadnut program. Ovakav napad je veoma opasan, a relativno jednostavan za sprovođenje. Na primer, pod operativnim sistemom Linux za pokretanje `shell` programa, tj. za poziv sistemske funkcije `system()` sa argumentom `/bin/sh/`, dovoljno je 83 bajta, prikazanih na slici 3.1. Prema tome, ukoliko postoji mogućnost zloupotrebe prekoračenja bafera za koji je rezervisano bar 83 bajta, tada je kôd potreban za pokretanje komandnog prozora moguće postaviti unutar samog bafera.

```
char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0"
  "\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8"
  "\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Slika 3.1: Sadržaj bafera čije izvršavanje omogućava pokretanje `shell` programa pod operativnim sistemom Linux na procesoru serije x86

Obično nije jednostavno odrediti adresu na koju će se upisati kôd koji napadač želi da izvrši i na koju on želi da preusmeri tok izvršavanja programa. Zbog toga napadač najčešće popunjava oblast, pre koda koji želi da izvrši, asemblerskom naredbom `nop` (eng. no operation), naredba bez dejstva. Ova naredba samo povećava programski brojač i time omogućava stizanje do željenog koda. Na taj način, nije neophodno znati tačnu adresu na koju treba preusmer-

[1] Pod komandnim prozorom misli se na program `shell` na Unix zasnovanim sistemima i na program `command.com` ili `cmd.exe` na Windows operativnim sistemima.

iti tok izvršavanja programa već je dovoljno izvršiti preusmeravanje na neki deo oblasti popunjene naredbama bez dejstva, iza koje sledi napadački kôd.

3.2 Zloupotrebe prekoračenja bafera na programskom steku

Napadi na sigurnost programa specifični za zloupotrebu prekoračenja bafera na programskom steku su:

- izmena adrese povratka funkcije
- izmena vrednosti osnovnog pokazivača pozivaoca funkcije.

Postoji veliki broj dinamičkih tehnika koje sprečavaju izmene ovih vrednosti (te tehnike opisane su u poglavlju 4.2.2). Detaljan pregled napada zasnovanih na prekoračenju bafera na steku dat je u radovima [15, 73, 22, 83].

Izmena adrese povratka funkcije

Izmena adrese povratka funkcije je najčešći i najproučavaniji vid napada na bezbednost programa. U okviru ovog napada, napadač šalje podatke programu koji pokušava da ih sačuva u baferu neodgovarajuće, tj. manje veličine. Memorija rezervisana za bafer nalazi se na steku i napadač, koristeći prepunjenje bafera (tj. odsustvo odgovarajuće provere u programu), menja (ili omogućava izmenu) adrese povratka funkcije tako da ona ukazuje na mesto u memoriji na kojem je smešten kôd napadača. Kada se tekuća funkcija izvrši, program nastavlja izvršavanje od adrese koju je napadač upisao i time počinje izvršavanje koda napadača. Primer 3.1 ilustruje ovu vrstu napada. U klasičnom scenariju zloupotrebe prekoračenja bafera, kôd koji napadač želi da izvrši se takođe smešta u bafer kojim se realizuje prekoračenje[2]. Primer 3.2 ilustruje ovu vrstu napada.

```
#define BUF_SIZE 8
main(int argc, char* argv[])
{
char bafer[BUF_SIZE];   /*Na steku se alocira prostor za bafer */

strcpy(bafer, argv[1]); /*Kopira se prvi argument
                          komandne linije u bafer*/
}
```

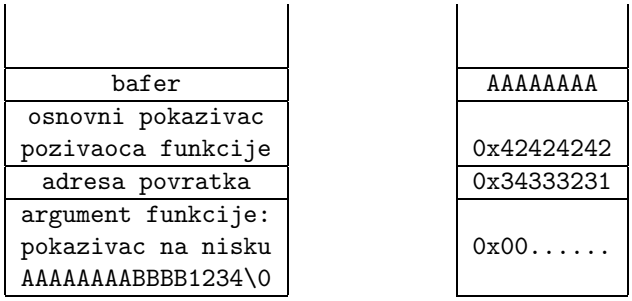
Slika 3.2: Primer programa koji omogućava zloupotrebu prekoračenja bafera na steku izmenom adrese povratka funkcije

[2]Da bi napad ove vrste bio moguć, neophodno je da sistem dozvoljava izvršavanje kôda koji se nalazi na steku.

Primer 3.1 *U programu koji je dat na slici 3.2, funkcija* `strcpy` *kopira sadržaj argumenta komandne linije programa u nisku* `bafer`*. Funkcija* `strcpy` *ne vrši proveru da li je za nisku u koju se kopira sadržaj rezervisan dovoljno veliki prostor i na taj način omogućava prekoračenje bafera. Neka se kôd, koji napadač želi da izvrši, nalazi na adresi* `0x34333231`*. Ako se program pokrene sa argumentom* `AAAAAAAABBBB1234`*, tada se*

- *rezervisana memorija za* `bafer` *popunjava slovima* `A`*,*
- *osnovni pokazivač pozivaoca funkcije se postavlja na* `0x42424242` *(što je ASCII vrednost za* `BBBB`*),*
- *adresa povratka funkcije se postavlja na* `0x34333231` *(što je ASCII vrednost za* `1234`*, u* little endian *obliku),*
- *prvi bajt lokalne kopije argumenta funkcije dobija vrednost* `'\0'`*.*

Kada funkcija završi sa radom, pokreće se kôd koji je napadač upisao na adresu `0x34333231`*. Prikaz stanja memorije dat je na slici 3.3.*

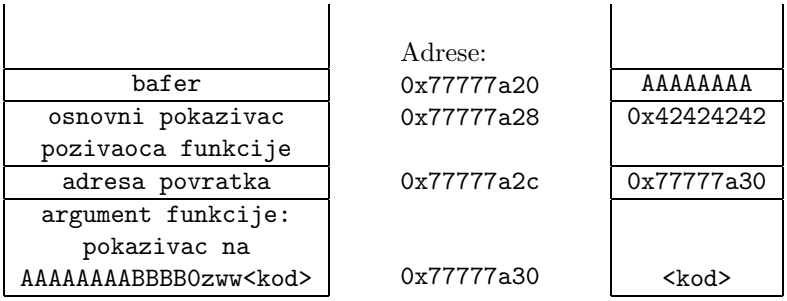


Slika 3.3: Stanje programskog steka za kôd iz primera 3.1, neposredno nakon pokretanja programa (levo) i posle poziva funkcije `strcpy()` (desno)

Primer 3.2 *Ukoliko se program koji je dat na slici 3.2 pokrene sa argumentom komandne linije* `AAAAAAAABBBB<adresa_argumenta><kod>` *tada se:*

- *rezervisana memorija za* `bafer` *popunjava slovima* `A`*,*
- *osnovni pokazivač pozivaoca funkcije se postavlja na* `0x42424242` *(što je ASCII vrednost za* `BBBB`*),*
- *adresa povratka funkcije dobija vrednost adrese argumenta funkcije,*
- *prostor, počevši od adrese na kojoj se nalazi argument funkcije, se prepisuje kodom napadača, koji će se izvršiti kada funkcija završi sa radom.*

Prikaz stanja memorije za program pokrenut sa argumentom `AAAAAAAABBBB0zww<kod>` *za koji se argument funkcije nalazi na adresi* `0x77777a30` *(što je ASCII vrednost za* `0zww`*), dat je na slici 3.4*



Slika 3.4: Stanje programskog steka za kôd iz primera 3.1, neposredno nakon pokretanja programa sa argumentom `AAAAAAAABBBB0zww<kod>` (levo) i posle poziva funkcije `strcpy()` (desno)

Menjanje celokupnog sadržaja koji se nalazi između bafera i adrese povratka funkcije, kao što je to bio slučaj u prethodnim primerima, predstavlja osnovu za otkrivanje prekoračenja bafera na kojoj se zasnivaju mnogi dinamički alati (na primer, alati StackGuard i StackShield, koji su opisani u poglavlju 4.2.2). Direktno menjanje samo adrese povratka funkcije, na primer uz pomoć pokazivača čiji je sadržaj izmenjen prekoračenjem bafera — kao što je to ilustrovano u primeru 3.3, onemogućava zaštitu koju nude ovi dinamički alati [6].

```
#define BUF_SIZE 64
main(int argc, char* argv[])
{
                        /*Na steku se alocira prostor za*/
long* p;                /*pokazivac, */
char bafer[BUF_SIZE];   /*bafer i */
int  i = 5;             /*celobrojnu promenljivu*/

p = &i;

strcpy(bafer, argv[1]);  /*Kopira se prvi argument
                           komandne linije u bafer*/
*p = atoi(argv[2]);      /*Skladisti se drugi argument
                           na adresi koju cuva pointer*/
}
```

Slika 3.5: Primer programa koji omogućava zloupotrebu prekoračenja bafera na steku i direktnu izmenu adrese povratka funkcije

Primer 3.3 *U programu koji je dat na slici 3.5, u okviru poziva funkcije* `strcpy` *može da dođe do prekoračenja niza* `bafer`*. Ovo prekoračenje menja sadržaj susedne lokalne promenljive — pokazivača* `p`*. Ukoliko se ova izmena izvrši tako*

da nova vrednost pokazivača `p` *ukazuje na adresu u memoriji na kojoj se čuva adresa povratka funkcije, tada se poslednjom naredbom programa direktno menja samo adresa povratka funkcije i ona dobija vrednost drugog argumenta komandne linije.*

Često nije jednostavano odrediti lokaciju adrese povratka funkcije. Zbog toga napadač najčešće upisuje novu adrese povratka funkcije na više uzastopnih lokacija u baferu, čime povećava verovatnoću da je neka od tih adresa postavljena na odgovarajuće mesto.

Izmena vrednosti osnovnog pokazivača

Zloupotreba prekoračenja bafera izmenom vrednosti osnovnog pokazivača pozivaoca funkcije je komplikovanija i manje česta od napada koji imaju za cilj izmenu adrese povratka funkcije.

Osnovna ideja ovog napada je upisivanje u memoriju lažnog stek okvira[3] čija adresa povratka ukazuje na kôd koji napadač želi da izvrši. Prekoračenjem bafera menja se vrednost osnovnog pokazivača pozivaoca funkcije tako da on ukazuje na podmetnuti stek okvir. Ovaj stek okvir se aktivira kada funkcija u kojoj je došlo do prekoračenja bafera završi sa radom. Nakon toga, lažni stek okvir, putem svoje adrese povratka, preusmerava izvršavanje programa na željeni kôd.

3.3 Zloupotrebe prekoračenja bafera na hipu

Zloupotreba prekoračenja specifična za bafere kod kojih je memorija rezervisana na hipu je značajno komplikovanija od ostalih vrsta napada. Ovi napadi postaju sve češći zbog velikog broja dinamičkih alata koji uspešno sprečavaju napade zasnovane na prekoračenju bafera na steku. Detaljan opis napada zasnovanih na zloupotrebi prekoračenja bafera koji se nalaze na hipu može se naći u radovima [9, 53, 57].

Da bi se realizovao napad prekoračenjem bafera na hipu, neophodno je detaljno poznavanje implementacije sistema za upravljanje memorijom na hipu. To je otežano činjenicom da svaka platforma ima svoj sistem za upravljenje memorijom na hipu, a da, pored toga, svaki program može da implementira svoj sopstveni sistem za upravljanje memorijom.

U opštem slučaju, upravljanje memorijom na hipu uključuje operacije rezervisanja, oslobađanja i spajanja oslobođenih blokova memorije. Zloupotreba prekoračenja bafera na hipu je moguća zato što se na hipu, neposredno uz prostor rezervisan od strane korisnika, čuvaju i informacije neophodne za realizaciju pomenutih operacija. Preciznije, svaki blok memorije sadrži i informacije o veličini bloka, veličini iskorišćenog dela bloka i pokazivače na sledeći i prethodni blok u memoriji. Ovi pokazivači se ažuriraju prilikom održavanja tekućeg stanja hipa (nakon operacija rezervisanja i oslobađanja memorije) i u

[3] Lažni stek okvir može se, na primer, upisati u pogodno izabrani bafer.

okviru tog ažuriranja uvek postoji bar jedna naredba obilika `p1=p2`, tj. naredba kojom jedan pokazivač dobija vrednost drugog pokazivača. Ova naredba može da se zloupotrebi ukoliko se vrednosti pokazivača izmene prekoračenjem nekog bafera na hipu. Na primer, moguće je izmeniti vrednosti ovih pokazivača tako da pokazivač `p1` ukazuje na mesto u memoriji na kojem se nalazi adresa povratka tekuće funkcije (tj. funkcije u kojoj se vrši ova dodela), a pokazivač `p2` ukazuje na mesto u memoriji na kojem se nalazi kôd koji napadač želi da izvrši. Tada ova dodela obezbeđuje da se, nakon izvršavanja funkcije, kontrola programa prepusti kôdu napadača.

3.4 Opšte zloupotrebe prekoračenja bafera

Prekoračenje bafera, nezavisno od segmenta memorije u kojem se nalazi bafer, može da se zloupotrebi izmenom neke vrednosti u memoriji koja je kritična za bezbednost, izmenom pokazivača na funkciju i izmenom bafera kada se koriste `setjmp/longjmp` funkcije.

Logički napadi

Logički napadi koriste prekoračenje bafera za izmenu neke vrednosti u memoriji koja je kritična za bezbednost, na primer identifikatora korisnika, imena datoteke ili pokazivača na datoteku. Logički napadi ne uključuju pokretanje koda napdača. Oni nisu česti, ali su veoma opasni. Najpoznatiji primer logičke zloupotrebe prekoračenja bafera je *Morris worm* (opisan u poglavlju 1.2). Primer 3.4 ilustruje logičke napade.

Primer 3.4 *U programu koji je dat na slici 3.6 deklarišu se dve statičke promenljive koje se čuvaju u BSS segmentu. Zatim se otvara datoteka* `file.txt` *za pisanje, nakon čega se učitava sadržaj sa standardnog ulaza i upisuje u* `bafer`*. Kako funkcija* `gets` *učitava sadržaj sa standardnog ulaza, bez obzira na veličinu samog ulaza i na veličinu rezervisane memorije za* `bafer`*, to je na ovom mestu moguće prekoračenje bafera. Ovim prekoračenjem može se izmeniti pokazivač* `pok_dat`*, čija je fizička lokacija na steku neposredno nakon fizičke lokacije niza* `bafer`*. Pokazivač se može izmenti tako da pokazuje na neku drugu datoteku, kritičnu za bezbednost sistema. Na primer, pokazivač može da se preusmeri da pokazuje na datoteku koja sadrži lozinke za pristup sistemu. U ovu datoteku će, pozivom funkcije* `fputs`*, biti upisan sadržaj niza* `bafer`*. Ova vrsta napada može se na sličan način realizovati i u ostalim segmentima memorije.*

Izmena pokazivača na funkciju

Ukoliko se prekoračenjem bafera izmeni vrednost nekog pokazivača na funkciju tako da nova vrednost ukazuje na kôd koji napadač želi da izvrši, tada će prilikom prvog narednog dereferenciranja pokazivača na funkciju biti izvršen kôd napadača. Primer 3.5 ilustruje ovakav napad.

```
#include <stdio.h>
#define BUF_SIZE 8

main(int argc, char* argv[])
{
                              /*U BSS segmentu skladiste se*/
static char  bafer[BUF_SIZE]; /*bafer i*/
static FILE* pok_dat;         /*pokazivac na datoteku*/

int i;

pok_dat = fopen("./file.txt", "w"); /*Otvara se datoteka file.txt*/
gets(bafer);                        /*Ucitava se sadrzaj sa
                                      standardnog ulaza i
                                      upisuje se u bafer*/
fputs(bafer, pok_dat);              /*Upisuje sa sadrzaj bafera
                                       u datoteku*/
}
```

Slika 3.6: Primer programa koji omogućava zloupotrebu prekoračenja bafera u BSS segmentu izmenom vrednosti pokazivača na datoteku

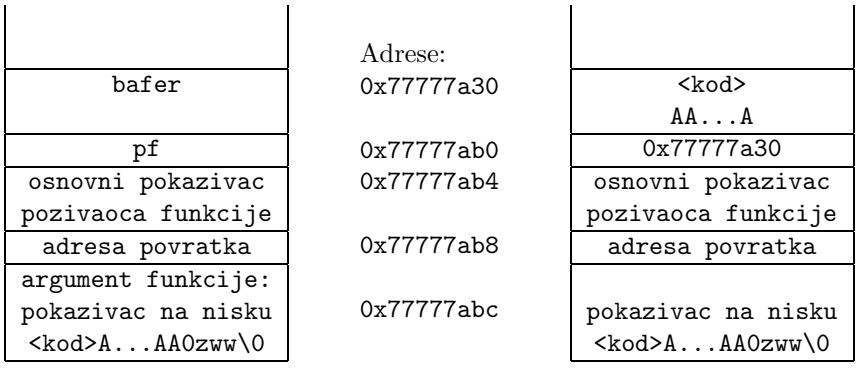
```
#define BUF_SIZE 128
int funkcija(char* string)
{
...
}
main(int argc, char* argv[])
{
                        /*Na steku se alocira prostor za*/
int (*pf)(char*);       /*pokazivac na funkciju */
char bafer[BUF_SIZE];   /*i za bafer */

pf = &funkcija;

strcpy(bafer, argv[1]);  /*Kopira se prvi argument
                           komandne linije u bafer*/
(int)(*pf)("Zdravo!");   /*Poziva se funkcija na koju ukazuje
                           pokazivac na funkciju pf*/
}
```

Slika 3.7: Primer programa koji omogućava zloupotrebu prekoračenja bafera na steku izmenom vrednosti pokazivača na funkciju

Primer 3.5 *U programu koji je dat na slici 3.7, deklarišu se pokazivač na funkciju i bafer veličine 128 bajtova. Nakon postavljanja pokazivača* `pf` *da ukazuje na funkciju* `funkcija`*, moguće je prekoračenjem bafera izmeniti sadržaj ovog pokazivača i postaviti ga da ukazuje na adresu na kojoj se nalazi kôd koji napadač želi da izvrši. U tom slučaju, pozivom funkcije kroz pokazivač* `pf` *menja se tok izvršavanja programa i pokreće se kôd koji napadač želi da izvrši umesto funkcije* `funkcija`*. Ilustracija stanja memorije nakon pokretanja programa sa argumentom*
`<kod za pokretanje komandnog prozora u 83 bajta><45 slova A>0zww`
pod pretpostavkom da se bafer smešta na adresu `0x77777a30` *(što je ASCII vrednost za* `0zww`*), dat je na slici 3.8.*



Slika 3.8: Stanje programskog steka za kôd iz primera 3.5, neposredno nakon pokretanja programa (levo) i posle poziva funkcije `strcpy()` (desno)

Izmena `setjmp/longjmp` bafera

Izmena `setjmp/longjmp` bafera moguća je samo u programima koji koriste funkcije `setjmp` i `longjmp`. Ove funkcije omogućavaju čuvanje i rekonstruisanje svih vrednosti tekućeg stek okvira:

- poziv funkcije `setjmp(bafer)` čuva vrednosti tekućeg stek okvira u promenljivoj `bafer`;
- poziv funkcije `longjmp(buffer,...)` rekonstruiše stek okvir koji je bio sačuvan u promenljivoj `bafer` i vraća kontrolu programa u tačku neposredno nakon poziva odgovarajuće funkcije `setjmp`.

Ukoliko se prekoračenjem bafera može izmeniti vrednost promenljive `bafer`, tada se izmena može napraviti tako da novopostavljena vrednost odgovara stek okviru sa adresom povratka koja ukazuje na kôd koji napadač želi da izvrši. Na taj način se, prilikom poziva funkcije `longjmp`, stek okvir rekonstruiše koristeći izmenjene vrednosti, zbog čega se prilikom okončanaj rada funkcije kontrola

prepušta kodu koji je napadač izabrao. Na ovaj način realizovan je napad na program `Perl 5.003` [15].

4

Tehnike za otkrivanje prekoračenja bafera

Prekoračenje bafera je opasno, a često i kritično za pouzdanost i bezbednost softvera. Kako uobičajeno testiranje nije dovoljno za eliminisanje ovog problema, veoma je važno razvijanje tehnika (i pratećih alata) za automatsko pronalaženje prekoračenja bafera i za odbranu od bezbednosnih zloupotreba prekoračenja bafera. Kao što je već rečeno, ove tehnike dele se na tehnike zasnovane na statičkoj analizi (analizi pre faze izvršavanja) i tehnike zasnovane na dinamičkoj analizi (analizi za vreme izvršavanja programa) ili, kraće, na statičke i dinamičke tehnike. Prve tehnike za otkrivanje prekoračenja bafera bile su dinamičke i razvijene su početkom devedestih godina prošlog veka kao odgovor na konkretne bezbednosne napade. Ovi sistemi unapređivani su paralelno sa otkrivanjem novih bezbednosnih ili drugih problema vezanih za prekoračenja bafera. Od početka veka sve više značaja se pridaje statičkim sistemima koji mogu da budu potpuni i da otkriju sva moguća prekoračenja bafera pre faze izvršavanja.

4.1 Tehnike zasnovane na statičkoj analizi

Statičke tehnike imaju cilj da analiziraju izvorni kôd pre izvršavanja programa i da otkriju sva moguća prekoračenja bafera (a da ne generišu lažna upozorenja). Ne samo da ovaj cilj nije lako ostvariv, već je i teorijski neostvariv u potpunosti. Naime, problem zaustavljanja (eng. halting problem) je neodlučiv, pa prilikom analize programa u opštem slučaju nije moguće utvrditi da li se neki deo koda ikada izvršava, a time i da li on predstavlja opasnost za bezbednost programa. Postoje i drugi razlozi koji onemogućavaju ili otežavaju pravljenje statičkog sistema koji otkriva sva moguća prekoračenja bafera a da pritom ne generiše lažna upozorenja (to jest, da ne prijavljuje mesta na kojima zapravo do prekoračenja ne može da dođe). Zbog svega toga, sistemi zasnovani na statičkoj analizi ne mogu istovremeno da postignu i potpunost i saglasnost, pa raznim heuristikama pokušavaju da pokriju što veći broj mogućih prekoračenja bafera

i to sa što manjim brojem lažnih upozorenja.

Prvi statički sistemi za otkrivanje prekoračenja bafera razvijeni su krajem dvadesetog i početkom dvadesetog prvog veka. Zbog nesmanjenog (ili rastućeg) značaja problema prekoračenja bafera, tokom prethodnih nekoliko godina, dosta istraživanja je usmereno ka razvoju statičkih tehnika i alata, kako onih akademskih, tako i komercijalnih.

Statičke tehnike mogu se podeliti na leksičke i semantičke. U daljem tekstu biće ukratko opisani najznačajniji predstavnici obe ove grupe.

4.1.1 Leksičke tehnike

Leksičke statičke tehnike za otkrivanje prekoračenja bafera zasnivaju se na relativno jednostavnim sintaksnim proverama, definisanim u skladu sa najčešćim obrascima prekoračenja bafera. Zbog svoje jednostavnosti, alati zasnovani na leksičkoj analizi su vrlo efikasni i mogu se, kao dodatak editoru, koristiti za proveravanje koda dok se on unosi. S druge strane, alati zasnovani na leksičkoj analizi tipično imaju veliki broj lažnih upozorenja.

grep

`Grep`[1] je program opšte namene koji je originalno pisan za operativni sistem Unix, ali postoje i odgovarajuće verzije ovog programa za ostale operativne sisteme. Program se koristi u okviru komandne linije operativnog sistema. Za zadate regularne izraze i listu ulaznih datoteka, `grep` pretražuje datoteke i pronalazi one redove koji sadrže tekst koji odgovara zadatim regularnim izrazima.

U kontekstu pronalaženja mogućih prekoračenja bafera, `grep` se može koristiti za listanje onih linija koda u kojima se nalaze pozivi određenih nebezbednih funkcija. Na primer, pokretanjem programa sa

