Reverzanje mrežnog stoga operacijskog sustava ADONIS

Delimar, Danko

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:662243

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-14**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1660

REVERZANJE MREŽNOG STOGA OPERACIJSKOG SUSTAVA ADONIS

Danko Delimar

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1660

REVERZANJE MREŽNOG STOGA OPERACIJSKOG SUSTAVA ADONIS

Danko Delimar

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1660

Pristupnik: **Danko Delimar (0036541606)**

Studij: Elektrotehnika i informacijska tehnologija i Računarstvo

Modul: Računarstvo

Mentor: izv. prof. dr. sc. Stjepan Groš

Zadatak: **Reverzanje mrežnog stoga operacijskog sustava ADONIS**

Opis zadatka:

PLC-ovi su specijalizirani uređaji koji se upotrebljavaju u industrijskim pogonima. Radi se o kritičnoj komponenti iz perspektive sigurnosti o čijem unutarnjem radu se malo zna zbog toga što proizvođači ne objaljuju informacije koje bi omogućile nezavisnu analizu sigurnosti takvih sustava. U tom smislu, mora se pribjeći reverzanju kako bi se otkrila funkcionalnost te utvrdilo postoje li kakve ranjivosti koje se mogu iskoristiti. U sklopu završnog rada potrebno je reverzati operacijski sustav ADONIS koji se upotrebljava na uređajima PLC proizvođača Siemens. Konkretno, potrebno je utvrditi koji dijelovi koda implementiraju mrežne protokole (Ethernet, IP, TCP) te ispratiti kako se obrađuje pojedini paket - od primitka s mreže - do isporuke aplikacijskom sloju. Naglasak je potrebno staviti na alate i metodologiju rada, a manje na konkretne rezultate.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

1. Uvod **3**

2. Opće informacije **4**

2.1. ADONIS OS 4

2.2. S7-1200 PLC 5

2.2.1. Sklopovlje 5

3. Statička analiza **7**

3.1. Ghidra 7

3.1.1. Skriptanje Ghidre 8

3.1.2. Otkrivanje informacija preko stringova 10

3.1.3. Otkrivanje informacija pomoću memorijske mape 12

4. Dinamička analiza **13**

4.1. QEMU 14

4.1.1. Pokretanje emulatora 14

4.2. GDB Plugin 17

4.2.1. Pojašnjenje primjera koda 18

5. Mrežni stog **20**

5.1. TCP/IP stog 20

5.2. S7Comm i S7CommPlus Protokoli 21

5.3. ACE 22

5.4. Web Server 22

5.5. Ostale mrežne sposobnosti 24

6. Zaključak **25**

Sažetak **28**

Abstract **29**

A: Kod skripte za detekciju podsustava **30**

1. Uvod

Otkad je 2010. otkriven Stuxnet virus, sigurnost industrijskih sustava upravljanja postala je jako važna [3]. Glavni uređaji u takvim sustavima su programirljivi logički kontroleri (engl. *Programmable logic controller, PLC*). PLC-ovi su računala specijalizirana za automatizaciju koja moraju uvijek raditi, a postoje i sve veći zahtjevi korisnika za novim mogućnostima tih računala. Zbog toga su PLC-ovi unikatna meta za razne vrste napada, ali i unikatna meta proučavanja sigurnosnim istraživačima.

Kako svijet sve više teži stalnoj mrežnoj povezanosti, tako i PLC-ovi dobivaju sve više mrežnih sposobnosti. Ovo je vrlo interesantno iz perspektive sigurnosti jer je jedan od glavnih načina osiguravanja PLC-ova ne povezivati ih s vanjskim svijetom. Tako da svaka mrežna mogućnost PLC-a mora biti jako dobro pregledana kako bismo bili sigurni da u njoj ne postoje greške. Također, bitno je navesti kako proizvođači PLC-ova ne žele široj populaciji istraživača omogućiti pristup izvornom kodu pa je često potrebno koristiti metode poput reverznog inženjeringa.

Ovaj će se rad fokusirati na otkrivanje mrežnih sposobnosti operacijskog sustava ADONIS koji se koristi u Siemensovim SIMATIC PLC-ovima s posebnim fokusom na SIMATIC S7-1200 PLC. Prvi dio će objasniti općenite značajke SIMATIC PLC-ova i ADONIS OS-a. Drugi dio će objasniti načine kako smo pristupili reverzanju ovog PLC-a kroz statičku i dinamičku analizu. U svakom od tih dvaju poglavlja objasnit ćemo alate koje smo koristili i alate koje smo razvili za analizu ovog PLC-a. Zadnje će poglavlje pokušati opisati sve mrežne sposobnosti PLC-a i što se sve saznalo o njima korištenjem reverznog inženjerstva.

2. Opće informacije

SIMATIC PLC-ovi [13] su Siemensova linija PLC-ova koja je zamišljena za široku uporabu u raznim industrijskim procesima. Ta linija PLC-ova na sebi ima Siemensov vlasnički operacijski sustav o kojem je malo toga poznato. Ovaj rad stavlja fokus na specifičan PLC u toj liniji S7-1200 koji je zamišljen kao PLC opće namjene za manje i srednje industrijske procese.

2.1. ADONIS OS

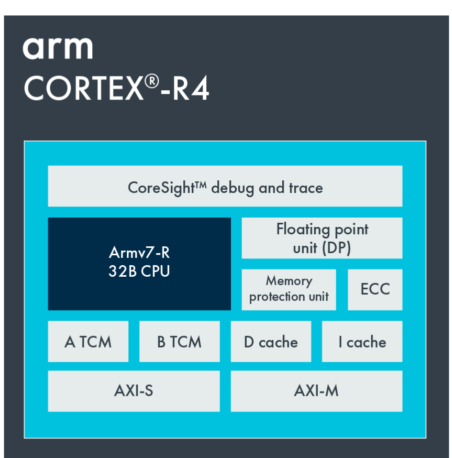
ADONIS je vlasnički *Real Time Operating System (RTOS)* tvrtke Siemens. Većina općenitih informacija o ovom OS-u dolazi iz prezentacije na Black Hatu iz 2019. [1], a specifičnije informacije iz ovog rada i iz radova na ovu temu koji su nastali prije našeg istraživanja [10] [9]. Puno informacija o operacijskom sustavu moguće je saznati preko sustava za praćenje dnevnika. Taj sustav u operacijskom sustavu omogućava da se, ako ijedna komponenta zakaže u nekom trenutku, u dnevnik upiše koja je točno komponenta zakazala, uz neke dodatne informacije. Tako je na primjer moguće vidjeti da operacijski sustav podržava dretve, signale, monitore, *spinlockove* i slično. Više o tome kako se iskoristio sustav dnevnika može se vidjeti u sekciji o statičkoj analizi. Operacijski sustav pokreće se tako da se izvrši niz inicijalizacijskih funkcija koje inicijaliziraju fizičke komponente poput *Vectored Interrupt Controllera (VIC)* i *Memory Management Unita (MMU)*, ali i ne-fizičke poput stvaranja reda za dretve i inicijaliziranja gomile. Nakon toga OS inicijalizira prvu dretvu i pokrene je, nakon čega je inicijalizacija OS-a gotova i PLC kreće s normalnim izvođenjem iz te početne dretve. Neki elementi poput gomile imaju i posebne zapise u dnevniku koji su specifični za verziju PLC-a, tako da je moguće da svaki PLC iz SIMATIC obitelji nadograđuje ADONIS na način koji je potreban za njegov rad.

2.2. S7-1200 PLC

S7-1200 PLC je uz S7-1500 jedan od dva općenamjenska PLC-a koje SIMATIC linija nudi. Glavna je točka prodaje da je PLC modularan i moguće ga je lagano nadograditi s dodatnim komponentama koje Siemens nudi, poput izdavanja posebnih vrsta signala ili komunikacija preko specifičnih protokola. Ovo znači da svaki PLC koji je ugrađen u nekom postrojenju ne mora podržavati sve funkcionalnosti koje podržava neki drugi i da se potencijalne ranjivosti u nekom od tih modula ne moraju prenositi na svako postrojenje. Jedna od glavnih mogućnosti PLC-a je integrirani web server o kojem će se više govoriti u poglavlju o mrežnim sposobnostima i preko kojeg je moguće pristupiti raznim dijelovima PLC-a. Preko tog web servera moguće je nadograditi PLC i taj je proces zanimljiv za buduća istraživanja jer kada bi se mogle zaobići zaštite od neovlaštene nadogradnje, bio bi odličan način za dobivanje kontrole nad PLC-om.