```
grep strcpy program.c
```

izdvajaju se sve linije koda datoteke `program.c` koje sadrže poziv nebezbedne funkcije `strcpy`. Nakon toga se za svaki izdvojen poziv mora ručno proveravati da li prestavlja opasnost za bezbednost programa.

Ovaj pristup ima značajne nedostatke. Na primer, broj dobijenih potencijalno problematičnih delova koda je prevelik čime se otežava uočavanje stvarnih problema. Takođe, ne postoji mogućnost sortiranja dobijenih podataka na inteligentan način, pa je zato potrebno uložiti puno dodatnog vremena i ekspertskog znanja da bi se interpretirali dobijeni rezultati.

ITS4

ITS4[2] [72] je alat za statičku analizu `C` i `C++` koda, koji je namenjen prvenstveno integrisanju u editor razvojnog okruženja, ali se može koristiti i za analizu već

[1] Grep je skraćenica za *Global Regular Expression Print*.

[2] ITS4 je skraćenica za *It's the Software Stupid! Security Scanner*

napisanog koda. Osnovna ideja ovog alata je da se olakša pisanje bezbednog koda korišćenjem sistema provera koji signalizira rizične pozive funkcija i daje uputstva kako te rizične pozive prevazići. Ova ideja potiče od već uveliko prihvaćene i korišćene tehnike signalizacije sintaksnih grešaka prilikom pisanja programa.

Realizacija alata ITS4 sadrži bazu podataka u kojoj su pohranjene razne vrste bezbednosnih problema. Baza je zasnovana na problemima koji se javljaju u Unix okruženjima. Za svaki problem čuva se njegov kratak i njegov detaljan opis, ozbiljnost rizika, kao i tip dodatne analize koju treba izvršiti kada se na ovaj problem naiđe. Baza ne sadrži informacije o tome kako prevazići probleme. Prilikom analize koda, ulazni kôd se deli na tokene koji se upoređuju sa bazom podataka. Kada ITS4 naiđe na potencijalni problem, pokreće se dalja analiza tog problema i na osnovu nje se zaključuje konačni stepen rizika i da li dati problem treba prijaviti programeru. Za sada postoji mogućnost integrisanja alata ITS4 samo u GNU Emacs editor.

ITS4 podržava označavanje delova koda koje alat treba da preskoči, omogućava različite vrste prikazivanja izlaznih rezultata, kao i prikaz podskupa svih rezultata u zavisnosti od izabranog stepena rizika. Na ovaj način se može redukovati količina izlaznih informacija i olakšati uočavanje stvarnih problema.

ITS4, za razliku od drugih alata za statičku analizu koda, ne koristi tehnike konteksno slobodnog parsiranja. Time se dobija na neophodnoj efikasnosti jer sistem treba da radi u realnom vremenu (tj. kako programer ukucava kôd, sistem treba da bude u mogućnosti da signalizira eventualne bezbednosne propuste). Za tehnike konteksno slobodnog parsiranja neophodno je da je program sintaksno ispravan i da se razmatra kao jedna nedeljiva celina. Ovo značajno otežava realizaciju osnovne ideje alata. Time što sto se vrši samo leksička analiza koda dobija se mogućnost analize svih mogućih varijanti programa nezavisno od pretprocesorskih direktiva. Tehnike konteksno slobodnog parsiranja ne omogućavaju analiziranje kompletnog koda odjednom, već samo jedne varijante programa u zavisnosti od pretprocesorskih direktiva.

Ovaj pristup zahteva visok nivo ljudskog učešća u otkrivanju prekoračenja bafera. Ipak, on značajno redukuje broj lažnih upozorenja koji se dobija korišćenjem programa `grep`. Time se olakšava pronalaženje ključnih bezbednosnih propusta u kodu.

Alat ITS4 je javno dostupan na Internet-adresi
`http://www.cigital.com/its4/`.

FlawFinder i RATS

Alati FlawFinder [80] i RATS[3] [68] zasnovani su na istim idejama kao ITS4. Oba sadrže baze podataka sa bezbednosnim problemima i vrše leksičku analizu koda i uparivanje sa elementima baze. RATS, pored podrške za jezike C i C++, ima i podršku za programske jezike Perl, PHP i Python. Ova dva alata su razvijana nezavisno, ali su veoma slična i trenutno se radi na njihovom spajanju.

[3] RATS je skraćenica za *Rough Auditing Tool for Security*.

Flawfinder je javno dostupan na Internet-adresi
http://www.dwheeler.com/flawfinder/
a RATS na adresi
http://www.securesw.com/rats/.

4.1.2 Semantičke tehnike

Semantičke statičke tehnike za otkrivanje prekoračenja bafera zasnivaju se na semantičkoj analizi izvornog koda i proveravanju uslova koji mogu dovesti do prekoračenja bafera. Ove tehnike obično uključuju analiziranje skupova linearnih ograničenja, ili drugih tipova formula, koji odgovaraju uslovima bezbednosti ili nebezbednosti pojedinačnih komandi.

BOON

Alat BOON[4] [79] problem otkrivanja prekoračenja bafera svodi na *problem celobrojnih ograničenja* (eng. integer constraint problem) i koristi jednostavne tehnike teorije grafova za efikasno rešavanje ovog problema. Da bi alat mogao da se koristi na velikim programima, napravljena su pojednostavljenja i korišćene su heuristike kojima se gubi na preciznosti, ali se dobija na efikasnosti. Ovaj alat ne može da pronađe sva prekoračenja bafera ali, i pored toga, daje veoma korisne rezultate.

Dve osnovne ideje koje ovaj alat uvodi su:

1. **Tretiranje znakovnih niski kao apstraktnog tipa podataka.** Kako se većina prekoračenja bafera dešava usled poziva standardnih bibliotečnih funkcija, ovaj metod modeluje `C` niske kao apstraktan tip podataka sa operacijama kao što su `strcpy`, `strcat` i slično.

2. **Modelovanje bafera kao parova celobrojnih intervala.** Umesto da se prati sadržaj bafera, pamte se dva broja: broj bajtova alociranih za bafer i broj bajtova koji se trenutno koriste.

Analiza se sastoji od obilaska drveta parsiranja koje se dobija za izvorni `C` kôd i generisanja sistema ograničenja celobrojnih opsega. Svaki sistem ograničenja ima jedinstveno najmanje rešenje koje alat pronalazi i na osnovu kojeg zaključuje da li i gde do prekoračenja bafera može da dođe.

Generisanje sistema ograničenja počinje tako što se svakoj celobrojnoj promenljivoj programa pridružuje opseg a svakoj znakovnoj nisci `s` pridružuju se dve promenljive: prva se odnosi na alociranu memoriju za nisku `s` (obeležava se sa $alloc(s)$) a druga na dužinu niske, to jest na broj bajtova koji se trenutno koriste za datu nisku (obeležava se sa $len(s)$). Svaka operacija nad niskama se modeluje u terminima uticaja na ove dve vrednosti. Dužina stringa se u ovom slučaju definiše uključujući i terminirajući znak `'\0'` što znači da je ograničenje koje treba da važi za svaku nisku `s`:

$$len(s) \leq alloc(s)$$

[4] BOON je skraćenica za Buffer Overrun detectiON.

C kôd	Interpretacija
`char s[n];`	$\{n\} \subseteq alloc(s)$
`strlen(s);`	$len(s) - 1$
`strcpy(dst, src);`	$len(src) \subseteq len(dst)$
`strncpy(dst, src, n);`	$\min(len(src), n) \subseteq len(dst)$
`s = "foo";`	$\{4\} \subseteq len(s), \{4\} \subseteq alloc(s)$
`p = malloc(n);`	$\{n\} \subseteq alloc(p)$
`p = strdup(s);`	$len(s) \subseteq len(p), alloc(s) \subseteq alloc(p)$
`strcat(s, suffix);`	$len(s) + len(suffix) - 1 \subseteq len(s)$
`strncat(s, suffix, n);`	$len(s) + \min(len(suffix) - 1, n)$ $\subseteq len(s)$
`gets(s);`	$[1, \infty] \subseteq len(s)$
`fgets(s, n, ...);`	$[1, n] \subseteq len(s)$
`sprintf(dst, "%s", src);`	$len(src) \subseteq len(dst)$
`sprintf(dst, "%d", n);`	$[1, 20] \subseteq len(dst)$
`snprintf(dst, n, "%s", src);`	$\min(len(src), n) \subseteq len(dst)$
`p[n] = '\0';`	$\min(len(p), n + 1) \subseteq len(p)$
`p = strchr(s, c);`	$p = s + n; [0, len(s)] \subseteq \{n\}$

Tabela 4.1: Primeri modelovanja operacija nad stringovima

Za svaku naredbu programa generiše se odgovarajuće ograničenje opsega. Celobrojni izrazi i promenljive se modeluju odgovarajućim operacijama nad opsezima. Za različite operacije nad niskama koriste se različita ograničenja, pri čemu su neka od njih data u tabeli 4.1. Leva kolona prikazuje `C` kôd koji se razmatra a desna odgovarajuće ograničenje koje dati kôd uvodi. Na primer, druga vrsta tabele govori da je povratna vrednost poziva funkcije `strlen()`, čiji je argument niska `s`, jednaka $len(s) - 1$. Treća vrsta tabele govori da je efekat funkcije `strcpy` (prepisivanje prvog argumenta drugim) da dužina prvog argumenta postaje jednaka dužini drugog argumenta.

Algoritam koji se koristi u ovom alatu efikasno pronalazi rešenja sistema ograničenja, a u praksi se najčešće ponaša linearno. Rešenje sistema ograničenja daje granice opsega svake programske promenljive posebno, i ne može da dâ informacije o odnosima koji postoje između promenljivih.

Kada se odrede svi mogući opsezi promenljivih, moguće je proveriti bezbednost korišćenja svake niske `s`. Ako pretpostavimo da je analiza utvrdila da su $len(s)$ i $alloc(s)$ vrednosti koje pripadaju intervalima $[a, b]$ i $[c, d]$, tada postoje tri mogućnosti:

1. Ako je $b \leq c$, može da se zaključi da za nisku `s` ne može da dođe do prekoračenja bafera.

2. Ako je $a > d$, može da se zaključi da se prekoračenje uvek dešava.

3. Ako se opsezi preklapaju, tada nije moguće zaključiti da li će do prekoračenja doći, ali se može zaključiti da postoji mogućnost da dođe do prekoračenja.

Zbog efikasnosti i jednostavnosti implementacije, ova analiza ne razmatra kontrolu toka naredbi i ne razmatra redosled naredbi. To bitno utiče na gubitak preciznosti. Prekoračenja prouzrokovana radom sa niskama, korišćenjem primitivnih operacija na pokazivačima, ne mogu da se detektuju. Takođe, pokazivači na funkcije, kao i pokazivači na pokazivače, u ovom metodu se ignorišu.

Alat je korišćen za testiranje različitih popularnih softverskih proizvoda, a dobijeni rezultati su ručno proveravani. Softverski paketi koji su testirani su *Linux net tools* paket, *Sendmail* verzija 8.9.3 i *Sendmail* verzija 8.7.5.

Performanse alata nisu velike ali je alat praktično upotrebljiv. Na primer, za analizu programa *Sendmail*, koji ima oko 32. hiljade linija `C` koda, potrebno je svega oko 15 minuta na Pentium III radnoj stanici. Od toga, potrebno je nekoliko minuta za parsiranje koda, ostatak za generisanje ograničenja, a svega nekoliko sekundi da se reše sistemi ograničenja.

Najveći nedostatak ovog alata je veliki broj lažnih upozorenja koje on proizvodi kao rezultat nepreciznosti analize opsega. Zbog toga i dalje mora da se odvoji značajno vreme za ručnu proveru potencijalnih prekoračenja bafera. Na primer, za *Sendmail* 8.9.3 alat generiše 44, upozorenja od kojih su samo 4 prava prekoračenja.

Alat BOON je javno dostupan na Internet-adresi
http://www.cs.berkeley.edu/~daw/boon/

SPLINT

Alat SPLINT[5] [49, 28, 29] (ranije poznat kao LCLint) je deo svoje funkcionalnosti preuzeo iz popularnog alata Lint [41], za statičku analizu `C` koda koji je razvijen kasnih sedamdesetih godina. U osnovi alata SPLINT nalaze se semantički komentari (u daljem tekstu *obeležja*, eng. annotations) koji se dodaju u kôd i u standardne biblioteke.

SPLINT prepoznaje obeležja na osnovu znaka `@` koji sledi nakon znaka `/*`. Kompajler ovakva obeležja tretira kao obične `C` komentare. Obeležja opisuju namere i pretpostavke programera. Na primer, ako se obeležje `/*@notnull@*/` koristi u deklaraciji parametra funkcije, onda ono ukazuje na to da vrednost, koja se prosleđuje tom parametru, ne sme da bude `NULL`. Obeležja mogu da opisuju i preduslove i postuslove funkcija. Za opisivanje preduslova i postuslova funkcija, koje operišu sa baferima, koriste se vrednosti `minSet`, `maxSet` (najmanji i najveći indeks bafera kojima se mogu dodeliti vrednosti), `minRead` i `maxRead` (najmanji i najveći indeks bafera za koje mogu da se čitaju vrednosti). Na primer, za funkciju `strcpy(s1,s2)`, preduslov korektnosti funkcije može se zapisati kao `/*@requires maxSet(s1)>=maxRead(s2)@*/`.

Na mestu poziva funkcije, SPLINT proverava da li su ispunjeni preduslovi funkcije, a nakon poziva funkcije, SPLINT podrazumeva da je ispunjen postuslov funkcije. SPLINT analizira telo funkcije počevši od obeležja preduslova i proverava da implementacija funkcije obezbeđuje važenje postuslova.

[5] Splint je skraćenica za Secure Programming Lint

SPLINT generiše preduslove i postuslove na nivou izraza u drvetu parsiranja ili, na mestima poziva funkcija, koristi obeležje date funkcije. Na primer, za deklaraciju `char buf[MAXSIZE]` generiše se postuslov `maxSet(buf)=MAXSIZE - 1` i `minSet(buf) = 0`, dok se na mestima u programu, na kojima se koristi vrednost `buf[i]` sa leve strane operatora dodele, generiše preduslov `maxSet(buf) >=i`. Za obradu petlji SPLINT koristi heuristike za prepoznavanje različitih formi najčešćih petlji. Korišćenje tih heuristika omogućava analizu velikog broja petlji koje se pojavljuju u stvarnim programima. Pored ugrađenih provera, SPLINT ima obezbeđen mehanizam za definisanje novih provera i obeležja za otkrivanje ranjivosti ili narušavanja osobina specifičnih za dati program.

Korišćenje SPLINT alata je iterativni proces. U prvoj iteraciji, SPLINT generiše upozorenja (o greškama u kodu), nakon čega programer ili menja kôd ili menja obeležja, pa prilikom sledećeg pokretanja alata dobija drugačiji, odnosno manji skup upozorenja. Ovaj postupak se ponavlja dok se ne eliminišu sva upozorenja. S obzirom da SPLINT može da proverava oko hiljadu linija koda u sekundi, ponovno pokretanje nije veliko opterećenje za programera. Međutim, radi obezbeđivanja ove efikasnost SPLINT nije precizan i generiše veliki broj lažnih upozorenja dok, pravi problemi mogu da prođu nezapaženo. Pored toga, programer je opterećen održavanjem velikog broja obeležja.

SPLINT je testiran na programu *wu-ftpd* (verzija 2.5.0). Detektovao je neke poznate, ali i neke do tada nepoznate greške u programu. Pokretanje alata bez dodatih obeležja proizvelo je 166 upozorenja za potencijalna prekoračenja bafera. Nakon iterativnog dodavanja 66 obeležja, broj upozorenja se smanjio na 101. Od tih upozorenja, 25 su bili stvarni problemi a 76 su bila lažna upozorenja. Od lažnih upozorenja 6 su potekla iz pretpostavki koje su se nalazile van programa, a 10 je generaisano zbog petlji koje su bile korektne, ali za čiju analizu nisu postojale odgovarajuće heuristike. Preostalih 60 lažnih upozorenja je bilo generisano zbog ograničenih mogućnosti koje SPLINT ima za rasuđivanje o aritmetici, kontroli podataka i aliasima.

Alat Splint je javno dostupan na Internet-adresi
http://www.splint.org/.

CSSV

Alat CSSV[6] [24] teži pronalaženju svih prekoračenja bafera sa veoma malim brojem lažnih upozorenja. U osnovi alata je označavanje koda pisanjem *ugovora* (eng. contracts). Ugovori se koriste da se obeleže očekivani ulazi, sporedni efekti i očekivani izlazi funkcije. CSSV je potpun i, bez obzira na preciznost zadatih ugovora, greške ne mogu da prođu nezapaženo.

CSSV proverava tri vrste grešaka:

1. Pristupe znakovnim niskama van njihovih granica.

2. Narušavanje preduslova i postuslova funkcija u odnosu na zadate ugovore.

[6] CSSV je skraćenica za *C String Static Verifyer*.

3. Pristupanje elementima bafera nakon znaka `'\0'` za kraj niske (ako on postoji).

Da bi se umanjio napor potreban za pisanje ugovora, CSSV može da funkcioniše i kada preduslovi i postuslovi ne opisuju kompletno ponašanje funkcije. Štaviše, moguće je koristiti CSSV sa ugovorima koji uključuju samo sporedne efekte funkcije, bez zadavanja preduslova i postuslova. Alat CSSV automatski pojačava i generiše preduslove i postuslove koristeći algoritam opisan u radu [51]. Analiza je, u ovom slučaju, manje precizna (postoji veći broj lažnih upozorenja), ali je i dalje potpuna (tj. greške u kodu ne mogu da prođu nezapaženo). U zavisnosti od zadatog ugovora, greška će biti detektovana kada se analizira telo funkcije ili kada se analizira mestu na kojem se funkcija poziva. Ako je kôd procedure izostavljen, kao što je to slučaj sa bibliotečnim funkcijama, CSSV pretpostavlja da su ugovori korektni i ne može da ih proveri. CSSV ne zahteva označavanja unutar samog koda, kao što to zahtevaju neki drugi alati. Informacije o pokazivačima se automatski obrađuju i zato ugovori ne moraju da sadrže informacije o tome kako se koriste pokazivači.

Analiza koda prolazi kroz nekoliko faza:

- U prvoj fazi se funkcija, koja se analizira, transformiše u semantički ekvivalentan kôd koji uključuje samo naredbe jezika `CoreC` [87] — podskupu skupa naredbi programskog jezika `C`. Osnovne osobine `CoreC` jezika su: naredbe kontrole toka su `if`, `goto`, `break` i `continue`, izrazi su bez sporednih efekata i ne mogu biti ugnježdeni, deklaracije nemaju inicijalizaciju. Algoritam transformisanja `C` programa u `CoreC` program je opisan u radu [87]. U ovoj fazi se, takođe, na mestima poziva funkcija dodaju `assert` naredbe koje odražavaju ugovore datih funkcija.

- U drugoj fazi, CSSV analizira interakcije između pokazivača [18, 19]. Za otkrivanje koji pokazivači mogu da ukazuju na istu osnovnu adresu, CSSV pojednostavljeno analizira ceo program — nezavisno od toka podataka. Zatim se na osnovu dobijenih podataka generišu informacije relevantne za funkciju koja se analizira.

- U trećoj fazi, kôd i informacije o pokazivačima se predaju transformatoru koji generiše odgovarajući celobrojni program.

- U četvrtoj fazi, rezultujući celobrojni program se obrađuje koristeći algoritam [12, 35] za celobrojnu analizu da bi se utvrdilo da li postoji potencijalno narušavanje navedenih ugovora. Zato što su celobrojna i pokazivačka analiza potpune i zato što se ugovori proveravaju na mestima poziva i u okviru procedure, greške ne mogu da prođu nezapaženo. Da bi se smanjio broj lažnih upozorenja, CSSV koristi preciznu celobrojnu analizu [23] koja opisuje linearne odnose između promenljivih.

Konačan rezultat analize koda je lista potencijalnih grešaka. Za svaku grešku generiše se odgovarajući primer koji može da pomogne programeru da utvrdi da li je u pitanju prava greška ili lažno upozorenje. Do lažnih upozorenja može

da dođe ukoliko postoje pogrešni ili previše slabi ugovori, zbog apstrahovanja koje se koristi u transformatoru, ili zbog nepreciznosti pokazivačke i celobrojne analize.

Značajna prednost alata CSSV u odnosu na druge alate je potpunost, kao i nizak nivo lažnih upozorenja. Na primer, prilikom testiranja programa *fixwrites* (koji je deo *web2c* aplikacije), CSSV je pronašao 8 grešaka pri čemu je od toga bilo svega dva lažna upozorenja. CSSV, za razliku od ostalih alata, može da barata punim jezikom `C`, uključujući dinamički alocirane strukture, višedimenzione nizove, višedimenzione pokazivače i kastovanje. Takođe, CSSV može samostalno da pojačava preduslove i postuslove funkcija tako da ne zahteva precizno ručno obeležavanje programa. S druge strane, ovaj alat ima veoma slabe performanse: za analiziranje oko 400 linija koda utroši više od 200 sekundi.

iCSSV [26] je nova verzija alata CSSV, koja eliminiše potrebu za ručnim obeležavanjem koda i efikasnije se izvršava.

ARCHER

ARCHER[7] [86] je alat dizajniran za rad sa velikim (već napisanim) programima. Zbog toga su u ovom alatu prisutne heuristike koje omogućavaju smanjivanje broja lažnih upozorenja, što je ključno za uočavanje pravih problema i bezbednosnih propusta u velikim programima. ARCHER prijavljuje ona mesta u programu za koja može da utvrdi da su moguća prekoračenja bafera i, za razliku od drugih alata, ne prijavljuje mesta u kodu za koja ne može da utvrdi da su bezbedna. Time se smanjuje izlaz i broj lažnih upozorenja koje alat generiše, ali je zato alat veoma neprecizan i ne uspeva da prijavi veliki broj grešaka.

ARCHER sprovodi analizu programa u tri faze:

- U prvoj fazi se `C` program parsira i prevodi u apstraktno sintaksno stablo.
- U drugoj fazi se vrši transformisanje stabla u kanonski oblik. Ova transformacija uvodi privremene promenljive da bi se eliminisali sporedni efekti i da bi se eliminisali ugnježdeni pozivi funkcija. Takođe, transformišu se izrazi koji sadrže operatore `&&`, `||` i `?` u odgovarajuće `if-else` iskaze. Rezultat je `C` program koji je sematnički ekvivalentan originalnom kodu, ali sa smanjenim brojem sintaksnih konstrukcija. Ova transformacija inspirisana je alatima CIL [59] i Microsoft AST Toolkit [10]. Slična transformacija vrši se i u alatu CSSV [24]. Od kanonskog oblika programa, konstruiše se graf toka kontrole za svaku funkciju i za sâm program.
- U trećoj fazi ARCHER vrši analizu dobijenih grafova, pri čemu počinje analizom grafova funkcija, a završava analizom grafa programa. Iscrpnom pretragom u dubinu, obilaze se mogući putevi kroz svaki graf, čime se simuliraju različiti mogući scenariji prolaska kroz grane programa. ARCHER time simbolički izvršava kôd, pri čemu informacije o promenljivama čuva u bazi podataka. Ako u grafu postoji ciklus (usled korišćenja petlje

[7] ARCHER je skraćenica za ARray CHeckER.

Program	vreme	broj fajlova	broj funkcija	broj linija koda	broj pronađenih grešaka	broj lažnih upozorenja
Sendmail	5′ 23″	134	829	97K	2	4
PostgreSQL	18′ 3″	404	5.7K	295K	9	0
OpenBSD	1h 26′ 5″	850	10.7K	628K	31	12
Linux	4h 10′ 4″	2158	36.5K	1.6M	118	39
Ukupno		1388	53.7K	2.6M	160	55

Tabela 4.2: Rezultati testiranja alata ARCHER

u funkciji) kroz taj ciklus se prolazi nasumičan broj puta ukoliko, što je najčešći slučaj, na osnovu uslova petlje nije moguće odrediti tačan broj prolazaka kroz petlju. Prilikom obilaska grafa, generišu se celobrojni uslovi čija nekonzistentnost signalizira pristupe nizovima van njihovih granica. Uslovi se šalju na proveru rešavaču (eng. solver) linearnih relacija, koji je razvijen specijalno za ARCHER. Ovaj rešavač je veoma efikasan, ali nije ni saglasan ni potpun. On sadrži razne heuristike koje ubrzavaju rešavanja linearnih ograničenja koja su često prisutna u radu sa nizovima i pokazivačima i koje smanjuju broj lažnih upozorenja. Konačan izlaz iz alata ARCHER je spisak potencijalnih prekoračenja bafera.

ARCHER je testiran na programima Sendmail (verzija 8.12.7), PostgreSQL (verzija 7.3.1), OpenBSD (verzija 3.2) i Linux (verzija 2.5.53) na procesoru Xeon 2.8GH sa 512M memorije. Rezultati testiranja sažeti su u tabeli 4.2.

UNO

UNO [39] je alat koji je namenjen otkrivanju tri vrste grešaka u `C` programima: korišćenje neinicijalizovanih podataka (eng. **U**ninitialized data), dereferenciranje NULL pokazivača (eng. **N**il-pointer dereferencing) i indeksiranje van granica niza (eng. **O**ut-of-bound array indexing).

UNO koristi `ctree`, modul javno dostupnog proširenja kompajlera [30], koji generiše drvo parsiranja za svaku funkciju u programu. Na osnovu drveta parsiranja gradi se složena struktura podataka koja svakom čvoru drveta parsiranja dodeljuje povezanu listu sačinjenu od svih objekata koji se koriste u tom čvoru, kao i svi objekti koji se koriste ispod datog čvora parsiranja. Dalje, za svaki objekat čuvaju se i dodatne informacije, na primer da li je objekat deklaracija, uzimanje vrednosti, dodela nove vrednosti, čitanje adrese, dereferenciranje objekta ili poziv funkcije. Takođe, prikupljaju se informacije o granicama nizova i o promenljivama koje se koriste kao indeksi za pristupanje nizovima. Drveta parsiranja se zatim prevode u grafove toka podataka, pri čemu se generiše i globalni graf poziva funkcija. Sva dalja analiza se sprovodi korišćenjem algoritma pretrage u dubinu grafa.

UNO analizira svaku izvornu datoteku posebno, izvršavajući detaljnu lokalnu analizu funkcija definisanih u datoj datoteci. Informacije se čuvaju da bi se omogućila dalja globalna analiza svih izvornih datoteka. Pri tome, ako se neki izvorni kôd promeni, potrebno je ponovo izvršiti detaljnu analizu samo te datoteke i onih koji od te datoteke zavise, tj. nije potrebno ponovo vršiti kompletnu analizu koda.

U prvoj fazi analize, UNO proverava korišćenje lokalnih promenljivih i statički deklarisanih promenljivih. U drugoj fazi, sprovodi se globalna analiza koja se bazira na informacijama koje su sakupljene za svaku izvornu datoteku posebno. UNO ne proverava indekse nizova koji su opšti izrazi ili koji uključuju pozive funkcija, već može da izvršava provere samo kada je indeks konstanta ili promenljiva, i kada se opseg indeksa može izračunati.

Program	vreme za lokalnu analizu	vreme za globalnu analizu	broj fajlova	broj linija koda	broj pronađenih grešaka	broj lažnih upozorenja
Sendmail	21.9″	24.6″	38	75K	3	2
Unravel	12.1″	4″	36	21K	12	3

Tabela 4.3: Rezultati testiranja alata UNO

UNO je testiran na programima unravel (verzija od 26. jula 1996. godine) i Sendmail (verzija 8.11.6). Analiza je vršena na računaru 250MHz SGI MIPS R10000 (radi poređenja, dobijena vremena izvršavanja treba povećati osam puta za procenu izvršavanja na računaru 2GHz Pentium-4). Vremena izvršavanja lokalne analize na ovim programima je uporediva sa vremenima potrebnim za prevođenje programa. Unravel je program koji se sastoji od oko 21K linija koda koji su raspoređeni u 36 datoteka. Lokalna analiza pojedinačnih datoteka trajala je 12.1 sekundu. Globalna analiza trajala je samo 4 sekunde. UNO je prijavio 15 grešaka u lokalnoj analizi. Svih 15 su se odnosila na upozorenja vezana za moguće neinicijalizovane promenljive u različitim kontekstima, pri čemu je bilo 12 stvarnih grešaka. Preostale tri prijavljene greške odnosile su se na putanje za koje UNO nije mogao da utvrdi da su nedostupne. Sendmail verzija 8.11.6 se sastoji od 75K linija koda raspoređenog u 38 datoteka. Lokalna analiza koda trajala je 21.9 sekundi a globalna 24.6 sekundi. UNO je prijavio 2 greške u lokalnoj analizi i 3 u globalnoj analizi. Ranije verzije ovog programa su statički analizirane tako da je veliki deo grešaka već uklonjen. UNO nije uspeo da detektuje greške prekoračenja bafera. Rezultati testiranja sažeti su u tabeli 4.3.

Alat UNO je javno dostupan na Internet-adresi
`http://spinroot.com/uno/`.

PolySpace C Verifier

PolySpace C Verifier je alat za statičku analizu C koda koji je razvila kompanija *PolySpace Tehnologies* [71]. Ovaj alat se delom zasniva na istraživanjima opisanim u radovima [11, 20, 21], ali detalje algoritama i tehnika koje se koriste u realizaciji ovog alata kompanija *PolySpace Tehnologies* ne želi da otkrije [90]. Fokus ovog alata je analiza ugrađenih (eng. embedded) aplikacija tako da je spektar grešaka koje alat otkriva nešto drugačiji nego kod prethodno pomenutih alata. PolySpace C Verifier omogućava otkrivanje deljenja nulom, izračunavanja korena negativnog broja, dereferenciranja NULL pokazivača, konflikata prilikom pristupanja deljenoj memoriji, korišćenja neinicijalizovanih promenljivih, uočavanje nezavršavajućih petlji i mrtvih delova koda, ali i otkrivanje prekoračenja bafera.

Drugi alati

Majkrosoftov alat PREfix [8], kao i njegova unapređena verzija PREfast [64], vrše simboličku analizu koda na sličan način kao sistem ARCHER. Ovi alati mogu da detektuju širi spektar grešaka u programu u odnosu na ARCHER, uključujući pristup neinicijalizovanom ili već oslobođenom delu memorije, dereferenciranje NULL pokazivača i curenje memorije. Za detektovanje prekoračenja bafera PREfix je manje precizan nego ARCHER.

Postoje i drugi alati za statičku analizu koda na programskom jeziku `C`. Ti alati namenjeni su otkrivanju različitih grešaka u programima, uključujući (za neke alate) i prekoračenja bafera.[8] Mogućnosti ovih alata opštije namene manje su od alata specijalizovanih za otkrivanje prekoračenja bafera.

4.1.3 Poređenje tehnika

Tehnike za automatsko pronalaženje prekoračenja bafera intenzivno se proučavaju i razvijaju poslednjih petnaestak godina. Međutim, opšteprihvaćeni test primeri (eng. benchmarks) na kojima je moguće izvršiti objektivno poređenje i validaciju postojećih alata još uvek nisu iskristalisani. Objektivno poređenje alata je teško i zbog njihovih različitih domena i interfejsa. Postoje svega tri publikovane studije koje se bave poređenjem statičkih alata za otkrivanje prekoračenja bafera [84, 90, 47]. Sudeći po ovim studijama, poređenje alata veoma zavisi od izabranih test primera.

U radu [84] vrši se poređenje alata ITS4, RATS, FlawFinder, Splint i BOON. Poređenje se vrši na veštačkom C kodu koji se sastoji od poziva 20 funkcija koje su rizične za bezbednost programa. Funkcije koje se koriste su izabrane na osnovu rada [81] i baze podataka koja čuva informacije o rizičnim funkcijama za alat ITS4. Tih 20 funkcija se poziva 44 puta, 21 put na bezbedan način i 23 puta na nebezbedan način, pri čemu se na greške prekoračenja bafera odnosi 13 bezbednih i 15 nebezbednih poziva. Kôd korišćen za ovo poređenje može se

[8] Jedan spisak takvih alata može se naći na Internet-adresi http://spinroot.com/static/

naći na Internet-adresi http://www.ida.liu.se/~johwi. Primer bezbednog i nebezbednog poziva iz pomenutog skupa je dat na slici 4.1.