2.2.1. Sklopovlje

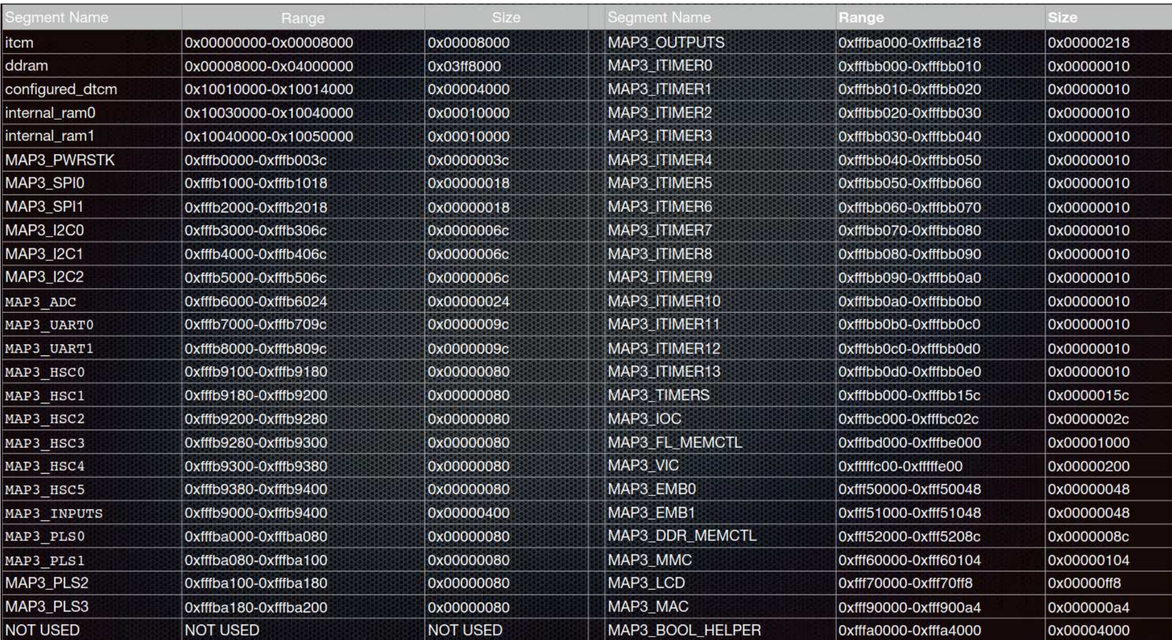
Za ovaj je rad najbitniji aspekt PLC-a arhitektura na kojoj je baziran. PLC koristi ARM [2] Cortex R4 procesor. Taj je procesor posebno dizajniran za brze i energetski efikasne operacije u stvarnom vremenu. Procesor podržava ARMv7 arhitekturu koja radi u *big endian* načinu rada, a podržava i ARM-ov Thumb način rada. Službeni dijagram koji pokazuje arhitekturu Cortex R4 procesora prikazan je na 2.1.



Slika 2.1. Dijagram Cortex R4 procesora

Ostatak sklopovlja za ovaj rad nije bio previše bitan, osim kada je trebalo određeni uređaj emulirati za dinamičku analizu, ali pokazuje se da za samo početno pokretanje

PLC-a nije potrebno veliko razumijevanje sklopovlja. Interesantno je primijetiti da procesor podržava CoreSight koji omogućuje *debugging* procesora, ali postoji *watchdog* koji će prepisati memoriju PLC-a s nulama ako se proces *debugginga* pokuša preko CoreSighta. Ako se pokuša koristiti stvarno sklopovlje PLC-a za istraživanje, treba biti oprezan oko takvih zaštita jer je u slučaju neopreza moguće uništiti PLC. Za ostatak periferije PLC koristi *Memory Mapped Input/Output (MMIO)* pristup, što znatno olakšava proces emuliranja periferije. Memorijska mapa perifernih uređaja dostupna je ovdje 2.2., a dolazi iz prijašnjeg rada (Abbasi et al.) [1].