```
char buffer[BUFSIZE];
if(strlen(input_string)<BUFSIZE)
    strcpy(buffer, input_string);    /* Bezbedan poziv */
strcpy(buffer, input_string);        /* Nebezbedan poziv */
```

Slika 4.1: Deo koda koji je korišćen za poređenje alata u radu [84]

Rezultati analize sažeti su u tabeli 4.4. Na osnovu ove analize, leksički alati, ITS4, RATS i FlawFinder se ponašaju dosta slično u pronalaženju grešaka u pozivima funkcija. Međutim, veoma se razlikuju u broju lažnih upozorenja, pri čemu ITS4 generiše najmanji broj lažnih upozorenja. Broj lažnih upozorenja je ipak, za sve ove alate, isuviše visok što onemogućava uočavanje pravih problema. S druge strane, alati BOON i Splint pronalaze isuviše mali broj grešaka. Splint je jedini alat koji može da napravi razliku između bezbednog i nebezbednog poziva funkcija `strcat()` i `strcpy()` i to umanjuje broj lažnih upozorenja u odnosu na ostale alate.

Alat	Broj pronađenih grešaka	Broj lažnih upozorenja	Broj neprijavljenih ispravnih poziva funkcija	Broj neprijavljenih grešaka
Flawfinder	22 (96%)	15 (71%)	6 (29%)	1 (4%)
ITS4	21 (91%)	11 (52%)	10 (48%)	2 (9%)
RATS	19 (83%)	14 (67%)	7 (33%)	4 (17%)
Splint	7 (30%)	4 (19%)	17 (81%)	16 (70%)
BOON	4 (27%)	4 (31%)	9 (69%)	11 (73%)

Tabela 4.4: Rezultati poređenja alata Flawfinder, ITS4, RATS, Splint i BOON (BOON je testiran samo za pronalaženje prekoračenja bafera).

Kako alati, koji se zasnivaju na leksičkoj analizi, proizvode preveliki broj lažnih upozorenja (zbog čega je potrebno puno dodatnih ljudskih provera), a alati koji vrše dublju analizu ne pronalaze dovoljan broj grešaka (što je opasno za bezbednost sistema), autori studije zaključuju da bi glavna svrha svih ovih alata pre bila podrška za vreme razvijanja i pregleda koda, nego zamena za ručno debagovanje i testiranje. Takođe, autori izvode zaključak da su alati Splint i BOON još uvek u fazi prototipa i da nijedan od alata nema zadovoljavajući odnos broja pronađenih grešaka sa brojem lažnih upozorenja. Detalji ove analize mogu se naći u [82].

U radovima [90, 47] vrši se poređenje alata ARCHER, BOON, Splint, UNO i PolySpace C Verifier. Poređenja se vrše samo prema mogućnosti alata da

detektuju prekoračenja bafera u C kodu, pri čemu su izabrane različite vrste test primera.

U radu [90] performanse nabrojanih alata se analiziraju korišćenjem 14 programa generisanih na osnovu 14 poznatih kritičnih prekoračenja bafera iz programa BIND [2], WU-FTP [78] i Sendmail [67]. Umesto originalnih programa, korišćen je kôd koji oponaša njihove rizične delove, jer većina alata nije u mogućnosti da analizira kompleksne i velike programe. Uporedo sa ovim programima, alati su testirani i nad korigovanim verzijama koje ne sadrže prekoračenja bafera. Test primeri korišćeni u ovoj studiji mogu se naći na Internet-adresi `http://www.ll.mit.edu/IST/corpora.html`. Kôd koji ilistruje programe, korišćene za ovo poređenje, je dat na slici 4.2.

```
void buildfname(gecos, login, buf)
    register char *gecos;
    char *login;
    char *buf;
{
...
register char *bp = buf;
/* fill in buffer */
for (p = gecos;
         *p != '\0' && *p != ','  && *p != ';'  && *p != '%';
                                                          p++)
{
    if (*p == '&')
    {
        (void) strcpy(bp, login);   /* Prekoracenje bafera */
        *bp = toupper(*bp);
        while (*bp != '\0')
            bp++;
    }
    else
        *bp++ = *p;                 /* Prekoracenje bafera */
}
*bp = '\0';                         /* Prekoracenje bafera */
}
```

Slika 4.2: Deo test primera koji je korišćen za poređenje alata u radu [90]

Rezultati ove studije sažeti su u tabeli 4.5. PolySpace i Splint su uspeli da detektuju deo prekoračenja bafera dok su preostala tri alata pokazala značajno lošije rezultate. BOON je upozorio na dva prekoračenja bafera ali nije bio u mogućnosti da napravi razliku između problematičnog i korigovanog koda. ARCHER je uspeo da otkrije samo jedno prekoračenje bafera i nije imao lažnih upozorenja. UNO nije generisao nijedno upozorenje za prekoračenje bafera (ali

Alat	verovatnoća pronalaženja prekoračenja bafera	verovatnoća izdavanja lažnog upozorenja	verovatnoća da neće biti izdato lažno upozorenje na korigovanom kodu
PolySpace	0.87	0.5	0.37
Splint	0.57	0.43	0.30
Boon	0.05	0.05	-
Archer	0.01	0	-
Uno	0	0	-

Tabela 4.5: Rezultati poređenja alata ARCHER, BOON, Splint, UNO i PolySpace C Verifier prema [90]

jeste za neke druge propuste u kodu).

Dobijeni rezultati ukazuju da jednostavni tipovi statičke analize ne omogućavaju otkrivanje prekoračenja bafera koje se dešavaju u stvarnim Internet server programima. Naime, procenat otkrivenih grešaka za tri od pet testiranih alata je ispod 5%. Ovako loš procenat može da bude posledica kompleksnosti analiziranih prekoračenja bafera. Iako su alati PolySpace i Splint imali visok procenat detektovanih prekoračenja (87% i 57%), njihova korisnost je diskutabilna zbog generisanja visokog procenta lažnih upozorenja (43% i 50%, a broj upozorenja po liniji koda je jedno upozorenje na 46 linija koda za PolySpace i jedno upozorenje na 12 linija koda za Splint). Ovi alati mogu da otkriju stvarna prekoračenja bafera, ali bi programeri, koji razvijaju kôd, verovatno ignorisali generisana upozorenja zbog previsokog broja lažnih upozorenja. Takođe, ovi alati ne mogu da razlikuju problematičan i odgovarajući korigovan kôd, što ih čini neupotrebljive za iterativnu upotrebu prilikom debagovanja. Detalji ove analize mogu se naći u [89].

U radu [47] analiza alata za otkrivanje prekoračenja bafera vrši se na posebno osmišljenom skupu test primera koji su izabrani s ciljem da odrede prednosti i slabosti alata za detektovanje različitih vrsta prekoračenja bafera. Autori su test primere odabrali tako da pokriju sve moguće vrste prekoračenja bafera u skladu sa klasifikacijom prekoračenja koju su razvili. Skup test primera se sastoji od kratkih programa (ukupno 291), koji u sebi sadrže prekoračenja bafera i odgovarajućih korigovanih programa koji ne sadrže prekoračenja. Test primeri korišćeni u ovoj studiji mogu se naći na Internet-adresi
`http://www.ll.mit.edu/IST/corpora.html`. Primer programa koji sadrži prekoračenje bafera je dat na slici 4.3.

Razultati ove analize sažeti su u tabeli 4.6. Vremena izvršavanja test primera data su u tabeli 4.7. Na osnovu ove studije, PolySpace C Verifier je pokazao superiornu mogućnost otkrivanja prekoračenja bafera, promašivši samo jedno od 291 prekoračenja. Međutim, vreme izvršavanja ovog alata na test primerima je neuporedivo veće u odnosu na ostale alate. ARCHER je takođe dao dobre rezultate, otkrivši više od 90% prekoračenja, pri tom ne proizvodeći lažna upozorenja. Splint je uspeo da otkrije značajno manji broj prekoračenja bafera i pri