Segment Name	Range	Size	Segment Name	Range	Size
itcm	0x00000000-0x00008000	0x00008000	MAP3_OUTPUTS	0xfffba000-0xfffba218	0x00000218
ddram	0x00008000-0x04000000	0x03ff8000	MAP3_ITIMER0	0xfffbb000-0xfffbb010	0x00000010
configured_dtcm	0x10010000-0x10014000	0x00004000	MAP3_ITIMER1	0xfffbb010-0xfffbb020	0x00000010
internal_ram0	0x10030000-0x10040000	0x00010000	MAP3_ITIMER2	0xfffbb020-0xfffbb030	0x00000010
internal_ram1	0x10040000-0x10050000	0x00010000	MAP3_ITIMER3	0xfffbb030-0xfffbb040	0x00000010
MAP3_PWRSTK	0xfffb0000-0xfffb003c	0x0000003c	MAP3_ITIMER4	0xfffbb040-0xfffbb050	0x00000010
MAP3_SPI0	0xfffb1000-0xfffb1018	0x00000018	MAP3_ITIMER5	0xfffbb050-0xfffbb060	0x00000010
MAP3_SPI1	0xfffb2000-0xfffb2018	0x00000018	MAP3_ITIMER6	0xfffbb060-0xfffbb070	0x00000010
MAP3_I2C0	0xfffb3000-0xfffb306c	0x0000006c	MAP3_ITIMER7	0xfffbb070-0xfffbb080	0x00000010
MAP3_I2C1	0xfffb4000-0xfffb406c	0x0000006c	MAP3_ITIMER8	0xfffbb080-0xfffbb090	0x00000010
MAP3_I2C2	0xfffb5000-0xfffb506c	0x0000006c	MAP3_ITIMER9	0xfffbb090-0xfffbb0a0	0x00000010
MAP3_ADC	0xfffb6000-0xfffb6024	0x00000024	MAP3_ITIMER10	0xfffbb0a0-0xfffbb0b0	0x00000010
MAP3_UART0	0xfffb7000-0xfffb709c	0x0000009c	MAP3_ITIMER11	0xfffbb0b0-0xfffbb0c0	0x00000010
MAP3_UART1	0xfffb8000-0xfffb809c	0x0000009c	MAP3_ITIMER12	0xfffbb0c0-0xfffbb0d0	0x00000010
MAP3_HSC0	0xfffb9100-0xfffb9180	0x00000080	MAP3_ITIMER13	0xfffbb0d0-0xfffbb0e0	0x00000010
MAP3_HSC1	0xfffb9180-0xfffb9200	0x00000080	MAP3_TIMERS	0xfffbb000-0xfffbb15c	0x0000015c
MAP3_HSC2	0xfffb9200-0xfffb9280	0x00000080	MAP3_IOC	0xfffbc000-0xfffbc02c	0x0000002c
MAP3_HSC3	0xfffb9280-0xfffb9300	0x00000080	MAP3_FL_MEMCTL	0xfffbd000-0xfffbe000	0x00001000
MAP3_HSC4	0xfffb9300-0xfffb9380	0x00000080	MAP3_VIC	0xfffffc00-0xfffffe00	0x00000200
MAP3_HSC5	0xfffb9380-0xfffb9400	0x00000080	MAP3_EMB0	0xfff50000-0xfff50048	0x00000048
MAP3_INPUTS	0xfffb9000-0xfffb9400	0x00000400	MAP3_EMB1	0xfff51000-0xfff51048	0x00000048
MAP3_PLS0	0xfffba000-0xfffba080	0x00000080	MAP3_DDR_MEMCTL	0xfff52000-0xfff5208c	0x0000008c
MAP3_PLS1	0xfffba080-0xfffba100	0x00000080	MAP3_MMC	0xfff60000-0xfff60104	0x00000104
MAP3_PLS2	0xfffba100-0xfffba180	0x00000080	MAP3_LCD	0xfff70000-0xfff70ff8	0x00000ff8
MAP3_PLS3	0xfffba180-0xfffba200	0x00000080	MAP3_MAC	0xfff90000-0xfff900a4	0x000000a4
NOT USED	NOT USED	NOT USED	MAP3_BOOL_HELPER	0xfffa0000-0xfffa4000	0x00004000