```
int main(int argc, char *argv[])
{
int flag;
char buf[10];
flag = 1;
if (flag)
{
    /*  Prekoracenje bafera  */
    buf[10] = 'A';
}
return 0;
}
```

Slika 4.3: Test primer korišćen za poređenje alata u radu [47]

tom je imao najveći broj lažnih upozorenja. UNO nije imao lažnih upozorenja ali nije uspeo da detektuje skoro polovinu prekoračenja koja se javljaju u test primerima. BOON je pokazao najslabije rezultate, ne uspevši da otkrije čak ni ona prekoračenja za koje je prvenstveno dizajniran.

Alat	Procenat pronađenih prekoračenja	Procenat lažnih upozorenja	Stepen konfuzije
ARCHER	90.7	0.0	0.0
BOON	0.7	0.0	0.0
PolySpace	99.7	2.4	2.4
Splint	56.4	12.0	21.3
UNO	51.9	0.0	0.0

Tabela 4.6: Rezultati poređenja alata ARCHER, BOON, Splint, UNO i PolySpace C Verifier na osvnovu rada [47]; stepen konfuzije jednak je *ukupan broj prijavljenih prekoračenja i na pogrešnom i na korigovanom kodu/broj prijavljenih prekoračenja na pogrešnom kodu*

Test primeri korišćeni za analizu opisanu u radu [47] mogu da se koriste za ocenjivanje najboljeg mogućeg ponašanja alata, a ne mogu da se koriste za opšte zaključivanje kako će se alati ponašati na realnim programima. Naime, ako neki alat ne uspe da otkrije neku vrstu prekoračenja bafera na test primerima, onda verovatno neće moći da otkrije tu vrstu prekoračenja ni u kompleksnijoj, realnoj sredini. S druge strane, ako alat uspe da otkrije prekoračenje u ovim test primerima, tada ne može da se garantuje da će slično prekoračenje alat uspeti da detektuje u realnom kodu. Detalji ove analize mogu se naći u [46].

Kao što se vidi iz svih navedenih rezultata, različite studije daju bitno različite zaključke. To govori da je teško sprovesti objektivnu analizu i poređenje alata za otkrivanje prekoračenja bafera, ali i da ponašanje alata zavisi u velikoj meri od prirode programa koji se obrađuje. Opšti zaključak je da ovi alati

Alat	Ukupno vreme izvršavanja	Prosečno vreme izvršavanja
ARCHER	288′	0.247′
BOON	73′	0.063′
PolySpace	200820′ (56 sati)	172.526′
Splint	24′	0.021′
UNO	27′	0.023′

Tabela 4.7: Vreme izvršavanja za test primere iz [47]

mogu da pruže pomoć u razvoju programa, ali da ne mogu (još uvek) da zamene uobičajeno testirajne i reviziju koda. I pored pojedinačnih odličnih rezultata, svi testirani alati imaju neke slabosti — mali broj pronađenih grešaka, veliki broj lažnih upozorenja ili veliko vreme izvršavanja. Može se očekivati da će postojeći alati biti dalje razvijani i da će stečena iskustva u budućnosti dovesti do nove generacije, znatno uspešnijih alata.

4.2 Tehnike zasnovane na dinamičkoj analizi

Dinamička analiza koda ima za cilj otkrivanje i/ili prevenciju prekoračenja bafera u fazi izvršavanja koda. Dinamičke tehnike mogu da budu implementirane tako da vrše efikasnu zaštitu od poznatih napada u fazi izvršavanja programa. Jedan od osnovnih problema sa ovim alatima je u tome što ostaju neprimećena sva prekoračenja bafera u delovima programa koji još nisu izvršavani. Pored toga, program obično mora da bude zaustavljen čim se otkrije prekoračenje bafera. U okviru dinamičke analize postoji nekoliko različitih tehnika: dinamičko testiranje; dinamička prevencija zasnovana na korišćenju specijalizovanih kompajlera; pristup zasnovan na bibliotekama funkcija; pristup zasnovan na korišćenju operativnog sistema. U daljem tekstu biće ukratko opisani najznačajniji predstavnici svih ovih grupa.

4.2.1 Dinamičko testiranje

Alati, koji vrše dinamičko testiranje, zasnivaju se na dodavanju provera granica bafera u izvorni kôd programa ili u izvršnu verziju programa. Ovi alati omogućavaju otkrivanje prekoračenja bafera u programu izvršavanjem različitih test primera. Alati za dinamičko testiranje najčešće značajno usporavaju izvršavanje programa, a nekada mogu da zahtevaju i velike količine memorije. Ova cena može da bude prihvatljiva u fazi testiranja programa, ali predstavlja ozbiljan problem ukoliko se takav, prošireni kôd, koristi za vreme eksploatacije programa.

Prednost alata, koji koriste dinamičko testiranje, je to što su sve vrednosti promenljivih poznate u fazi izvršavanja. To omogućava uočavanje neispravnih dereferenciranja pokazivača i narušavanje granica bafera. Kada dođe do prekoračenja bafera, pronalaženje uzroka greške je jednostavno, s obzirom na to da alati

omogućavaju praćenje stanja hipa i svih programskih promenljivih. Međutim, osnovni problem ovog pristupa je što se mnoga prekoračenja dešavaju na mestima u programu do kojih se retko stiže. Testiranje takvih mesta u kodu je vremenski značajno zahtevnije u odnosu na korišćenje alata za statičku analizu koda (jer zahteva test primere koje pokrivaju sve moguće puteve kroz program što je praktično nemoguće). Prema tome, osnovni nedostatak ove tehnike je što ako konkretan test ulaz ne dovodi do prekoračenja bafera, to prekoračenje neće biti detektovano. Pored toga, neki programi ne mogu se testirati korišćenjem dinamičkih tehnika. Na primer, značajan deo koda operativnog sistema se zasniva na korišćenju drajvera za uređaje i oni se ne mogu realistično testirati, pre nego što se odgovarajući uređaji instaliraju. Verovatno najveći problem u korišćenju ove tehnike je veliko usporavanje izvršavanja (i po nekoliko desetina puta), kao i značajno povećavanje korišćene memorije.

Primeri alata koji se zasnivaju na principima dinamičkog testiranja su Purify [36, 37] i gdb [33]. Ovi alati mogu se koristiti za programske jezike C i C++.

Alat Purify dodaje provere granica bafera u izvršnu verziju programa (ne zahteva izvorni kôd programa). Ovaj alat, pored prekoračenja bafera, može da detektuje i curenje memorije, kao i pristupe neinicijalizovanoj ili nealociranoj memoriji. Međutim, on ne može da detektuje neispravne pristupe u okviru alocirane memorije. Alat je podesan za debagovanje, ali usporava izvršavanje programa 10 do 30 puta i značajno uvećava korišćenu memoriju. Alat gdb zahteva izvornu verziju koda i nije tako moćan kao Purify. Gdb omogućava prolazak kroz program u fazi izvršavanja i praćenje sadržaja steka i hipa.

Pored pomenutih alata, u okviru dinamičkog testiranja po svojim karakteristikama izdvajaju se tehnika mini-simulacija i tehnika ubacivanja grešaka. Ove tehnike odstupaju od standardnog dinamičkog testiranja po posebnim vrstama test primera koji se za ove alate generišu.

Mini-simulacija (eng. mini-simulation) je tehnika dinamičkog testiranja koja je nastala u okviru projekta PROTOS[9] [43]. Fokus ovog projekta su aplikacije koje koriste standardne protokole kao što su HTTP, TCP, SNMP i drugi. Za datu aplikaciju, ponašanja protokola se definišu ručno korišćenjem gramatika. Izvorni kôd aplikacije, koja se testira, nije potreban. Kada se generiše odgovarajuća gramatika, automatski se generiše velika klasa standardnih i nestandardnih test primera koji se zatim koriste za testiranje. Ovako generisani test primeri povećavaju verovatnoću pronalaženja grešaka u kodu, ali pripremanje jednog skupa testova traje jako dugo (i po nekoliko čovek meseci). PROTOS je testiran na 49 aplikacija od čega su kod 40 pronađena prekoračenja bafera.

Ubacivanje grešaka u softver (eng. software fault injection) je tehnika dinamičkog testiranja koja forsira program da dolazi u nestandardna stanja za vreme izvršavanja. Često je, razmatranjem kako se sistem ponaša u nestandardnim uslovima, moguće otkriti ranjivosti softvera. Nestandardni zahtevi se najčešće postavljaju unutrašnjim i/ili spoljašnjim interfejsima. Na primer, za testiranje servera, spoljašnjem interfejsu se mogu poslati nekorektni zahtevi ili prevelik broj zahteva. Time server prelazi u nestandardno stanje čijim je pažljivim

[9] PROTOS je skraćenica za *protocol security*.

posmatranjem, u nekim situacijama, moguće otkriti postojeće propuste i greške u programu. Popularni alati koji se zasnivaju na ovoj tehnici su FIST [34], Fuzz [56] i Ballista [48].

4.2.2 Prevencija zasnovana na specijalizovanim kompajlerima

Alati za dinamičku prevenciju, zasnovani na specijalizovanim kompajlerima, sprečavaju prekoračenja bafera i njihovu zloupotrebu za vreme izvršavanja programa. Sav izvorni kôd neophodno je prevesti uz pomoć specijalizovanog kompajlera koji proizvodi kôd sa dodacima za kontrolu granice bafera. Velika mana ovih alata je to što oni, ukoliko dođe do prekoračenja bafera, obično prekidaju izvršavanje programa. Time se zloupotreba prekoračenja bafera svodi na napad koji ima za cilj zaustavljanje rada programa (eng. denial of service attack).

Većina alata, koji vrše dinamičku prevenciju zasnovanu na specijalizovanim kompajlerima, ne može da spreči prekoračenja bafera koja se dešavaju na hipu, BSS segmentu i segmentu podataka. Oni značajno povećevaju veličinu koda i smanjuju performanse programa — vreme izvršavanja i veličina koda mogu da se povećaju i više od tri puta [32]. Alati koji se zasnivaju na ovoj tehnici su StackGuard [16], ProPolice [27], PointGuard [14] i StackShield [69].

Osnovni cilj alata StackGuard je zaštita adrese povratka funkcije upisivanjem specijalne, pomoćne vrednosti (eng. canary value) u programski stek pored adrese povratka funkcije. Ako se, usled prekoračenja bafera, adresa povratka funkcije izmeni, tada će i ova specijalna vrednost biti takođe izmenjena. To signalizira alatu da je došlo do napada. Ovaj alat, međutim, ne može da detektuje prekoračenja u drugim oblastima memorije dodeljene programu pa čak ni druge vrste prekoračenja na steku, kao što je, na primer, prekoračenja koje utiču na lokalne promenljive funkcije. Alat ProPolice je dodatak za gcc kompajler i zasniva se na sličnim idejama, ali ima mogućnost zaštite lokalnih promenljivih (koje se nalaze na steku). Alat PointGuard (koji je još uvek u fazi razvoja) se zasniva na sličnim idejama, ali sa ciljem da se obezbedi opštija zaštita programa. Majkrosoftov C/C++ kompajler, za Visual Studio .NET [55], ima specijalnu opciju (/GS) koja omogućava sličnu zaštitu kao StackGuard.

Alat StackShield ima mogućnost zaštite adrese povratka funkcije, ali i zaštitu od zloupotrebe pokazivača na funkcije. StackShield omogućava zaštitu adrese povratka funkcije tako što adresu čuva, ne samo na steku, već i na posebnom mestu koje ne može da bude izmenjeno. Zbog toga, ukoliko dođe do napada, alat može da nesmetano nastavi izvršavanje programa koristeći sačuvanu adresu. I ovaj pristup ima ograničenja — na primer, lokalne promenljive mogu da budu izmenjene, alat dozvoljava samo dubinu od 256 poziva na steku. StackShield ima i opciju pamćenja samo poslednje adrese na steku i završetka rada programa ukoliko dođe do napada.

Postoje i dinamičke tehnike zasnovane na specijalizovanim kompajlerima koje najpre vrše statičku analizu izvornog koda. Mesta u kodu, za koja se ne može statičkim putem utvrditi da su bezbedna, dopunjuju se proverama koje se sprovode u fazi izvršavanja programa, kako bi svaki napad na program

mogao da bude detektovan. Alati, koji se zasnivaju na ovoj tehnici, su CCured [58] i Cyclone [40]. Ove tehnike se ponekad svrstavaju u hibridne (statičko–dinamičke) tehnike.

Alat CCured transformiše C kôd u kôd na njegovom bezbednijem dijalektu (koji se isto naziva CCured) i to sa veoma malo intervencija korisnika ili u potpunosti bez njih. CCured izvršava statičku analizu programa i pokušava da odredi koji se pokazivači koriste bezbedno. Ako ne može sa sigurnošću da odredi bezbednost korišćenja nekog pokazivača, CCured dodaje provere u izvorni kôd, koje se koriste u vreme izvršavanja programa da bi se detektovala pristupanja van granica bafera. Ovako selektivno ubacivanje provera pomaže da se smanjenje performansi značajno umanji u odnosu na dinamičke alate koji provere ubacuju u kôd svuda gde se pristupa elementima bafera. Prosečno usporavanje izvršavanja programa, kod ovog alata, je za faktor dva. Ukoliko dođe do prekoračenja bafera, tada CCured prekida izvršavanje programa.

Alat Cyclon koristi sličan pristup kao alat CCured. Ovaj alat koristi bezbedan dijalekat programskog jezika C koji se takođe naziva Cyclon. Cyclon ima nekoliko klasa pokazivača koji se čuvaju drugačije nego u programskom jeziku C. Na primer, jedna vrsta pokazivača, pored svoje adrese, čuva i informacije o granicama odgovarajućeg bafera. Problem sa alatom Cyclone je u tome što nije moguće automatsko prevođenje C koda u Cyclon dijalekat, već je potrebno oko 10% koda ručno izmeniti. Usporenje izvršavanja programa, u odnosu na originalni program na programskom jeziku C, za Cyclone je i do tri puta.

Pored pomenutih pristupa, postoje i pristupi nadogradnje kompajlera (realizovani za gcc kompajler) koji modifikuju kompajler da dodaje provere granica za svaki bafer u programu [54, 5, 42]. Međutim, ovi alati značajno usporavaju izvršavanje programa.

4.2.3 Pristup zasnovan na korišćenju bibliotečkih funkcija

Pristup zasnovan na korišćenju bibliotečkih funkcija pokušava da reši problem prekoračenja bafera implementiranjem bezbednih alternativa za standardne C funkcije. Bezbedne alternative opasnih C funkcija vrše dinamičke provere granica bafera i štite vrednost adrese povratka funkcije. Najpoznatije implementacije biblioteka, koje sadrže bezbedne funkcije, su Libsafe [3], Libverify [4] i StrSafe [70].

Libsafe je dinamička biblioteka, koja prihvata pozive nebezbednih funkcija standardne glibc biblioteke (kao što su `strcpy`, `strcat`, `getwd`, `gets`, `scanf`, `realpath`, `sprintf`) i zamenjuje ih bezbednim funkcijama. Bezbedne funkcije računaju rastojanja od početka svakog bafera do osnovnog pokazivača funkcije i koriste to rastojanje kao dozvoljene granice za pisanje u okviru bafera. Ukoliko dođe do prekoračenja bafera, koje ugrožava adresu povratka funkcije, Libsafe prekida izvršavanje programa. Na ovaj način se sprečava menjanje adrese povratka funkcije, ali nije onemogućeno zlonamerno menjanje sadržaja lokalnih promenljivih u programu. Takođe, nije moguće sprečiti napade koji se odnose na druge segmente memorije.

Biblioteka Libverify je unapređenje biblioteke Libsafe. Ova biblioteka sadrži

širu klasu funkcija u odnosu na Libsafe i radi na drugačijem principu. Libverify pamti adresu povratka funkcije i sa njom vrši poređenje svaki put kada funkcija završava svoj rad, slično kao StackShield. Glavni nedostatak ovog alata je to što memorijski prostor, koji pamti adrese povratka funkcija, nije zaštićen od napada.

Majkrosoftova statička biblioteka StrSafe implementira bezbedne funkcije za rad sa stringovima. Da bi se koristila ova biblioteka, potrebno je zameniti svako pojavljivanje nebezbedne C funkcije u kodu, njenom bezbednom alternativom iz biblioteke StrSafe. Ovo rešenje je dobro ukoliko se koristi prilikom razvoja softvera, ali zahteva značajan programerski napor ukoliko je potrebno ispravljati kôd koji koristi nebezbedne pozive funkcija standardne C biblioteke.

4.2.4 Pristup zasnovan na nadgradnji operativnog sistema

Pristup zasnovan na nadgradnji operativnog sistema rešava problem prekoračenja bafera modifikacijom operativnog sistema. U okviru ovog pristupa, operativni sistem je zadužen za analizu programa koji se izvršava i za sprečavanje zloupotrebe prekoračenja bafera. Najpoznatije realizacije ovog pristupa su nadgradnja kernela za Linux [62] i sistem OpenBSD [52].

Nadgradnja kernela za Linux onemogućava izvršavanje koda koji se nalazi na steku. Time se sprečava zloupotreba prekoračenja bafera koja ima za cilj promenu adrese povratka funkcije. Ovo rešenje nije u potpunosti sigurno jer priroda operativnog sistema Linux zahteva izuzetke (na primer, za rad sa signalima) koji mogu da dovedu do ranjivosti jer u tim situacijama operativni sistem ipak mora da dozvoli izvršavanje koda koji se nalazi na steku [13]. Takođe, ovo rešenje ne pokriva moguće zloupotrebe prekoračenja u drugim segmentima memorije.

Sistem OpenBSD pretenduje na to da u potpunosti eliminiše mogućnost zloupotrebe prekoračenja bafera. OpenBSD koristi slučajno izabranu vrednost pozicije u memoriji za početnu adresu steka, kao i specijalne vrednosti (na sličan način kao StackGuard) za otkrivanje napada koji imaju za cilj menjanje adrese povratka funkcije. Napadač se sprečava da upiše i izvrši zlonameran kôd time što sistem deli memoriju na dva disjunktna dela, jedan po kojem samo može da se piše i drugi sa kodom koji može da se izvršava. Prebacivanje koda sa drugog na ovaj operativni sistem često nije jednostavno i to je značajna mana ovog rešenja.

4.2.5 Poređenje tehnika

Dinamičke tehnike za otkrivanje prekoračenja bafera su, kao što se vidi iz prethodnog teksta, vrlo raznorodne i alati zasnovani na dinamičkim tehnikama se često suštinski razlikuju po svojim svrhama i domenima. Većina njih je specijalizovana za rad u specifičnom okruženju, na primer, za rad u okviru integrisanog razvojnog okruženja ili kao deo operativnog sistema. Pored toga, većina alata je usmerena samo ka pojedinim tipovima prekoračenja bafera, na primer, ka otkrivanju prekoračenja koje ugrožavaju adresu povratka funkcije. Zbog toga,

nema mnogo smisla porediti alate koji imaju različite svrhe i domene. Ukoliko dva alata pokrivaju slične klase prekoračenja bafera, može se vršiti uporedna analiza njihove vremenske i memorijske zahtevnosti. Kako se u nastavku ovog teksta govori o statičkom otkrivanju prekoračenja bafera, ovde neće biti predstavljena detaljnija poređenja različitih dinamičkih alata. Izvesne uporedne analize mogućnosti i domena alata zasnovanih na dinamičkoj analizi mogu se naći, na primer, u radovima [85, 88].

5

Opis predloženog pristupa

U ovoj glavi opisana je nova statičke tehnika za otkrivanje prekoračenja bafera. Predloženi sistem je fleksibilan i modularan, a njegove komponente mogu biti lako kombinovane i zamenjene drugim komponentama.

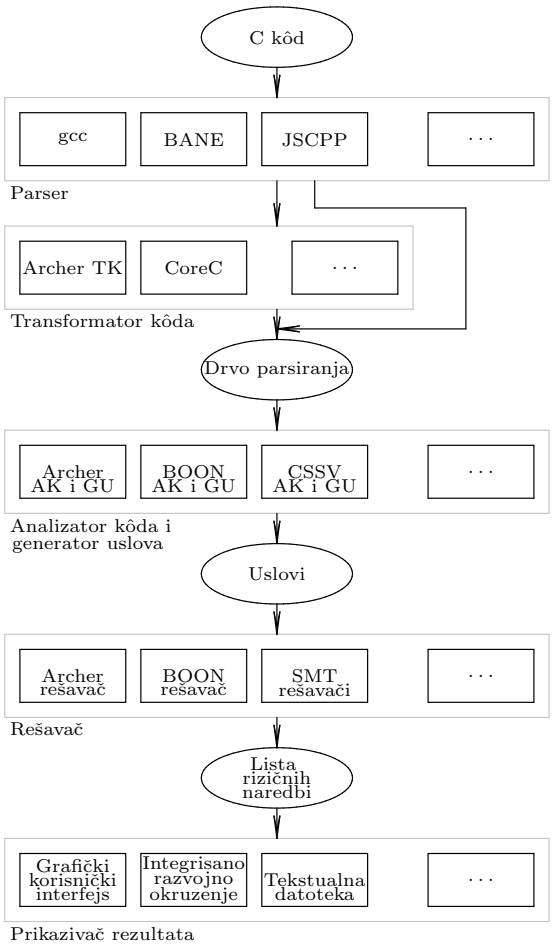
Predloženi sistem se oslanja na iskustva proistekla iz drugih sistema, ali pokušava da uvede određene novine i ispravi neke nedostatke postojećih sistema. Neke od novina proističu iz korišćenja automatskog dokazivača teorema koji, za bezbedne komande, može da izvede formalni dokaz bezbednosti, a za nebezbedne komande (bar u nekim situacijama), može da generiše model (jedan tok programa) koji dovodi do prekoračenja bafera.

Predloženi statički sistem uzima u obzir tok programa, interakciju između funkcija i kontekst poziva funkcija. Kôd se parsira, transformiše u jednostavniji oblik, analizom se preduslovi i postuslovi dodeljuju svakoj komadni krišćenjem spoljašnje biblioteke uslova. Na osnovu generisanih uslova formiraju se tvrđenja ispravnosti i tvrđenja neispravnosti komandi koja se, nakon odgovarajućih optimizacija i pripreme, šalju na proveru spoljašnjem dokazivaču teorema. Na osnovu rezultata dokazivača, obeležavaju se rizične komande i vrši se konačan prikaz rezultata.

5.1 Opšta arhitektura sistema za statičku analizu

U okviru postojećih sistema za statičku analizu prekoračenja bafera može se uočiti postojanje nekoliko faza u obradi programa (slika 5.1): parsiranje, transformacija koda, analiza koda i generisanje uslova, provera uslova i prikaz rezultata. Realizacija svake od pomenutih faza se veoma razlikuje kod pojedinačnih sistema. Sledi detaljan opis ovih faza.

Parsiranje Početna faza kod svih sistema je faza parsiranja. U okviru ove faze parser čita izvorni kôd, obrađuje ga i gradi drvo parsiranja koje se koristi u narednim fazama.



Slika 5.1: Modularna arhitektura alata za statičko otkrivanje prekoračenja bafera

Transformisanje koda Neki sistemi zahtevaju specifičan oblik koda koji se analizira. Zbog toga je potrebno transformisati parsirani kôd. Rezultatu transformacije odgovara C program koji zadržava semantiku početnog programa, ali sa suženim skupom komandi.

Analiza koda i generisanje uslova Kritičan modul za sve sisteme je modul koji analizira kôd (tj. drvo parsiranja) i generiše uslove ispravnosti/neispravnosti komandi. Način analize i generisanja ovih uslova je specifičan za svaki sistem. Pošto se pokazivačka aritmetika dobro modeluje teorijom linearne aritmetike, formule koje se u ovoj fazi generišu često pripadaju ovoj teoriji (nad realnim ili celim brojevima), ali se mogu koristiti i druge teorije.

Provera uslova U fazi provere uslova ispituje se važenje generisanih uslova. Priroda rešavača, koji proverava važenje uslova, zavisi od teorije kojoj generisani uslovi pripadaju. Pored toga, rešavač može da bude potpun i saglasan ili može da sadrži heuristike kojima se dobija na efikasnosti, ali zbog kojih nije potpun, a u nekim situacijama čak ni saglasan. Većina alata koristi sopstvene rešavače, ali je moguće koristiti i spoljašnje sisteme za ispitivanje važenja uslova.

Prikaz rezultata Prilikom generisanja uslova ispravnosti komande, uz svaki uslov, obično se čuva i linija izvornog koda za koju je uslov generisan. Zahvaljujući tome, konačan rezultat može da se pridruži odgovarajućem mestu u kodu. Rezultati mogu da budu prikazani na različite načine: u okviru tekstualne datoteke ili u okviru integrisanog razvojnog okruženja.

Postojeći sistemi za statičko otkrivanje prekoračenja bafera nemaju modularnu arhitekturu koja bi bila u skladu sa logičkim fazama kroz koje analiza prolazi. Zbog toga nije moguća uporedna analiza pojedinih komponentni sistema i svaki sistem može da bude testiran isključivo u celini, iako neke komponente sistema (na primer, analiza koda i generisanje uslova ispravnosti) mogu da imaju veoma dobre performanse, dok neke druge komponente (na primer ispitivanje generisanih uslova) mogu da imaju loše performanse. Modularna arhitektura sistema omogućila bi realno poređenje različitih alata i njihovih komponenti, kao i kombinovanje najefikasnijih modula različitih sistema (pod uslovom da moduli zadovoljavaju određene specifikacije).

U daljem tekstu opisuje se novi sistem za statičko otkrivanje prekoračenja bafera sa arhitekturom koja je motivisana navedenom opštom arhitekturom postojećih sistema.

5.2 Globalni opis predloženog sistema

Predloženi sistem se zasniva na statičkoj analizi programa, uzima u obzir kontrolu toka programa, kao i interakciju između funkcija i kontekst poziva funkcija. Kôd se parsira, transformiše u jednostavniji (ali semantički ekvivalentan) kôd, analizira liniju po liniju, a preduslovi i postuslovi se dodeljuju svakoj komadni na osnovu spoljašnje biblioteke uslova. Biblioteka uslova sadrži informacije o osnovnim operatorima programskog jezika C i o nebezbednim funkcijama iz standardne C biblioteke. Na osnovu generisanih preduslova i postuslova, formiraju se uslovi ispravnosti i uslovi neispravnosti komandi koji se, nakon odgovarajućih optimizacija i pripreme, šalju na proveru spoljašnjem dokazivaču teorema. Na osnovu rezultata dokazivača, obeležavaju se nebezbedne komande i vrši se konačan prikaz rezultata. U nekim situacijama, za nebezbedne komande, sistem može da ukaže na konkretan primer vrednosti promenljivih koje dovode do prekoračenja bafera.

Opšta arhitektura sistema (slika 5.2) je modularna. Moduli su kreirani u skladu sa arhitekturom opisanom u prethodnom poglavlju. Zahvaljujući tome

moguće je lakše porediti novokreirani sistem sa postojećim ili, ukoliko je potrebno, zameniti neki modul modulom iz drugog sistema. U narednim poglavljima dati su deteljni opisi gradivnih blokova sistema.

5.3 Modelovanje semantike programa

U definisanju preduslova i postuslova komandi, za modelovanje toka podataka i semantike programa, koristi se funkcija *value* i dve funkcije za rad sa pokazivačima *size* i *used*:

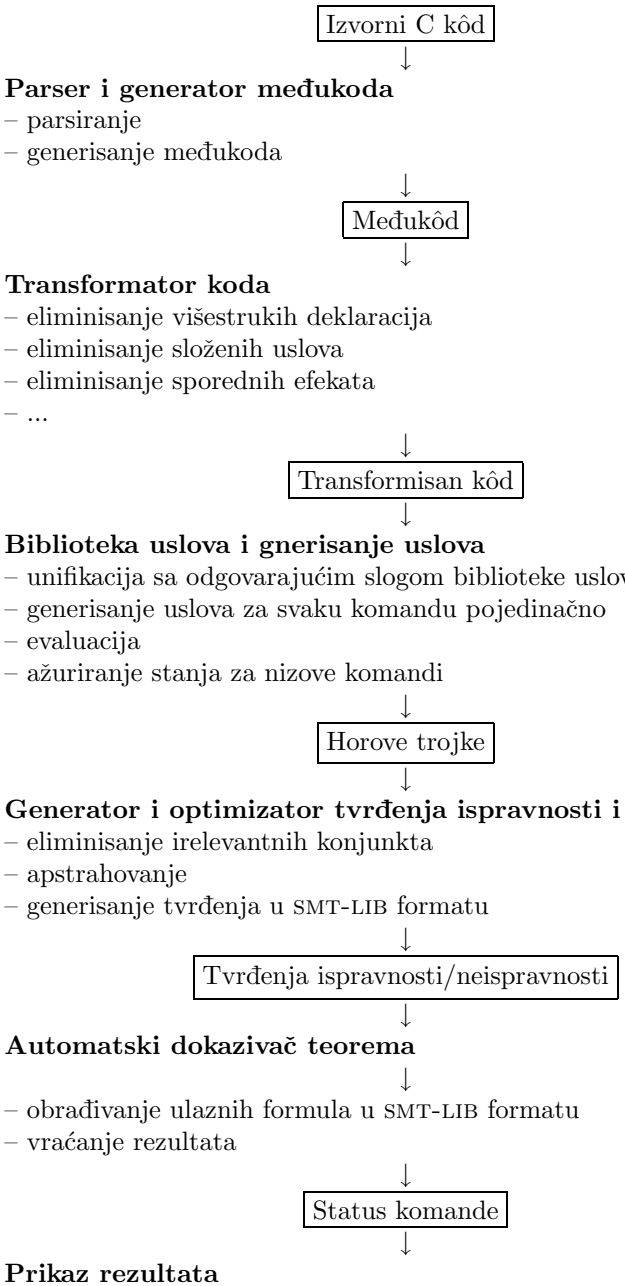
- funkcija *value* vraća tekuću vrednost promenljive;
- funkcija *size* vraća tekući broj elemenata alocirane memorije bafera;
- funkcija *used* je relevantna samo za niske (tj. nizove karaktera), vraća tekući broj iskorišćenih elemenata bafera (uključujući znak `'\0'` za završetak niske).

Sve tri funkcije imaju po dva argumenta. Prvi argument je ime promenljive, a drugi argument modeluje tok podataka tokom izvršavanja programa. Ovaj argument je celobrojnog tipa i on predstavlja *vremenski trenutak* (ili *stanje*) u kojem se razmatra vrednost promenljive. Na primer, $value(k, 0)$ vraća vrednost promenljive k u trenutku 0, $used(s, 1)$ vraća broj bajtova koje koristi bafer s u trenutku 1 i slično. Ove vrednosti su relativne — vezane za pojedinačne promenljive. Dakle, ne postoje apsolutna vremenska skala za sve promenljive u programu, već samo relativne skale za svaku promenljivu posebno. Štaviše, posebno se, za jednu promenljivu, prate i čuvaju vrednosti za funkcije *value*, *size* i *used* (jer se veličina i broj korišćenih bajtova niza mogu nezavisno menjati, a i nezavisno od vrednosti samog pokazivača). Ove vrednosti mogu biti relativne i za jednu promenljivu, tj. mogu polaziti od neke referentne tačke u programu. Prilikom obrađivanja niza komandi, stanja za *value*, *size* i *used* se uvećavaju u zavisnosti od komande i prethodnog stanja (za datu promenljivu i funkciju).

5.4 Parsiranje C programa i transformisanje koda

Parser čita kôd iz izvornih datoteka, obrađuje ga i gradi drvo parsiranja. Drvo parsiranja se zatim prevodi u specifičan međukôd koji čini sistem nezavisnim od parsera. Zahvaljujući tome što se celokupna dalja obrada vrši nad međukodom, parser se, po potrebi, može jednostavno zameniti nekim drugim parserom (na primer, parserom koji je efikasniji ili parserom koji se nalazi u sklopu nekog razvojnog okruženja).

Generisan međukôd se šalje na obradu transformatoru. Transformator prevodi međukôd u kôd koji je semantički ekvivalentan početnom kodu, ali čiji je dozvoljen skup komandi redukovan u odnosu na komande programskog jezika C. Transformacija sadrži niz koraka koji obuhvataju eliminisanje višestrukih deklaracija i inicijalizacija u okviru deklaracija, eliminisanje sporednih efekata



Slika 5.2: Arhitektura sistema

(kao što su eliminisanje složenih logičkih izraza, eliminisanje operatora `=`, `++` i `--` koji se nalaze sa desne strane operatora dodele, eliminisanje ugnježdenih poziva funkcija), eliminisanje svih složenih operatora dodele (kao što su `+=`, `-=`, `*=` itd.), i slično. Transformacijom koda potencijalno se uvode nove promenljive u program. Kontrola toka programa se uprošćava, eliminišu se `else` grane kod `if-else` iskaza, a sve petlje se svode na odgovarajuće `do-while` petlje. Opisana transformacija koda uprošćava i ubrzava naredne faze obrade. Ciljni jezik, koji se ovde koristi, je sličan jeziku CoreC koji je opisan u radu [87]. U tom radu je detaljno opisano i zašto se uvodi ovakav jezik, kao i kako se vrši transformacija iz standardnog jezika C u ovaj jezik.

5.5 Analiza koda i generisanje uslova ispravnosti

Analiza koda vrši se nad transformisanim kodom. Za generisanje opštih preduslova i postuslova pojedinačnih komandi koristi se biblioteka uslova. Uzimajući u obzir ove preduslove i postuslove, kao i kontekst u kojem se komanda nalazi, vrši se evaluacija konstantnih izraza i ažuriranje stanja za sve promenljive koje učestvuju u programu.

Biblioteka uslova

Biblioteka uslova je spoljašnja biblioteka koju korisnik može menjati. Inicijalno, biblioteka uslova sadrži informacije o standardnim C operatorima i funkcijama iz standardne C biblioteke. Prilikom obrađivanja ulaznih C programa, biblioteka se može privremeno proširiti da bi obuhvatila i specifične funkcije programa koji se obrađuje.

Biblioteka uslova čuva trojke oblika *(preduslov, komanda, postuslov)*. Semantika trojke (ϕ, Ψ, ψ) biblioteke uslova je:

- ako važi uslov ϕ, onda je upotreba komande Ψ bezbedna;
- ako važi uslov $\neg\phi$, onda je upotreba komande Ψ neispravna;
- nakon izvršavanja komande Ψ, važi uslov ψ.

Trojka (ϕ, Ψ, ψ) je slična Horovim trojkama, ali postoji razlika u semantici. Naime, dok je u ovom slučaju fokus na proveri da li se komanda Ψ može bezbedno primeniti, odnosno da li važi uslov ϕ, sematika Horovih trojki usmerena je ka transformaciji koju obezbeđuje komanda Ψ.

U predloženom pristupu, u slogu (ϕ, Ψ, ψ), u okviru formule ϕ, za bilo koju promenljivu figuriše samo (relativno) stanje 0 (*pre izvršenja komande*). U okviru formule ψ, za bilo koju promenljivu može da figuriše i (relativno) stanje 1 (*posle izvršenja komande*). Zahvaljujući transformisanju koda, u okviru formule ψ ne mogu da figurišu stanja veća od 1.

Primer 5.1 *Biblioteka uslova može da sadrži slog:*

$$(\mathit{size}(x,0) \geq \mathit{used}(y,0) \text{ , } \mathit{strcpy(x,y)}, \ \mathit{used}(x,1) = \mathit{used}(y,0)) \tag{5.1}$$

sa značenjem:

- *da bi poziv fiunkcije* `strcpy` *bio bezbedan, potrebno je da važi uslov* $size(x,0) \geq used(y,0)$*;*
- *da bi poziv fiunkcije* `strcpy` *bio neispravan, potrebno je da važi uslov* $\neg(size(x,0) \geq used(y,0))$*;*
- *nakon poziva funkcije* `strcpy` *važi uslov* $used(x,1) = used(y,0)$.

Slog 5.1 ukazuje na promenu stanja bafera: $size(x,0) \geq used(y,0)$ *ukazuje na potrebne odnose pre izvršavanja komande (stanje* 0*), dok* $used(x,1) = used(y,0)$ *ukazuje da će, nakon izvršavanja komande (u stanju* 1*), promenljiva* x *koristiti isti broj bajtova kao promenljiva* y *pre izvršavanja komande.*

Preduslov i postuslov komande

Biblioteka uslova se koristi za generisanje preduslova i postuslova pojedinačnih komandi. Preduslov komande K, koja nastaje transformacijom koda, se konstruiše na sledeći način: ako postoji slog (ϕ, Ψ, ψ) u biblioteci uslova, takav da postoji jednosmerno uparivanje koje K uparuje sa Φ, tj. ako postoji zamena σ takva da je $K = \Psi\sigma$, tada je preduslov komande K jednak $\phi\sigma$, a postuslov je jednak $\psi\sigma$.

Primer 5.2 *Razmatrimo sledeći fragment koda:*

```
char src[200];
fgets(src,200,stdin);
```

i pretpostavimo da biblioteka uslova sadrži sledeće slogove:

preduslov	*komanda*	*postuslov*
–	`char x[N]`	$size(x,1) = value(N,0)$
$size(x,0) \geq value(y,0)$	`fgets(x,y,z)`	$used(x,1) \leq value(y,0)$

Preduslov komande `fgets(src,200,stdin)` *računa se na osnovu drugog sloga biblioteke uslova. Jednosmerno uparivanje koje* `fgets(src,200,stdin)` *uparuje sa* `fgets(x,y,z)` *je* $\sigma = [x \to src, y \to 200, z \to stdin]$ *pa je, prema tome, preduslov* ϕ *ove komande određen na sledeći način:*

$$\phi = (size(x,0) \geq value(y,0))[x \to src, y \to 200, z \to stdin]$$

$$= (size(src,0) \geq value(200,0)).$$

Prilikom generisanja uslova, za svako *size* se dodaje uslov nenegativnosti (na primer, $size(src,0) \geq 0$), sa značenjem da je rezervisan prostor za bafer uvek nenegativan, a za svako *used* se dodaje uslov pozitivnosti (na primer, $used(src,1) > 0$), sa značenjem da iskorišćen prostor za bafer uvek sadrži barem znak `'\0'`. Zbog jednostavnosti, u narednom tekstu ovi uslovi će se podrazumevati i kada ne budu eksplicitno navedeni. Vrednosti funkcija $size(s,.)$ i $used(s,.)$ su nezavisne i zbog toga se ne dodaje nikakav uslov o njihovoj povezanosti.

Nad izračunatim preduslovima i postuslovima komandi, vrši se evaluacija konstantnih izraza, a izrazi koji uključuju pokazivačku aritmetiku se pojednostavljuju. Na primer,

- $size(\text{"}test\text{"}, .)$ se zamenjuje sa 5;
- $value(200, .)$ se zamenjuje sa 200;
- $size(buf + k, .)$ se zamenjuje sa $size(buf, .) - value(k, 0)$, pri čemu je buf pokazivač ili niz, a k celobrojna vrednost.

Nakon evaluacije preduslova i postuslova pojedinačnih komandi, stanja funkcija *value*, *size*, i *used* se ažuriraju sa ciljem da se uzme u obzir širi kontekst komandi. Ažuriranje se vrši komandu po komandu na sledeći način:

- stanja za funkcije *value*, *size* i *used* se čuvaju nezavisno:
  - za svaku promenljivu v, početno stanje za funkciju *value* je 0;
  - za svaku pokazivačku promenljivu v, početno stanje za funkciju *size* je 0;
  - za svaku promenljivu v tipa `char*`, početno stanje za funkciju *used* je 0;
- neka f označava jednu od funkcija *value*, *size* i *used*. Važi sledeće:
  - ako je tekuće stanje promenljive v za vrednost funkcije f jednako S, tada se u tekućoj komandi $f(v, 0)$ zamenjuje sa $f(v, S)$, a $f(v, 1)$ se zamenjuje sa $f(v, S + 1)$;
  - ako se $f(v, 1)$ pojavljuje u postuslovu komande, tada se tekuće stanje S za promenljivu v i funkciju f povećava za jedan;
- kada god se uveća stanje funkcije *value* za bafer p (odnosno, kada god se promeni vrednost pokazivača p)
  - uveća se za jedan i tekuće stanje funkcije *size* za promenljivu p;
  - ako je pokazivač p tipa `char*`, tada se uveća za jedan i tekuće stanje funkcije *used* za promenljivu p.

Primer 5.3 *Neka su kôd i biblioteka uslova isti kao u primeru 5.2. Nakon inicijalne transformacije koda (u ovom primeru transformisani kôd je isti kao i originalni kôd), vrši se analiza. Preduslovi i postuslovi se generišu na osnovu biblioteke uslova:*

preduslov	*komanda*	*postuslov*
–	`char src[200]`	$size(src, 1) = value(200, 0)$
$size(src, 0) \geq value(200, 0)$	`fgets(src,200,stdin)`	$used(src, 1) \leq value(200, 0)$

Evaluacijom konstantnih izraza, $value(200, 0)$ *se transformiše u* 200 *i dobijaju se sledeći uslovi:*

preduslov	*komanda*	*postuslov*
–	`char src[200];`	$size(src, 1) = 200$
$size(src, 0) \geq 200$	`fgets(src,200,stdin);`	$used(src, 1) \leq 200$

Ažuriranjem stanja funkcija size i used dobijaju se sledeći uslovi:

preduslov	*komanda*	*postuslov*
–	`char src[200];`	$size(src, 1) = 200$
$size(src, 1) \geq 200$	`fgets(src,200,stdin);`	$used(src, 1) \leq 200$.

Preduslovi i postuslovi za naredbe koje se nalaze u okviru bloka `if` komande (`else` delovi se eliminišu u fazi transformisanja koda), konstruišu se na isti način kao i za komande koje se nalaze van bloka `if` komande. Postuslov, koji se vezuje za ulazak u blok komande `if(p)`, je jednak p. Postuslov `if` komande se definiše na sledeći način:

$$\text{postuslov}(\texttt{if(p) \{ K1; K2; ... Kn; }}) =$$

$$(p \wedge postuslov(K1) \wedge postuslov(K2)... \wedge postuslov(Kn)) \vee (\neg p \wedge A).$$

Formula A odnosi se na konjunkciju uslova dobijenih usled ažuriranja stanja na osnovu naredbi koje se nalaze u okviru `if` komande. Stanja u okviru `if` komande se ažuriraju na sledeći način:

1. neka je f jedna od funkcija *value*, *size* ili *used* i neka je S stanje promenljive v za funkciju f pre izvršavanja komande `if`;
2. stanja uslova u okviru komande `if` se ažuriraju na isti način kao i za ostale komande; ažuriranjem stanja uslova u okviru `if` komande stanje promenljive v za funkciju f postaje S';
3. u konjunkciju uslova A dodaje se jednakost $f(v, S) = f(v, S')$;
4. odgovarajuća jednakost se dodaje u konjunkciju uslova A za sve promenljive čija se stanja menjaju u okviru `if` komande.

Zahvaljujući ovom mehanizmu, tekuće stanje za bilo koju promenljivu v nakon komande `if` je uvek S', bez obzira da li je uslov p ispunjen ili nije. Generisanje preduslova i postuslova za naredbe u okviru `if` komande prikazano je na slici 5.3.

preduslov	komanda	postuslov
–	`if(p)`	
–	`{`	p
$preduslov(K1)$	`K1;`	$postuslov(K1)$
$preduslov(K2)$	`K2;`	$postuslov(K2)$
	`...;`	...
–	`}`	$(p \wedge postuslov(K1) \wedge postuslov(K2)...) \vee (\neg p \wedge A)$

Slika 5.3: Generisanje uslova ispravnosti za `if` komandu

Analiza petlji i generisanje postuslova petlje se, za sada, vrši na ograničen način. Prateći ideje iz [49], sistem testira samo prvi prolazak kroz petlju (što je razumno i dovoljno u nekim slučajevima) i pokriva pozive funkcija sa konstantnim argumentima. Postuslov petlje je konjunkcija negacije uslova ulaska u

petlju i postuslova komandi unutar petlje. Ovako generisan postuslov može da ograniči otkrivanje prekoračenja bafera nakon petlje.[1]

Preduslovi i postuslovi za korisnički definisane funkcije mogu se uneti u biblioteku uslova. Ukoliko korisnik to ne uradi za neku funkciju `f`, sistem ipak može da nastavi sa radom, a pozivanja na formule $preduslov(f)$ i $postuslov(f)$ biće zamenjena odgovarajućim novim promenljivama (čime će se uvek razmatrati opštija tvrđenja). Prema tome, sistem može da radi i bez intervencija korisnika, iako sa smanjenom moći[2]. Mnogi sistemi za statičku analizu koda takođe zahtevaju neke vrste obeležavanja korisnički definisanih funkcija. Ovo ograničenje nije kritično jer mnoga (ili većina) prekoračenja bafera ne zavise od kompleksne strukture toka kontrole podataka i međuproceduralne komunikacije, već su posledica nebezbednog korišćenja standardnih C funkcija [49, 79].

5.6 Tvrđenja ispravnosti i neispravnosti

Za svaku komandu, generiše se tvrđenje koje, ako se dokaže, potvrđuje ispravnost komande. Takođe, generiše se i tvrđenje koje, ako se dokaže, potvrđuje neispravnost komande.

Za komandu K, neka je Φ konjunkcija postuslova za sve komande koje prethode komandi K (u okviru iste funkcije). Tada važi:

1. Komanda K je *ispravna* (*bezbedna*), tj. *nikada ne dovodi do greške* prilikom izvršavanja programa, ako je formula $\Phi \Rightarrow preduslov(K)$ valjana (podrazumeva se univerzalno zatvorenje formule).

2. Komanda K je *neispravna*, tj. *uvek dovodi do greške* prilikom izvršavanja programa, ako je formula $\Phi \Rightarrow \neg preduslov(K)$ valjana (podrazumeva se univerzalno zatvorenje formule).

3. Komanda K je *nebezbedna*, tj. prilikom izvršavanja programa *može da dovede do greške*, ako ne važi uslov (1) i ne važi uslov (2).

4. Komanda K je *nedostupna*, tj. ova komanda se nikada neće izvršiti, ako važi i uslov (1) i uslov (2).

Primeri ispravne, neispravne, nebezbedne i nedostupne komande su dati na slici 5.4.

Generisana tvrđenja ispravnosti i neispravnosti komandi se šalju na proveru automatskom dokazivaču teorema. Pre nego što se tvrđenja pošalju dokazivaču, vrši se priprema tvrđenja. U okviru pripreme, tvrđenja se optimizuju i apstrahuju.

Tvrđenja se pojednostavljuju eliminisanjem irelevantnih konjunkta. Iz tvrđenja $\Phi \Rightarrow preduslov(K)$, uklanjaju se svi *irelevantni* konjunkti. Konjunkt iz

[1] Uprkos činjenici da heuristike za rad sa petljama mogu da budu veoma efikasne i da pokrivaju širok spektar petlji, u daljem radu planiramo da proširimo sistem da vrši saglasnu analizu petlji na sličan način kao u [24].

[2] U daljem radu planiramo razvijanje tehnika za automatsko generisanje preduslova i postuslova funkcija kako bi sistem bio u potpunosti automatizovan, slično kao u sistemu [26].