Slika 2.2. MMIO periferije PLC-a

3. Statička analiza

Statička analiza jedan je od dvaju glavnih pristupa korištenih u procesu reverzanja. Općeniti postupak statičke analize uključuje korištenje specijaliziranih alata koji uzimaju binarne datoteke i pretvaraju ih u asemblerski ispis ili u nekim slučajevima čak dekompajliraju taj asemblerski kod u pseudo c kod. Statička se analiza često koristila kada je trebalo zaključiti kako određeni periferni uređaj funkcionira, na kojim memorijskim adresama se nalazi u MMIO-u, kako pristupa tim adresama i koje vrijednosti vraća ako se pristupi određenim adresama unutar adresnog prostora uređaja. Tijekom statičke i dinamičke analize fokus je bio na *firmware* verziji 4.5.2, ali ne očekuju se prevelike razlike između svih verzija *firmwarea* ako je ta verzija dio velike verzije 4.

3.1. Ghidra

Ghidra [8] je alat otvorenog tipa koda koji je izradila američka Nacionalna sigurnosna agencija (NSA) koji olakšava proces reverznog inženjerstva. Glavna funkcionalnost Ghidre je mogućnost dekompilacije asemblerskog koda u pseudo c kod koji je puno lakše čitati od tradicionalnog asemblera. Taj dekompajliran kod ima sposobnost prilagođavanja i poboljšavanja ovisno o tome koje korisnik strukture, funkcije i tipove podataka ručno definira unutar Ghidre. Također, Ghidra podržava pregled asembler koda, pregled svih *stringova* u datoteci, pregled poziva funkcija u grafu i mnoge druge funkcionalnosti. Ghidra također pokušava automatski prepoznati funkcije, strukture, enumeracije i slične koncepte iz jezika poput c unutar asemblerskog koda, a dozvoljava korisniku i da sam definira te koncepte ako su potrebni i bolje opisuju stanje unutar programa. Funkcionalnost koju ovaj rad najviše iskorištava jest mogućnost pisanja skripti koje imaju pristup svim elementima unutar Ghidre kao što su asemblerski kod, funkcije strukture i slično i mogu manipulirati svim tim elementima kako bi korisnik bolje razumio program koji po-

kušava reverzati. Napisana je jedna relativno velika skripta koja omogućuje automatsku detekciju funkcija koje pripadaju određenom podsustavu PLC-a.

3.1.1. Skriptanje Ghidre

Skripte unutar Ghidre pišu se u programskom jeziku Java ili Python. U ovom radu je korištena Java jer je podrška unutar Ghidre bolja za skripte u Javi i one se izvršavaju puno brže.

```java
public class NewScript extends GhidraScript {
    @Override
    public void run() throws Exception {
        println("Hello World");
    }
}
```

Listing 3..1: Minimalna Ghidra skripta

Kao što je vidljivo iz primjera minimalne Ghidra 3.1.1. skripte, svaka skripta mora naslijediti klasu GhidraScript i nadjačati metodu run unutar koje zatim može pisati proizvoljni kod.

Spomenuta skripta za automatsku detekciju funkcija funkcionira na temelju sustava dnevnika unutar PLC-a. Kako je već prije objašnjeno, taj sustav, kad god neka komponenta sustava zataji, u dnevnik upiše koji je podsustav zatajio i određene dodane informacije. Za ovaj rad bitna je jedino informacija koji je sustav zatajio jer određene funkcije unutar *firmwarea* pozivaju dnevničke funkcije i to s točnim argumentom u kojem smo trenutno podsustavu. Tako je moguće pronaći puno funkcija koje pripadaju nekom podsustavu. Također, ako se pretraže *stringovi* unutar *firmwarea*, moguće je pronaći sve nazive komponenata podsustava i kodova koji se šalju unutar funkcije za upis u dnevnik. Primjer takvog jednog opisnika sustava može se vidjeti ovdje 3.1. Ovaj opisnik odgovara podsustavu [ADONIS_THREADS] i na prvoj liniji slike vidi se kod 0x13 koji odgovara kodu ovog podsustava. Kada se poziva funkcija dnevnika, taj kod će se poslati kao prvi argument toj funkciji. U ARM arhitekturi prvi argument šalje se kroz registar r0 pa tako prije poziva funkcije dnevnika moramo provjeriti stanje registra r0. Ako zatim prođemo kroz cijeli kod i svaki put kada se pozove neka od dnevničkih funkcija, a stanje registra