```
int k;
char buf[10];

k = 5;
/*Ispravna komanda:*/
buf[k]='a';

k = 10;
/*Neispravna komanda:*/
buf[k]='a';

k = 15;
/*Nebezbedna komanda:*/
fgets(buf, k, stdin);

if(k<0)
    /*Nedostupna komanda:*/
    fgets(buf, k, stdin);
```

Slika 5.4: Primeri ispravne, neispravne, nebezbedne i nedostupne komande.

Φ je *irelevantan* za tvrđenje ako nije *relevantan* za formulu $preduslov(K)$. Da li je konjunkt iz Φ *relevantan* za formulu $preduslov(K)$ određuje se rekurzivno:

- konjunkt je relevantan za formulu $preduslov(K)$ ako u sebi sadrži neku od promenljivih koja se pojavljuje u formuli $preduslov(K)$;
- konjunkt je relevantan ako uključuje promenljivu ili funkcijski poziv koji se pojavljuje u nekom relevantnom konjunktu.

Na primer, eliminisanjem irelevantnih konjunkata iz tvrđenja:

$$(size(a,0) = 100) \wedge (size(a,0) = used(b,0)) \wedge (used(c,1) = 10)$$

$$\Rightarrow (used(b,0) > 50)$$

dobija se tvrđenje:

$$(size(a,0) = 100) \wedge (size(a,0) = used(b,0)) \Rightarrow (used(b,0) > 50).$$

Sledeća faza pripreme, apstrahovanje terma koji ne pripadaju linearnoj aritmetici, je zamena terma novim promenljivama. Na primer,

- $size(t,2)$ se apstrahuje u $size\_t\_2$;
- $value(x,N)$ se, zarad jednostavnije notacije, apstrahuje u x_N;
- $preduslov(f)$ se apstrahuje u $preduslov\_f$.

Ova transformacija nije potpuna, ali jeste saglasna: ako je apstrahovana formula valjana, tada je i originalna formula takođe valjana. Obratno ne mora da važi, što znači da sistem nije kompletan za komande koje uključuju apstrahovane termove (recimo za nerazrešene pozive funkcija).

Primer 5.4 *Za kôd iz primera 5.3, tvrđenje ispravnosti za komandu* `fgets (src,200,stdin);` *je (podrazumeva se univerzalno zatvorenje):*

$$(0 \leq size(src, 1) \wedge size(src, 1) = 200) \Rightarrow (size(src, 1) \geq 200).$$

Eliminisanjem irelevantnih konjunkata, tvrđenje se ne menja. Nakon apstrahovanja, tvrđenje postaje:

$$(0 \leq size\_src\_1) \wedge (size\_src\_1 = 200) \Rightarrow (size\_src\_1 \geq 200).$$

Ovo tvrđenje može biti dokazano dokazivačem teorema koji pokriva linearnu aritmetiku.

5.7 Pozivanje SMT rešavača

SMT rešavači ispituju nezadovoljivost formula. Kako je proizvoljna formula valjana ako i samo ako je njena negacija nezadovoljiva, dokazivaču treba proslediti negaciju formule čiju valjanost treba ispitati. Dakle, formule koje proizvede generator uslova se negiraju, prevode u SMT-LIB format i prosleđuju dokazivaču. Kako je tvrđenje ispravnosti formula oblika $\forall * F$, dokazivaču se šalje formula oblika $\exists * \neg F$. Ukoliko dokazivač utvrdi nezadovoljivost ove, negirane formule, to potvrđuje valjanost originalnog tvrđenja čime se dobija informacija o ispravnosti/neispravnosti komande.

Ukoliko nije moguće dokazati bezbednost komande, dokazivač može da generiše model na osnovu kojeg je, u nekim situacijama, moguće konstruisati konkretan primer prekoračenja bafera. Ta informacija može značajno da olakša pronalaženje greške u kodu i stoga je veoma važna korisniku.

Tvrđenja ispravnosti/neispravnosti mogu da se testiraju bilo kojim SMT dokazivačem koji pokriva linearnu aritmetiku. Dokazivač teorema, koji pokriva linearnu aritmetiku, je podesan za proveru važenja generisanih tvrđenja jer većina uslova ispravnosti/neispravnosti komandi pripada linearnoj aritmetici. Naime, pokazivačka aritmetika se zasniva na sabiranju i oduzimanju, pa se može dobro modelovati linearnom aritmetikom.

Primer 5.5 *Negacija tvrđenja ispravnosti generisanog u primeru 5.4 je (podrazumeva se egzistencijalno zatvorenje):*

$$(0 \leq size\_src\_1) \wedge (size\_src\_1 = 200) \wedge \neg(size\_src\_1 \geq 200).$$

SMT-LIB *oblik ove formule je:*

```
(: formula
    ( and
        ( <= 0 size_src_1)
        ( = size_src_1 200)
        ( not ( >= size_src_1 200) )
    )
)
```

Dokazivač može da utvrdi nezadovoljivost ove formule pa je originalno tvrđenje valjana formula i ispunjen je uslov ispravnosti. Kako se, pored toga, ne može potvrditi uslov neispravnosti, to znači da je komanda `fgets` *iz primera 5.4 bezbedna.*

5.8 Prikaz rezultata

Za svaku komandu sistem pamti broj linije koda izvorne datoteke. Na osnovu toga, rezultat dokazivača se pridružuje odgovarajućoj liniji koda i prijavljuje se korisniku.

Komande, koje sistem obeleži kao *bezbedne*, ne dovode do greške, te se informacije o njima i ne prijavljuju korisniku. Komande koje sistem obeleži kao *neispravne*, dovešće do greške prilikom svakog izvršavanja programa i moraju biti izmenjene. Ove greške su često trivijalne i često se mogu otkriti običnim testiranjem programa. Komande, koje sistem obeleži kao *nebezbedne*, su moguća prekoračenja bafera i moraju biti proverene od strane programera. Ako je komanda *neispravna* ili *nebezbedna*, sistem najčešće može da ponudi programeru konkretan primer koji dovodi do prekoračenja bafera. Ovim se olakšava pronalaženje greške u kodu i njeno ispravljanje.

Komande, koje sistem obeleži kao *nedostupne*, predstavljaju grane programa do kojih, prilikom izvršavanja, nije moguće stići. Korisnik treba da odstrani iz koda te komande (a to neće uticati na semantiku programa) ili da izmeni uslove koji dovode do nedostupnosti ovih komandi.

5.9 Domen pristupa

Nije moguće napraviti sistem koji je potpun i saglasan, tj. sistem koji otkriva sva prekoračenja bafera, bez lažnih upozorenja. Jedan od razloga je neodlučivost problema zaustavljanja (eng. halting problem).

Predloženi sistem ima sledeća ograničenja:

- rad sa petljama je ograničen i to otežava otkrivanje prekoračenja bafera kako unutar petlje tako i nakon petlje;
- za računanje preduslova i postuslova korisnički definisanih funkcija, sistem zahteva pomoć korisnika;
- generisane pretpostavke pripadaju linearnoj aritmetici, što znači da se sve ostale teorije koje su umešane ne razmatraju (na primer, teorija nizova).

I pored prethodnih ograničenja, sistem može da otkrije veliki broj prekoračenja bafera.

Moć sistema određuje i sadržaj biblioteke uslova. Ona je spoljašnja i otvorena da bi sadržaj mogao da bude proširen od strane korisnika. Njen inicijalni sadržaj treba da pokriva operatore i funkcije iz standardne C biblioteke, ali i neke

specifične slogove koji mogu da prošire domašaj sistema (na primer, dodela `a[k]=0` ima postuslov $used(a, 1) \leq k + 1$).

Na moć sistema utiče i moć dokazivača koji se koristi. Može se, na primer, koristiti dokazivač za linearnu aritmetiku nad celim brojevima ili dokazivač za linearnu aritmetiku nad racionalnim brojevima. Kako je pokazivačka aritmetika celobrojna, prikladnije je koristiti dokazivač za linearnu aritmetiku nad celim brojevima. Međutim, dokazivači za linearnu aritmetiku nad racionalnim brojevima su znatno efikasniji. Ta efikasnost smanjuje moć sistema: sistem ostaje saglasan ali ne i potpun. To znači da su sva tvrđenja ispravnosti koja se dokažu zaista tačna, ali da postoje i tačna tvrđenja koja ne mogu biti dokazana, a koja bi mogla biti dokazana od strane dokazivača za linearnu aritmetiku nad celim brojevima. Na primer, ispravnost tvrđenja $x \geq y+1 \vee x \leq y$, gde su x i y celi brojevi, može se dokazati dokazivačem za linearnu aritmetiku nad celim brojevima, ali ne i dokazivačem za linearnu aritmetiku nad racionalnim brojevima. Pored ovoga, dokazivač za linearnu aritmetiku nad racionalnim brojevima, utiče na moć sistema i generisanim kontramodelima. U nekim situacijama, za tvrđenje koje nije tačno u linearnoj aritmetici nad racionalnim brojevima, dokazivač za ovu teoriju može da generiše kontramodel koji uključuje racionalne brojeve. Tako generisan kontramodel nije adekvatan ako u uslovu postoje promenljive koje su celobrojne. Na primer, tvrđenje $x > 5$ nije tačno ni u linearnoj aritmetici za racionalne ni u linearnoj aritmetici za cele brojeve, ali dokazivač za linearnu aritmetiku za racionalne brojeve može da ponudi kontramodel $x = 2.5$ koji nije adekvatan ako je promenljiva x celobrojna.

5.10 Primer

Razmotrimo fragment koda prikazan na slici 5.6 (levo). Navedeni kôd dodaje ekstenziju `.txt` na sadržaj niske `t` i upisuje rezultat u nisku `s`.

#	preduslov	komanda	postuslov
1		`char x[N]`	$size(x, 1) = value(N, 0)$
2		`x=strlen(y)`	$value(x, 1) = used(y, 0) - 1$
3	$size(x, 0) \leq used(y, 0)$	`strcpy(x,y)`	$used(x, 1) = used(y, 0)$
4		`x=malloc(y)`	$(value(y, 0) \neq 0$
			$\wedge sizeof(*x) \cdot size(x, 1) = value(y, 0))$
			$\vee value(x, 1) = 0$
5		`x=y`	$value(x, 1) = value(y, 0)$
6		`x=y`	$value(x, 1) = value(y, 0)$
			$\wedge size(x, 1) = size(y, 0)$
7		`x=y`	$value(x, 1) = value(y, 0)$
			$\wedge size(x, 1) = size(y, 0)$
			$\wedge used(x, 1) = used(y, 0)$

Slika 5.5: Slogovi biblioteke uslova

Pretpostavimo da biblioteka uslova sadrži slogove date na slici 5.5. Četvrti slog biblioteke uključuje informaciju o veličini promenljive korišćenjem funkcije $sizeof$. Ova funkcija je funkcija meta nivoa i vraća veličinu u bajtovima njenog argumenta, slično kao funkcija `sizeof` u programskom jeziku C. Ova veličina

```
char *s;

s = (char*)malloc(strlen(t)+4);

if(s!=NULL)
{
    strcpy(s,t);

    strcpy(s+strlen(s),".txt");
    return_value = 0;
}
else

    return_value = -1;
```

```
int _fado_1;
int _fado_2;
int _fado_3;
char *_fado_4;
char *_fado_5;
char *s;
_fado_1 = strlen(t);
_fado_2 = _fado_1+4;
s = malloc(_fado_2);
_fado_4 = s;
if(_fado_4!=NULL)
{
    strcpy(s,t);
    _fado_3 = strlen(s);
    _fado_5 = s+_fado_3;
    strcpy(_fado_5,".txt");
    return_value = 0;
}
if(_fado_4==NULL)
{
    return_value = -1;
}
```

Slika 5.6: C kôd (levo) i transformisan kôd (desno)

uvek može da se izračuna i vraća fiksnu celobrojnu vrednost pa sva pozivanja na ovu funkciju mogu biti razrešena u fazi evaluacije. Pored toga, slog biblioteke uslova može da ima i restrikciju tipa. Na primer, ograničenje za slog 5 je da promenljive `x` i `y` nisu pokazivači, ograničenje za slog 6 je da promenljive `x` i `y` jesu pokazivači, ali ne tipa `char*`, a ograničenje za slog 7 je da su promenljive `x` i `y` tipa `char*`.

Slika 5.6 (desno) prikazuje transformisani kôd. Slika 5.7 pokazuje uslove koji se konstruišu za svaku komandu pojedinačno. Slika 5.8 prikazuje uslove nakon evaluacije, ažuriranja stanja i konstruisanja uslova za `if` komande.

Da bi komanda `strcpy(s,t)` bila bezbedna, potrebno je da sledeća formula bude valjana:

$$
\begin{aligned}
&(value(\texttt{\_fado\_1}, 1) = used(t, 0) - 1 \wedge \\
&used(t, 0) > 0 \wedge \\
&value(\texttt{\_fado\_2}, 1) = value(\texttt{\_fado\_1}, 1) + 4 \wedge \\
&((value(\texttt{\_fado\_2}, 1) \neq 0 \wedge size(s, 1) = value(\texttt{\_fado\_2}, 1)) \vee value(s, 1) = 0) \wedge \\
&value(\texttt{\_fado\_4}, 1) = value(s, 1) \wedge \\
&value(\texttt{\_fado\_4}, 1) \neq 0 \wedge size(s, 1) \geq 0) \\
&\Rightarrow size(s, 1) \leq used(t, 0)
\end{aligned}
$$

Kako su u ovom tvrđenju svi konjunkti relevantni, korak eliminisanja irelevantnih konjunkata nema efekta. Nakon apstrahovanja, tvrđenje postaje:

preduslov	transformisani kôd	postuslov
	`int _fado_1;`	
	`int _fado_2;`	
	`int _fado_3;`	
	`char *_fado_4;`	
	`char *_fado_5;`	
	`char *s;`	
	`_fado_1 = strlen(t);`	$value(\texttt{\_fado\_1}, 1) = used(t, 0) - 1$
	`_fado_2 = _fado_1+4;`	$value(\texttt{\_fado\_2}, 1) =$ $value(\texttt{\_fado\_1}, 0) + value(4, 0)$
	`s = malloc(_fado_2);` —	$(value(\texttt{\_fado\_2}, 0) \neq 0 \wedge$ $sizeof(*s) \cdot size(s, 1) = value(\texttt{\_fado\_2}, 0))$ $\vee value(s, 1) = 0$
	`_fado_4 = s;`	$value(\texttt{\_fado\_4}, 1) = value(s, 0)$
	`if(_fado_4!=NULL)`	
	`{`	$value(\texttt{\_fado\_4}, 0) \neq 0$
$size(s, 0) \geq$ $used(t, 0)$	`strcpy(s,t);`	$used(s, 1) = used(t, 0)$
	`_fado_3 = strlen(s);`	$value(\texttt{\_fado\_3}, 1) = used(s, 0) - 1$
	`_fado_5 = s+_fado_3;`	$value(\texttt{\_fado\_5}, 1) =$ $value(s, 0) + value(\texttt{\_fado\_3}, 0)$
$size(\texttt{\_fado\_5}, 0) \geq$ $used(".txt", 0)$	`strcpy(_fado_5,".txt");`	$used(\texttt{\_fado\_5}, 1) = used(".txt", 0)$
	`return_value = 0;`	$value(return\_value, 1) = value(0, 0)$
	`}`	
	`if(_fado_4==NULL)`	
	`{`	$value(\texttt{\_fado\_4}, 0) = 0$
	`return_value = -1;`	$value(return\_value, 1) = value(-1, 0)$
	`}`	

Slika 5.7: Transformisan kôd i uslovi za pojedinačne komande

$$
\begin{aligned}
&(\texttt{\_fado\_1\_1} = used\_t\_0 - 1 \wedge \\
&used\_t\_0 > 0 \wedge \\
&\texttt{\_fado\_2\_1} = \texttt{\_fado\_1\_1} + 4 \wedge \\
&((\texttt{\_fado\_2\_1} \neq 0 \wedge size\_s\_1 = \texttt{\_fado\_2\_1}) \vee s\_1 = 0) \wedge \\
&\texttt{\_fado\_4\_1} = s\_1 \wedge \\
&\texttt{\_fado\_4\_1} \neq 0 \wedge \\
&size\_s\_1 \geq 0) \\
&\Rightarrow size\_s\_1 \leq used\_t\_0
\end{aligned}
$$

Tvrđenje se negira, konvertuje u SMT-LIB format i šalje dokazivaču teorema na proveru. Negirano tvrđenje je:

$$
\begin{aligned}
&\texttt{\_fado\_1\_1} = used\_t\_0 - 1 \wedge \\
&used\_t\_0 > 0 \wedge \\
&\texttt{\_fado\_2\_1} = \texttt{\_fado\_1\_1} + 4 \wedge \\
&((\texttt{\_fado\_2\_1} \neq 0 \wedge size\_s\_1 = \texttt{\_fado\_2\_1}) \vee s\_1 = 0) \wedge \\
&\texttt{\_fado\_4\_1} = s\_1 \wedge \\
&\texttt{\_fado\_4\_1} \neq 0 \wedge \\
&size\_s\_1 \geq 0 \wedge \\
&\neg(size\_s\_1 \leq used\_t\_0)
\end{aligned}
$$

SMT-LIB oblik negiranog tvrđenja je dat na slici 5.9.

preduslov	transformisani kôd	postuslov
	`int _fado_1;`	
	`int _fado_2;`	
	`int _fado_3;`	
	`char *_fado_4;`	
	`char *_fado_5;`	
	`char *s;`	
	`_fado_1 = strlen(t);`	$value(\texttt{\_fado\_1}, 1) = used(t, 0) - 1$
	`_fado_2 = _fado_1+4;`	$value(\texttt{\_fado\_2}, 1) = value(\texttt{\_fado\_1}, 1) + 4$
	`s = malloc(_fado_2);`	$(value(\texttt{\_fado\_2}, 1) \neq 0$ $\wedge size(s, 1) = value(\texttt{\_fado\_2}, 1))$ $\vee value(s, 1) = 0$
	`_fado_4 = s;`	$value(\texttt{\_fado\_4}, 1) = value(s, 1)$
	`if(_fado_4!=NULL)`	
	`{`	$value(\texttt{\_fado\_4}, 1) \neq 0$
$size(s, 1)$ $\geq used(t, 0)$	`strcpy(s,t);`	$used(s, 2) = used(t, 0)$
	`_fado_3 = strlen(s);`	$value(\texttt{\_fado\_3}, 1) = used(s, 2) - 1$
	`_fado_5 = s+_fado_3;`	$value(\texttt{\_fado\_5}, 1) =$ $value(s, 1) + value(\texttt{\_fado\_3}, 1)$
$size(\texttt{\_fado\_5}, 1)$ ≥ 5	`strcpy(_fado_5,".txt");`	$used(\texttt{\_fado\_5}, 2) = 5$
	`return_value = 0;`	$value(return\_value, 1) = 0$
	`}`	$value(\texttt{\_fado\_4}, 1) \neq 0$ $\wedge used(s, 2) = used(t, 0)$ $\wedge value(\texttt{\_fado\_3}, 1) = used(s, 2) - 1$ $\wedge value(\texttt{\_fado\_5}, 1) =$ $value(s, 1) + value(\texttt{\_fado\_3}, 1)$ $\wedge used(\texttt{\_fado\_5}, 2) = 5$ $\wedge value(return\_value, 1) = 0)$ $\vee(\neg(value(\texttt{\_fado\_4}, 1) \neq 0)$ $\wedge used(s, 2) = used(s, 1)$ $\wedge value(\texttt{\_fado\_3}, 1) = used(\texttt{\_fado\_3}, 0))$ $\wedge value(\texttt{\_fado\_5}, 1) = value(\texttt{\_fado\_5}, 0)$ $\wedge used(\texttt{\_fado\_5}, 2) = used(\texttt{\_fado\_5}, 1)$ $\wedge value(return\_value, 1) =$ $value(return\_value, 0)$
	`if(_fado_4==NULL)`	
	`{`	$value(\texttt{\_fado\_4}, 1) = 0$
	`return_value = -1;`	$value(return\_value, 1) = -1$
	`}`	$(value(\texttt{\_fado\_4}, 1) = 0$ $\wedge value(return\_value, 1) = -1))$ $\vee(\neg(value(\texttt{\_fado\_4}, 1) = 0) \wedge$ $value(return\_value, 1) =$ $value(return\_value, 0))$

Slika 5.8: Transformisan kôd i uslovi nakon evaluacije

Dokazivač potvrđuje nezadovoljivost negiranog tvrđenja ispravnosti, pa je originalno tvrđenje valjano. Pored toga, kako se ne može dokazati valjanost tvrđenja neispravnosti, to znači da je poziv funkcije `strcpy(s,t)` bezbedan.

Tvrđenja za komandu `strcpy(_fado_5,".txt")` se generišu na sličan način, ali ova komanda (a, time, i odgovarajuća komanda iz originalnog programa — `strcpy(s+strlen(s), ".txt")` dobija status neispravne komande. Ukoliko bi se komanda `s=(char*)malloc(strlen(t)+4)`, u polaznom kodu, zamenila komandom `s = (char*)malloc (strlen(t)+5)`, tada bi komanda `strcpy (_fado_5,".txt")` (tj. odgovarajuća komanda `strcpy(s+strlen(s), ".txt")`) bila bezbedna. Ovaj primer ilustruje veoma čestu grešku koja se javlja kada se pri rezervisanju prostora za nisku ne predvidi i prostor za karakter '`\0`'.

```
(: formula
    ( and
        ( =
            _fado_1_1
            ( - used_t_0 1 )
        )
        ( > used_t_0 0 )
        ( =
            _fado_2 1
            ( + _fado_1_1 4 )
        )
        ( or
            ( and
                ( not
                    ( = _fado_2_1 0 )
                )
                ( = size_s_1 fado_2 1 )
            )
            ( = s_1 0 )
        )
        ( = _fado_4_1 s_1 )
        ( not
            ( = _fado_4_1 0 )
        )
        (>= size_s_1 0 )
        (not
            ( <= size_s_1 used_t_0 )
        )
    )
)
```

Slika 5.9: SMT-LIB oblik tvrđenja ispravnosti komande `strcpy(s,t)`

6

Alat FADO

Alat za statičko otkrivanje prekoračenja bafera, FADO[1], je prototip sistema koji je predstavljen u glavi 5. Alat je implementiram u programskom jeziku C++, sastoji se od oko 13 000 linija koda raspoređenih u 35 klasa. Od drugih sistema alat FADO koristi samo parser za jezik C i automatski dokazivač teorema za linearnu aritmetiku. Alat FADO je implementiran poštujući standardni jezik C++, te se može kompilirati i izvršavati na raznim platformama (na primer, Windows i Linux). U nastavku će biti opisani i eksperimentalni rezultati primene i testiranja alata FADO na operativnom sistemu Windows. Ovi rezultati govore da je alat FADO po raznim kriterijumima uporediv ili bolji od nekih značajnih alata iste namene.

6.1 Opis implementacije

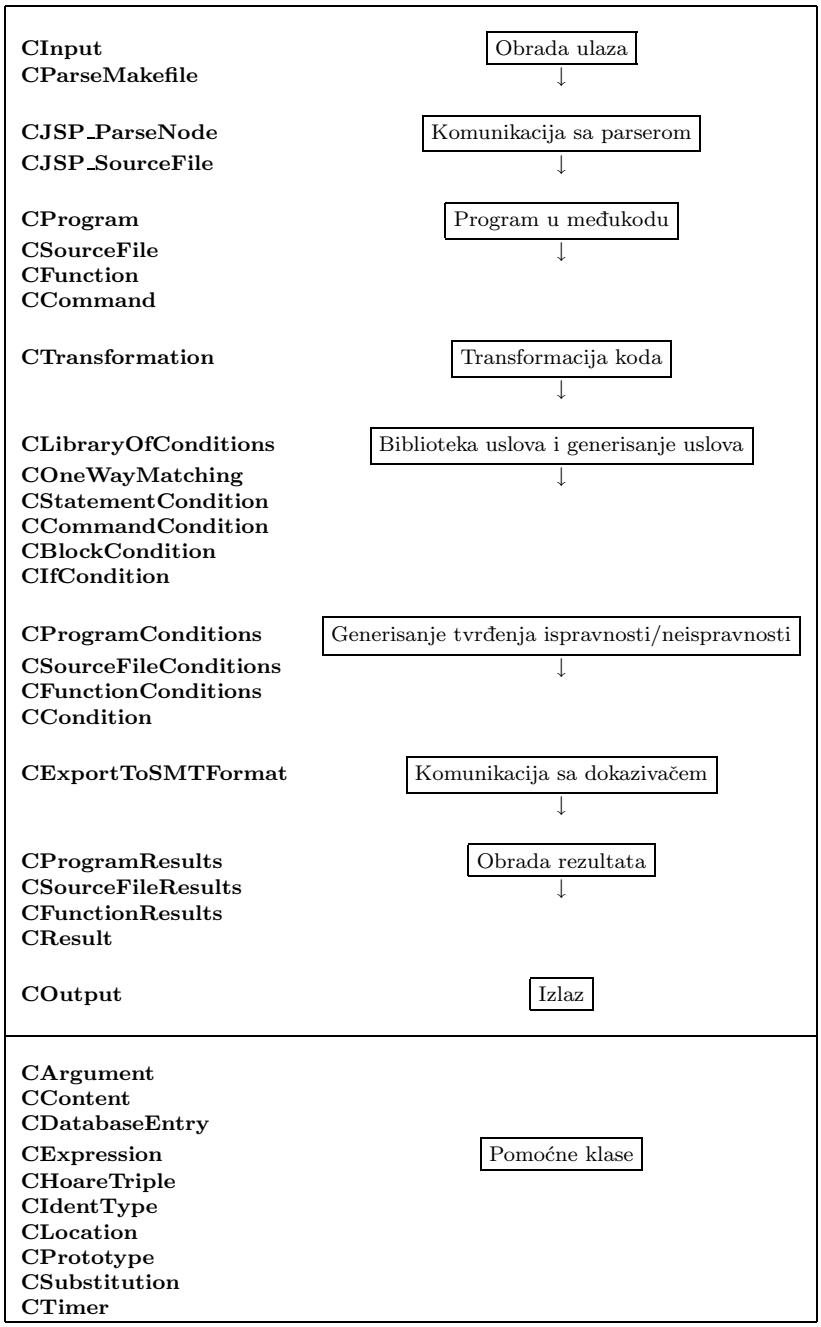
Arhitektura alata FADO prati ideje o opštoj arhitekturi sistema za statičku analizu iz poglavlja 5.1. Modularnost implementacije čini da je sistem veoma fleksibilan i da se različite komponente mogu jednostavno zameniti alternativnim. Implementirane klase sa svojim mestima u arhitekturi alata prikazane su na slici 6.1. U narednim poglavljima biće opisane pojedine komponente alata FADO, tj. odgovarajuće faze obrade C programa, kao i klase zadužene za te obrade.

Spoljašnji sistemi

Alat FADO koristi dva spoljašnja sistema: parser C programa i automatski dokazivač teorema za razrešavanje tvrđenja ispravnosti komandi. Trenutna verzija alata koristi parser `JSCPP` i on, relativno jednostavno, može biti zamenjen drugim parserom. Podrazumevani dokazivač je ARGO-LIB, a može se koristiti bilo koji dokazivač koji podržava SMT-LIB standarde.

`JSCPP` parser je deo kompleksnog sistema (implementiranog u ANSI C-u) za obradu C programa, koji razvija Jörg Schön (iz Nemačke). Sistem omogućava

[1] *Fado* je skraćenica za *F*lexible *A*utomated *D*etection of buffer *O*verflows.



Slika 6.1: Klase koje u učestvuju u implementaciji sistema

preprocesiranje i parsiranje koda. Takođe, sistem može da vrši analizu međuzavisnosti funkcija, da prati pristupe i izmene promenljivih, kao i da pomogne u izdvajanju novih funkcija iz dugih i kompleksnih blokova koda. Sistem je javno dostupan na Internet-adresi:
http://www.die-schoens.de/prg/index.html

ARGO-LIB je opštenamenska biblioteka za korišćenje procedura odlučivanja, koju razvija Filip Marić (sa Matematičkog fakulteta Univerziteta u Beogradu). Po pitanju osnovne logike, teorija koje podržava, opisa teorija i formata, ARGO-LIB se strogo pridržava standarda SMT-LIB inicijative [65]. Dokazivač koji FADO koristi zasnovan je na varijanti simpleks metode koja je opisana u radu [25]. Biblioteka ARGO-LIB je dostupna na Internet-adresi
http://argo.matf.bg.ac.yu.

Obrada ulaza, parsiranje i transformisanje koda

Klasa `CInput` obrađuje podatke koje korisnik unosi prilikom pokretanja programa. Klasa `CParseMakefile` obrađuje ulaznu datoteku i izdvaja imena izvornih C datoteka koje alat treba da obradi.

U početnoj fazi obrade, alat parsira izvorne C datoteke koristeći parser `JSCPP`. Klase `CJSP_ParseNode` i `CJSP_SourceFile` predstavljaju C++ omotač (eng. wrapper) parsera. Ovaj omotač prevodi specifičnu rekurzivnu strukturu stabla parsiranja koji proizvodi `JSCPP` u međukôd nad kojim se vrši dalja obrada. Na taj način je komunikacija sa parserom izolovana u ove dve klase te je, po potrebi, parser moguće jednostavno zameniti nekim drugim parserom (na primer, efikasnijim parserom ili parserom koji je integrisan u razvojno okruženje). Za reprezentovanje međukoda koriste se klase `CProgram`, `CSourceFile`, `CFunction` i `CCommand`.

Međukôd se transformiše u klasi `CTransformation`. Ova klasa implementira složen algoritam transformacije koda o kojem govori poglavlje 5.4. Implementacijom su obuhvaćeni sledeći koraci:

1. Složene deklaracije i inicijalizacije u okviru deklaracija transformišu se u proste deklaracije i inicijalizacije van deklaracija. Na primer, deklaracija

   ```
   int a, b, c=5;
   ```

   se transformiše u tri deklaracije i dodelu

   ```
   int a; int b; int c; c=5;
   ```

2. Sve petlje prevode se u `do-while` petlje. Na primer, petlja

   ```
   for(i=0; i<n; i++) {...}
   ```

   se prevodi u sledeći segment koda:

   ```
   i=0;
   if(i<n)
      do
      {
   ```

```
    ...
    i++;
}
while(i<n);
```

3. Složeni izrazi dodele `+=, -=, *=, /=, %=, <<=, >>=, ~=, |=, &=, ^=` prevode se u obične izraze dodele. Na primer, `a+=(b*c)` se prevodi u `a=a+(b*c)`.

4. Eliminišu se operatori `++` i `--` u skladu sa svojim pozicijama u izrazu. U okviru ove faze može da dođe do uvođenja novih promenljivih u program.

5. Pristupi višedimenzionim nizovima prevode se u odgovarajuće pristupe jednodimenzionom nizu. Na primer, za dvodimenzioni niz `int a[3][5];` pristup `a[1][1]` se prevodi u `a[6]`.

6. Svi složeni aritmetički izrazi se prevode u *proste aritmetičke izraze*. Prost aritmetički izraz može da ima jedan od sledećih oblika:

   - ϕ, pri čemu je ϕ *prost term* — promenljiva, konstanta, dereferencirani pokazivač ili element niza čiji je indeks promenljiva ili konstanta;
   - $\phi = \psi$ pri čemu je
     - ϕ prost term, ali ne konstanta;
     - ψ prost term;
   - $\phi = \psi \star \xi$ pri čemu je
     - ϕ je prost term, ali ne konstanta ili derefencirani pokazivač,
     - ψ je prost term, ali ne derefencirani pokazivač,
     - ξ je prost term, ali ne derefencirani pokazivač,
     - $\star$ je neki od operatora `+, -, *, /, %, << >> | & ~ ^`;
   - $\phi = f(a_1, a_2, ..., a_n)$ pri čemu je
     - ϕ promenljiva,
     - f funkcija standardne C biblioteke ili korisnički definisana funkcija,
     - a_1, a_2, ... a_n su promenljive ili konstante.

   U okviru ove faze može da dođe do uvođenja novih promenljivih u program.

7. Svi složeni logički izrazi se eliminišu prevođenjem u odgovarajuće `if` komande. Na primer, komanda

   ```
   if (a && b)
       {...}
   ```

   se prevodi u niz komandi

```
if (a)
  if (b)
      {...}
```

U okviru ove faze može da dođe do uvođenja novih promenljivih u program.

8. Komande `if-else` se transformišu u `if` komande bez dela `else`, pri čemu se uslov komande `if` transformiše ili u promenljivu ili u poređenje dve promenljive, promenljive i konstante ili dve konstante. Na primer, komanda

```
if (a>(b+c))
  {...}
else
  {...}
```

se transformiše u niz komandi

```
fado_1 = b + c;
if (a>fado_1)
    {...}
if (a<=fado_1)
    {...}
```

Trenutna verzija klase `CTransformation` ne podržava transformisanje koda koji sadrži nestrukturne forme programiranja (kao što su naredbe skoka `goto`, `jmp` i slično), unije i strukture, kao ni transformisanje naredbe višestrukog grananja (`switch`). Ova transformisanja će biti podržana u narednoj verziji alata.

Biblioteka uslova i generisanje uslova

Za generisanje uslova ispravnosti komandi koristi se apstraktna klasa `CStatementCondition` koja predstavlja interfejs za njene klase naslednice `CBlockCondition`, `CCommandCondition` i `CIfCondition`. Ove klase, na osnovu biblioteke uslova, generišu preduslove i postuslove pojedinačnih naredbi, blokova naredbi i naredbi `if`, kao što je to opisano u poglavlju 5.5.

Biblioteka uslova je spoljašnja tekstualna datoteka koja sadrži niz trojki oblika `(preduslov, komanda, postuslov)`. Elementi ovih trojki zapisani su u datoteci kao izrazi u prefiksnom obliku:

```
dominantni_simbol [0|1] [arnost] argumenti
```

pri čemu važi:

- `dominantni_simbol` može da bude ključna reč:

  `ident` — označava proizvoljnu promenljivu i izraz se zapisuje u formi `ident ime_identifikatora`;

  `constant` — označava proizvoljnu konstantu i izraz se zapisuje u formi `constant c`, gde je `c` znakovna ili brojevna konstanta;

`pointer` — označava sadržaj memorijske lokacije na koju ukazuje navedena promenljiva; na primer, `pointer 0 1 ident p`, ima značenje C izraza `*p`;

`buffer` — označava pristup elementu niza, pri čemu se podrazumeva da je prvi argument ovog izraza ime niza, dok su ostali argumenti ovog izraza indeksi niza; na primer,
`buffer 0 2 ident a constant 5`
ima značenje C izraza `a[5]`;

`_value`, `_size` **ili** `_used` — označavaju funkcije opisane u poglavlju 5.3; izraz u ovom slučaju kao prvi argument ima promenljivu ili konstantu, a kao drugi argument ima vremenski trenutak (stanje) `time 0` ili `time 1`; na primer, `_value 0 2 ident x time 0` ima značenje vrednosti promenljive `x` u trenutku `0`;

`FunctionCall` — označava izraz koji predstavlja poziv funkcije; prvi argument ovog izraza je znakovna konstanta — ime funkcije; na primer,
`FunctionCall 0 2 constant strlen ident buf`
ima značenje poziva `strlen(buf)`;

`and`, `or` **ili** `not` — označavaju logičke veznike `i`, `ili` i `negacija`;

`+`, `-`, `*`, `/`, `%`, <<, >>, |, , &, >, >=, <, <= **ili** = — označavaju operatore programskog jezika C.

- `[0|1]` — označava da li se dominantni simbol ispisuje u prefiksnom (0) ili infiksnom (1) obliku; ovaj argument se izostavlja samo za izraze koji opisuju promenljive, konstante i vremenske trenutke;
- `[arnost]` — označava arnost izraza; ovaj argument se izostavlja samo za izraze koji opisuju promenljive, konstante i vremenske trenutke;
- `argumenti` — označava listu argumenata; u opštem slučaju argumetni mogu da budu proizvoljni izrazi; broj argumenata je određen arnošću izraza.

Ukoliko komanda nema preduslov (ili postuslov), to se obeležava ključnom rečju `_empty`. Komentari se zapisuju nakon znaka `//` (kao u programskom jeziku `C++`). Primer jednog sloga biblioteke uslova dat je na slici 6.2.

Biblioteka inicijalno pokriva osnovne funkcije za rad sa niskama iz standardne C biblioteke `string.h`, osnovne funkcije za rad sa dinamičkom alokacijom memorije iz standardne C biblioteke `malloc.h` i osnovne funkcije za rad sa datotekama iz standardne C biblioteke `stdio.h`. Pored navedenih funkcija, biblioteka sadrži i neke specifične slogove koji proširuju domen alata (na primer, slogovi koji određuju preduslove i postuslove za komande `a[i] = x` i `*p = k`).

Za učitavanje sadržaja biblioteke i za obradu slogova biblioteke koristi se klasa `CLibraryOfConditions` i pomoćne klase `CDatabaseEntry` i `CHoareTriple`. Za pronalaženje odgovarajućeg sloga biblioteke koji se uparuje sa komandom programa koji se obrađuje, koristi se klasa `COneWayMatching` i pomoćna klasa `CSubstitution`. Klasa `COneWayMatching` implementira algoritam jednosmernog uparivanja koji je opisan u [7].

```
//////////////////////////////////////////
//strcpy(dst,src)
//////////////////////////////////////////
//
//KOMANDA:
//strcpy(dst,src)
//
FunctionCall 0 3
    constant strcpy
        ident dst
        ident src
//
//PREDUSLOV:
//size(dst,0)>=used(src,0)
//
>= 1 2
    _size 0 2
        ident dst
        time 0
    _used 0 2
        ident src
        time 0
//
//POSTUSLOV:
//used(dst,1)=used(src,0) and used(src,0)>0
//
and 1 2
    = 1 2
        _used 0 2
            ident dst
            time 1
        _used 0 2
            ident src
            time 0
    > 1 2
        _used 0 2
            ident src
            time 0
        constant 0
//////////////////////////////////////////
//
//////////////////////////////////////////
```

Slika 6.2: Primer sloga biblioteke uslova za funkciju `strcpy`

Generisanje tvrđenja i pozivanje dokazivača

Na osnovu uslova generisanih u prethodnoj fazi obrade, klase `CProgramConditions`, `CSourceFileConditions`, `CFunctionConditions` generišu tvrđenja ispravnosti/neispravnosti komandi na način opisan u poglavlju 5.6. Sama tvrđenja se (interno) skladište pomoću klase `CCondition`.

Tvrđenja ispravnosti/neispravnosti komandi se, pomoću klase `CExportToSMTFormat`, negiraju i eksportuju u spoljašnju datoteku u SMT-LIB formatu, na način opisan u poglavlju 5.7. Poziva se dokazivač ARGO-LIB, on učitava formulu, ispituje njenu nezadovoljivost/zadovoljivost i rezultat upisuje u drugu spoljašnju datoteku. Ove dve spoljašnje datoteke čine jedinu komunikaciju alata FADO i automatskog dokazivača teorema ARGO-LIB. Na taj način se tvrđenja mogu obrađivati i proizvoljnim drugim dokazivačem koji podržava SMT-LIB standarde. Alat FADO bi se mogao učiniti efikasnijim ukoliko bi se, umesto komunikacije

preko spoljašjnih datoteka, obezbedila direktna komunikacija sa dokazivačem. Međutim, na taj način bi alat izgubio na fleksibilnosti.

Formula koja se prosleđuje dokazivaču može da bude i sintaksno neispravna iz perspektive dokazivača, usled prisustva operatora koji ne pripadaju linearnoj aritmetici (na primer, operatora deljenja). U tom slučaju, dokazivač signalizira sintaksnu grešku u okviru formule. Ta greška ukazuje na to da je tvrđenje van domena dokazivača, a samim tim i alata FADO.

Obrada rezultata i izlaz

Na osnovu rezultata rada dokazivača izdvajaju se komande koje nisu bezbedne. U zavisnosti od opcija koje su bile pristune prilikom pokretanja programa, generišu se i ostale tražene informacije. Obradu rezultata vrše klase `CProgramResults`, `CSourceFileResults`, `CFunctionResults` i `CResult`. Rezultat se upisuje u tekstualnu i/ili HTML datoteku. Podatke o generisanom izlazu, kao što je to opisano u poglavlju 6.2, na standardni izlaz ispisuje klasa `COutput`. Primer izlaznih rezultata dat je u poglavlju 6.2.

6.2 Upotreba alata FADO

Alat FADO je jednostavan za korišćenje. Program se pokreće iz komandne linije i rezultate analize upisuje u tekstualnu i/ili HTML datoteku. Generisani izlaz sadrži pregledno organizovane rezultate izvršene analize.

Pokretanje alata

Program FADO pokreće se iz komandne linije, na sledeći način:

```
> fado ime_datoteke [opcije] [biblioteka] [dokazivac]
```

pri čemu

- `ime_datoteke` označava ime datoteke u kojoj se nalazi izvorni kôd koji program treba da analizira ili ime datoteke u kojoj se nalazi spisak datoteka koje program treba da analizira;
- `[opcije]` označava opcije koje se mogu uključiti:
  - `-t` za generisanje tekstualnog izlaza (podrazumevana vrsta izlaza);
  - `-h` za generisanje HTML izlaza;
  - `-m` za generisanje modela koji dovodi do prekoračenja bafera (model sadrži vrednosti promenljivih u terminima funkcija *value*, *size* i *used*, opisanih u poglavlju 5.3);
  - `-c` za ispisivanje, u okviru tekstualnog i/ili HTML izlaza, komandi koje nisu bezbedne (pored njihovih rednih brojeva linija);
  - `-l` za korišćenje korisnikove biblioteke uslova; ukoliko se ova opcija ne navede koristi se podrazumevana biblioteka uslova `Library.txt`;

`-p` za korišćenje alternativnog dokazivača; ukoliko se ova opcija ne navede koristi se podrazumevani dokazivač `ArgoLib`;

- `[biblioteka]` označava ime biblioteke uslova i relevantno je samo ukoliko je prisutna opcija `-l`;
- `[dokazivac]` označava ime dokazivača i relevantno je samo ukoliko je prisutna opcija `-p`.

Redosled navođenja opcija je proizvoljan. Ukoliko su prisutne opcije `-l` i `-p`, podrazumeva se da se prvo navodi ime biblioteke uslova, a zatim ime dokazivača. Ukoliko se program neispravno pokrene, ispisuje se uputstvo za ispravno pokretanje.

Na primer, program se može pokrenuti na sledeći način:

```
> fado hello.c -h -t -m -c
```

U ovom slučaju analizira se izvorna daoteteka `hello.c`, generiše se HTML (opcija `-h`) i tekstualni (opcija `-t`) izlaz, generišu se modeli nebezbednih i pogrešnih komandi (opcija `-m`) i ispisuju se, u okviru izlaza, komande koje nisu bezbedne (opcija `-c`). Program koristi podrazumevanu biblioteku uslova i podrazumevan dokazivač. Ako se program pokrene na sledeći način:

```
> fado Makefile -m -l -p NewLibrary.txt Prover.exe
```

analizira se skup datoteka koje su navedene u okviru datoteke `Makefile`, generiše se samo tekstualni izlaz (jer nije navedna nijedna opcija vezana za vrstu izlaza), generišu se modeli nebezbednih i pogrešnih komandi (opcija `-m`), alat koristi spoljašnju biblioteku uslova (opcija `-l`) koja se zove `NewLibrary.txt` i dokazivač (opcija `-p`) sa imenom `Prover.exe`.

Rezultati rada programa

Nakon kompletne obrade ulaznih datoteka, alat FADO ispisuje na standardni izlaz poruku koja sadrži:

- podatke o spoljašnjim komponentama koje je alat koristio (uključujući parser, biblioteku uslova i dokazivač);
- podatke o datotekama sa rezultatima analize koje je alat generisao;
- ukupno vreme izvršanja programa, kao i vremena izvršavanja pojedinih komponenti alata.

U zavisnosti od prisutnih opcija, prilikom pokretanja programa generiše se tekstualni i/ili HTML izlaz. U oba slučaja, rezultujuća datoteka sadrži:

Kratak opis namene alata — on sadrži opis alata i objašnjenje klasifikacije bezbednosnih statusa komandi (komanda za koju nije utvrđeno da je bezbedna može biti pogrešna, nebezbedna, nedostižna ili van domena alata).

Zbirne rezultate analize — oni uključuju:

- broj obrađenih datoteka;
- ukupno vreme izvršavanja programa i vremena izvršavanja pojedinih faza obrade, tj.:
  - vreme utrošeno za parsiranje,
  - vreme utrošeno za transformisanje,
  - vreme utrošeno za generisanja uslova,
  - vreme utrošeno za eksportovanje formula i provere uslova;
- ukupan broj otkrivenih komandi koje nisu bezbedne, tj.:
  - ukupan broj pogrešnih komandi,
  - ukupan broj nebezbednih komandi,
  - ukupan broj nedostižnih komandi,
  - ukupan broj komandi čije je utvrđivanje ispravnosti van domena alata.

Detaljne rezultate analize — oni uključuju spisak koji, za svaku analiziranu datoteku u kojoj je pronađena komanda koja nije bezbedna, sadrži:

- ime datoteke u kojoj je pronađena komanda koja nije bezbedna (za HTML izlaz — i link na odgovarajuću datoteku);
- ime funkcije u kojoj je pronađena komanda koja nije bezbedna;
- spisak linija i statusa svih komandi koje nisu bezbedne.

Detaljni rezultati analize, u zavisnosti od opcija koje su bile prisutne prilikom pokretanja programa, mogu da sadrže i:

- pored linije i statusa komande, sâmu komandu koja nije bezbedna (ukoliko je prilikom pokretanja programa bila prisutna opcija `-c`);
- ukoliko je komanda bezbedna ili pogrešna, ime generisane datoteke (za HTML izlaz — i link na datoteku) u kojoj se nalazi model za datu komandu (ukoliko je prilikom pokretanja programa bila prisutna opcija `-m`).

Deo jedne HTML izlazne datoteke dat je, kao primer, na slici 6.3.

6.3 Eksperimentalni rezultati

Za testiranje alata FADO korišćen je skup test primera opisan u radu [47] i dostupan na Internet-adresi: `http://www.ll.mit.edu/IST/corpora.html`. Ovaj skup test primera služio je kao osnova za poređenje alata BOON, Splint, UNO, ARCHER i PolySpace C Verifier (rezultati ovog poređenja su ukratko opisani u poglavlju 4.1.3).

Slika 6.3: Deo jedne HTML izlazne datoteke

Skup test primera obuhvata 291 program u po četiri verzije. Prva verzija sadrži ispravan kôd, dok preostale tri sadrže, redom, minimalno (za 1 element), srednje (za 8 elemenata) i veliko (za 4096 elemenata) prekoračenje granica bafera.

Alat FADO detektuje prekoračenje bafera bez obzira na veličinu prekoračenja, pa su za svaki program iz skupa test primera razmatrane ispravna verzija i samo jedna neispravna (rezultat rada je isti za svaku neispravnu verziju). Dakle, razmatrano je $291 \cdot 2 = 582$ test primera. U ovim test primerima ima ukupno 291 prekoračenje bafera.

Na test primerima, alat FADO je, bez dodavanja preduslova i postuslova korisnički definisanih funkcija u biblioteku uslova, uspešno detektovao približno

56% grešaka. Preostala prekoračenja alat nije detektovao i njih čine:

- 35.7% prekoračenja koja se nalaze u petljama koje alat nije u mogućnosti da obradi, a koje bi se mogle obraditi dodavanjem heuristika za rad sa petljama;
- 5.5% prekoračenja bafera koji su članovi unija i struktura (one nisu podržane trenutnom implementacijom alata);
- 1.4% prekoračenja koja su u primerima koji izlaze iz okvira standardnog jezika C;
- 1.4% prekoračenja koja su suštinski van domena alata (to su, na primer, prekoračenja bafera čiji indeks uključuje operator ostatka pri deljenju — on se ne može modelovati linearnom aritmetikom nad realnim brojevima).

Za ovaj skup test primera, procenat lažnih upozorenja je 6.5%, a stepen konfuzije 12.5%. Ukoliko se u biblioteku uslova dodaju preduslovi i postuslovi korisnički definisanih funkcija, procenat lažnih upozorenja se smanjuje na 3%, a stepen konfuzije na 6%.

Za obradu 291 test primera, alat utroši 46.8 sekundi na PC računaru sa 2.4GHz i 768MB RAM memorije. Prosečno vreme uzvršavanja je 0.16 sekundi po test primeru. Procenti utrošenog vremena po fazama obrade su:

- 1.2% za parsiranje koda;
- 0.5% za transformisanje koda;
- 51.8% za generisanje tvrđenja ispravnosti/neispravnosti komandi;
- 46.4% za eksportovanje tvrđenja u SMT oblik i proveru važenja tvrđenja;
- 0.1% za obradu i ispis rezultata.

Alat	Procenat pronađenih prekoračenja	Procenat lažnih upozorenja	Stepen konfuzije
PolySpace	99.7	2.4	2.4
ARCHER	90.7	0.0	0.0
Splint	56.4	12.0	21.3
FADO	56.0	6.5	12.5
UNO	51.9	0.0	0.0
BOON	0.7	0.0	0.0

Tabela 6.1: Rezultati alata FADO i rezultati poređenja alata ARCHER, BOON, Splint, UNO i PolySpace C Verifier na osvnovu rada [47]

Poređenje sa drugim sistemima

Na osnovu navedenih rezultata za alat FADO i rezultata iz rada [47] trenutna verzija alata FADO je, po broju detektovanih grešaka, uspešnija od alata BOON i UNO, a manje uspešna od alata Archer i PolySpace. Ova verzija alata je uporediva je sa alatom Splint, pri čemu, u odnosu njega, ima značajno manji procenat lažnih upozorenja i manji stepen konfuzije. Rezultati su sažeti u tabeli 6.1

Vreme izvršavanja alata FADO za test primere opisane u radu [47] je veće od vremena izvršavanja alata BOON, UNO i Splint, ali manje od vremena izvršavanja alata ARCHER i PolySpace C Verifier. Prosečna vremena izvršavanja za ove test primere sažeta su u tabeli 6.2.

Alat	prosečno vreme izvršavanja
PolySpace	172.53s
ARCHER	0.25s
FADO	0.16s
Splint	0.02s
UNO	0.02s
BOON	0.06s

Tabela 6.2: Prosečno vreme izvršavanja po test primeru za alat FADO i prosečna vremena izvršavanja po test primeru za alate ARCHER, BOON, Splint, UNO i PolySpace C Verifier na osnovu rezultata iz rada [47]

7

Zaključci i dalji rad

Prekoračenje bafera nastaje usled propusta u programu koji omogućava upisivanje sadržaja van granica rezervisane memorije bafera. Prekoračenje bafera može da dovede do neočekivanog toka izvršavanja programa, kao i da omogući razne vrste zloupotreba programa. Ovaj problem je posebno rasprostranjen u programima napisanim na programskom jeziku `C`. Najznačajnije tehnike za automatsko detektovanje prekoračenja bafera dele se na statičke i dinamičke tehnike. Statičke tehnike imaju za cilj otkrivanje prekoračenja bafera analizom izvornog koda (opisane su u poglavlju 4.1), dok dinamičke tehnike analiziraju program u fazi izvršavanja (opisane su u poglavlju 4.2).

U ovom radu opisan je (u poglavlju 5) nov sistem za statičko automatsko detektovanje prekoračenja bafera. Opisana je i inicijalna implementacija predloženog sistema — alat FADO (u poglavlju 6).

Predložen sistem vrši statičku analizu programa, uzima u obzir kontrolu toka programa i kontekst poziva funkcija. Sistem najpre vrši transformaciju ulaznog koda u sintaksno jednostavniji oblik koji je semantički ekvivalentan početnom kodu. Sistem zatim, uz pomoć spoljašnje biblioteke uslova, generiše uslove ispravnosti i uslove neispravnosti za svaku komandu programa koji se analizira. Generisani uslovi se šalju na proveru automatskom dokazivaču teorema. Na osnovu rezultata dokazivača, komande se obeležavaju kao bezbedne, nebezbedne ili komande koje uvek dovode do prekoračenja bafera. Ako se za neku komandu pokaže da može da dovede do prekoračenja bafera, sistem generiše primer koji pomaže programeru da razume prirodu greške i da je ispravi.

Ključne novine u opisanom sistemu su:

- fleksibilna i modularna arhitektura koja omogućava izmene komponenti sistema i jednostavnu komunikaciju sa različitim spoljašnjim sistemima;
- uslovi ispravnosti i neispravnosti komandi dati u terminima Horove logike, sa jasnim logičkim smislom;
- logika generisanja uslova ispravnosti i neispravnosti komandi je data u spoljašnjoj biblioteci, koja se može jednostavno proširiti, a time i uvećati

mogućnosti sistema;

- korišćenje spoljašnjeg dokazivača teorema, koji može da obezbedi i formalni dokaz korektnosti;
- ukoliko je neka komanda nebezbedna, sistem u određenim slučajevima može da ponudi primer toka programa koji dovodi do prekoračenja bafera.

Prototip implementacija predloženog sistema, alat FADO je testiran na primerima na kojima su testirani poznati alati za statičku analizu prekoračenja bafera i dao je dobre rezultate.

Neka od mogućih unapređenja predloženog sistema u budućem radu su:

- proširivanje mehanizama za rad sa petljama kako bi sistem postao saglasan;
- proširivanje mehanizama za rad sa korisnički definisanim funkcijama kako bi sistem postao u potpunosti automatizovan;
- povezivanje sistema sa dokazivačima teorema koji imaju širi domen;
- proširivanje mogućnosti sistema tako da može da detektuje i druge vrste grešaka (na primer, curenje memorije).

Literatura

[1] AlephOne. Smashing the stack for fun and profit. *Phrack Magazine*, 7(47), 1998.

[2] Internet software consortium - bind. October 2003. on-line at: http://www.isc.org/products/BIND.

[3] A. Baratloo, N. Singh, and T. Tsai. Libsafe: Protecting critical elements of stacks. In *White Paper*, December 1999. http://www.research.avayalabs.com/project/libsafe/.

[4] A. Baratloo, T. Tsai, and N. Singh. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[5] H. T. Brugge. Bounds checking patch for gcc. on-line at: http://web.inter.nl.net/hcc/Haj.Ten.Brugge/.

[6] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack Magazine*, May 2000. http://www.phrack.org/phrack.56/p56-0x05.

[7] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[8] W. R. Bush, J.D. Pincus, and D.J. Sielaff. A static analyzer for finding a dynamic programming errors. *Software, Practice and Experience*, 30(7):775 – 802, June 2000.

[9] M. Conover and w00w00 Security Tem. w00w00 on heap overflows. January 1999. http://www.w00w00.org/files/articles/heaptut.txt.

[10] Microsoft Corporation. Ast toolkit. http://research.microsoft.com/sbt/.

[11] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106 – 130, Dunod, Paris, France, 1976.

[12] P. Cousot and N. Halbwach. Automatic discovery of linear constraints among variables of a program. In *Symposium on Principles of Programming Languages*, 1978.

[13] C. Cowan. Solar designer's non-executable stack patch discussion. December 1997. on-line at: http://www.cse.ogi.edu/DISC/projects/immunix/StackGuard/usenixsc98-_html/node21.html.

[14] C. Cowan, J. Beattie, S. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Conference*, 2003.

[15] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Expo*, 2000.

[16] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.

[17] G. B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[18] M. Das. Unification-based pointer analysis with directional assignements. In *Proceedings of SIGPLAN Conference on Programming language Design and Implementation*, 2000.

[19] M. Das, B. Libilit, Fähndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of Static Analysis Symposium*, 2001.

[20] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, USA, 1994.

[21] A. Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371, Paris, France, 1997.

[22] DilDog. The tao od windows buffer overflow. April 1998. http://www.cultdeadcow.com/cDc_files/cDc-351/.

[23] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in c programs via integer analysis. In *Static Analysis Symposium*, 2001.

[24] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 155–167. ACM Press, New York, NY, USA, 2003.

[25] B. Dutertre and L. De Moura. A fast linear-arithmetic solver for dpll(t). In *CAV 2006*, volume 4144 of *LNCS*. Springer, 2006.

[26] R. Ellenbogen. Fully automatic verification of absence of errors via interprocedural integer analysis. Master's thesis, University of Tel-Aviv, Israel, December 2004.

[27] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. 2000. online at: http://www.trl.ibm.com/projects/security/ssp/main.html.

[28] D. Evans, J. Guttag, J. Horning, and M. Y. Tan. Lclint: A tool for using specifications to check code. In *SIGSOFT Symposium Foundations of Software Engineering*, December 1994.

[29] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *Software Engineering Education: A Focuson Practice*, January–February 2002.

[30] S. Flisakowski. Ctree distribution. July 1997. on-line at: http://www.kagi.com/flisakow/.

[31] A. Forsgren, P. E. Gill, and M. H. Wright. Interior methods for nonlinear optimization. *SIAM Rev*, 44:525–597, 2002.

[32] N. Frykholm. Countermeasures against buffer overflow attacks. November 2000. on-line at: http://www.rsasecurity.com/rsalabs/technotes/buffer/buffer_overflow.html.

[33] Gdb — gnu project debugger. http://sources.redhat.com/gdb/.

[34] A. K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. pages 104–114.

[35] N. Halbwach, Y. E. Proy, and P. Roumanoff. Verification of realtime systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[36] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX Conference*, 1992.

[37] Reed Hastings and Bob Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *Proceedings of the winter USENIX conference*, pages 125–138, December 1992.

[38] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[39] G. Holzmann. Static source code checking for user-defined properties. In *Proceedings of 6th World Conference on Integrated Design and Process Technology*, Pasadena, CA, June 2002.

[40] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[41] S. C. Johnson. Lint, a c program checker. July 1978. on-line at: http://citeseer.nj.nec.com/johnson78lint.html.

[42] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. on-line at: http://www-ala.doc.ic.ac.uk/ phjk/BoundsCheckinig.html.

[43] Rauli Kaksonen. A functional method for assessing protocol implementation security (licentiate thesis). Tehnical Research Centre of Finland, 2001. http://www.inf.vtt.fi/pdf/publications/2001/P448.pdf, PROTOS Project: http://www.ee.oulu.fi/research/ouspg/protos/.

[44] O. Kirch. The poisoned null byte. October 1998. post to bugtraq mailing list.

[45] V. Klee, G. J. Minty, and O. Shisha. How good is the simplex algorithm? *(Eds.) In Inequalities 3*, pages 159 – 175, 1972.

[46] K. Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Master's thesis, Harvard University, Cambridge, MA, 2005.

[47] K. Kratkiewicz and R. Lippmann. Using a diagnostic corpus of c programs to evaluate buffer overflow detection by static analysis tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, June 2005.

[48] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Symposium on Fault-Tolerant Computing*, pages 230–239, 1998.

[49] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington D.C., August 2001.

[50] J.-L. Lassez and M.J. Maher. On Fourier's algorithm for linear arithmetic constraints. *Journal of Automated Reasoning*, 9:373–379, 1992.

[51] G. Leavens and A. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. *Formal Methods*, 1999.

[52] R. Lemos. Open-source team fights buffer overflows. April 2003. CNET News.com on-line at: http://web.inter.nl.net/hcc/Haj.Ten.Brugge/.

[53] F. Lindner. A heap of risk — buffer overflows on the heap and how they are exploited. Jun 2006. http://www.heise-security.co.uk/articles/74634/2.

[54] G. McGary. Bounds checking projects. on-line at: http://gcc.gnu.org/projects/bp/main.html.

[55] Microsoft's c/c++ gs compiler option + exploit to bypass it. http://www.cigital.com/news/index.php?pg=art&artid=70.

[56] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, December 1990.

[57] Nsfocus security advisory, sun solaris xsun "-co" heap overflow. on-line at: http://www.online.securityfocus.com.archive/1/265370.

[58] G. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Portland, OR, January 2002.

[59] G. C. Necula, S. McPeak, S.P. Rahul, and Q. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *International Conference on Compiler Construction*, March 2002.

[60] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the Association for Computing Machinery*, 2006. accepted for publication.

[61] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer-Verlag, New York, 1999.

[62] Open wall project. Linux kernel patch from the Openwall project, http://www.openwall.com/linux/.

[63] The frame pointer overwrite. *Phrack Magazine*, 9(55), September 1999.

[64] J.D. Pincus. Analysis is necessary, but far from sufficient. In *Invited presentation, International Symposium on Software Testing and Analysis (ISSTA), ACM SigSoft*, Portland, Oregon, August 2000.

[65] S. Ranise and C. Tinelli. The SMT-LIB Format: An Initial Proposal. 2003. on-line at: http://goedel.cs.uiowa.edu/smt-lib/.

[66] Silvio Ranise and Cesare Tinelli. Satisfiability Modulo Theories. *Trends and Controversies - IEEE Intelligent Systems Magazine*, 21(6):71–81, 2006.

[67] Sendmail consortium. October 2003. on-line at: http://www.sendmail.org.

[68] Secure software solutions. Rats, the rough auditing tool for security. September 2001. on-line at: http://www.securesw.com/rats/.

[69] Stackshield. http://www.angelfire.com/sk/stackshield/index.html.

[70] Strsafe. http://msdn.microsoft.com/library/default.aso?url=/library/en-us/winui/winui/windowsuserinterfase/resources/strings/ustrngstrsafe-functions.asp.

[71] PolySpace Tehnologies. Polyspace c verifier. Paris, France, 2003. http://www.polyspace.com.

[72] J. Viega, J.T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In *16th Annual Computer Security Applications Conference (ACSAC'00)*, 2000.

[73] J. Viega and G. McGraw. An analysis of how buffer overflow attacks work. *IBM developer works, Security articles*, March 2000. htpp://www.106.ibm.com/developerworks/security/library/smash.html?-dwzone=security.

[74] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[75] M. Vujošević-Janičić. Automated Detection of Buffer Overflows in C Programs by Using the Simplex Method. *BALCOR 2007, 8th Balkan Conference on Operational Research*, Volume of Abstracts, p118, 2007. (Šira verzija [77], na predlog programskog odbora, prijavljena za objavljivanje u specijalnom broju časopisa YUJOR.)

[76] M. Vujošević-Janičić. Ensuring Safe Usage of Buffers in Programming Language C. In *Proceedings of International Conference on Software and Data Technologies*, Porto, Portugal, 2008.

[77] M. Vujošević-Janičić, F. Marić, D. Tošić. *Using Simplex Method in Verifying Software Safety*. Prijavljeno za objavljivanje u časopisu YUJOR, 2008.

[78] Wu-ftp development group. October 2003. on-line at: http://www.wu-ftpd.org.

[79] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed System Security*, pages 3–17, San Diego, CA, February 2000.

[80] D. A. Wheeler. Flawfinder. May 2001. on-line at: http://www.dwheeler.com/flawfinder/.

[81] D. A. Wheeler. Secure programming for linux and unix howto v2.89. October 2001. on-line at: http://www.dwheeler.com/secure-programs/.

[82] J. Wilander. Security intrusion and intrusion prevention. Master's thesis, Linkopings Universitet, Sweden, April 2002.

[83] J. Wilander. Licentiate thesis:policy and implementation assurance for software security. Master's thesis, Linkopings Universitet, Sweden, October 2005.

[84] J. Wilander and M. Kamkar. A comparison of publicly available tools for static intrusion prevention. In *Proceedings of the 7th Nordic Workshop on Secure IT Systems (Nordsec 2002)*, pages 68–84, Karlstad, Sweden, November 2002.

[85] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, February 2003.

[86] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327 – 336. ACM Press, 2003.

[87] G. Yorsh and N. Dor. The Design of CoreC. 2003. on-line at: http://www.cs.tau.ac.il/ gretay/GFC.htm.

[88] M. Zhivich, T. Leek, and R. Lippmann. Dynamic buffer overflow detection. In *Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, June 2005.

[89] M. Zitser. Securing software: An evaluation of static source code analyzers. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

[90] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT international symposium on Foundations of software engineering table of contents*, pages 97–106, Newport Beach, CA, USA, 2004. ACM.