r0 je primjerice 0x13, znamo da funkcija unutar koje je poziv dnevničkoj funkciji pripada podsustavu za dretve. Postoji komplikacija oko ovog pristupa, a to je da dnevničkih funkcija ima 10 (broj argumenata osim koda 0-9) i te se funkcije pozivaju preko virtualne tablice koja se inicijalizira kada se pokreće operacijski sustav. To znači da Ghidra ne može automatski detektirati argumente, ali barem može prepoznati kada postoji referenca na tu virtualnu tablicu. Zato na svakom mjestu gdje takva referenca postoji, prvo idemo unaprijed unutar asemblerskog koda kako bismo pronašli mjesto poziva funkcije, a zatim od mjesta poziva idemo unatrag kako bismo pronašli na što je postavljen registar r0. Ovaj postupak ima veliki nedostatak u tome da kompajler na više načina poziva funkcije iz virtualne tablice i potrebno je ovaj postupak traženja napisati drugačije za svaki od tih načina. U okviru ovog rada pronađena su tri distinktivna obrasca koja kompajler koristi za poziv funkcije iz virtualne tablice i to je bilo dovoljno da se pronađe većina funkcija. Kada je skripta gotova, ispisat će tablicu u kojoj će se moći vidjeti sve funkcije koje je skripta prepoznala kao dio određenog podsustava kao i mjesto unutar te funkcije gdje se nalazi poziv na neku od dnevničkih funkcija. Bitno je napomenuti da postoje opisnici podsustava za koje ne postoji niti jedan poziv na dnevničku funkciju. To može značiti ili da se podsustav stvarno ne koristi u osnovnoj verziji *firmwarea* (recimo samo je moguće dobiti taj sustav iz jedne od modularnih nadogradnji za PLC) ili se podsustav koristi normalno, ali nigdje u kodu ne postoji poziv na dnevničku funkciju za taj podsustav ili naša skripta iz nekog razloga ne radi za taj podsustav. Ovo se ne bi trebalo često događati ako se pregledavaju osnovni podsustavi operacijskog sustava poput podsustava za dretve ili podsustava za provođenje nadogradnje. Puni kod ove skripte moguće je vidjeti u privitku A ovog dokumenta.

```
00fcff58 00 13       dw     [ADONIS_THREADS]
00fcff5a 00 01       dw     1h
00fcff5c 5b 41 44    ds     "[ADONIS_THREADS]"
         4f 4e 49
         53 5f 54 ...
```

Slika 3.1. Primjer opisnika podsustava u dnevniku

Pisanje skripti unutar Ghidre jako je moćan alat za automatizaciju analize koda te je planirana i nadogradnja ove skripte kako bi mogla detektirati i sve funkcije koje pozivaju funkciju koju detektiramo da je iz nekog podsustava. Tako bi se vrlo efikasno moglo po-

tencijalno mapirati sve funkcije koje pripadaju određenom podsustavu. Nažalost, ovaj proces je i dalje prilično spor jer jako ovisi o sposobnosti Ghidre da automatski detektira blokove koda koji pripadaju nekoj funkciji, a trenutno ta sposobnost na ARM arhitekturi nije savršena. Stoga ovaj proces automatske detekcije funkcija zahtijeva puno ručnog popravljanja unutar Ghidre kako bi se svaka funkcija definirala točno, ali uz pomoć skripti proces je dosta ubrzan i definitivno je izvediv.

3.1.2. Otkrivanje informacija preko stringova

Korištenje *stringova* jedna je od osnovnih tehnika statičke analize jer vrlo često u programima postoje *stringovi* koji mogu otkriti ponašanje određenih dijelova koda. Ghidra tu opet može pomoći jer olakšava detekciju i filtriranje *stringova* unutar *firmwarea*, ali moguće je koristiti i jednostavne alate unutar komandne linije poput alata *strings* i *grep*. *Firmware* ovog PLC-a jako je velik i postoji velik broj *stringova* koji se mogu iskoristiti kako bi se saznalo nešto novo o PLC-u.

Jedan od glavnih izvora korištenih informacija bile su poruke unutar funkcije *assert*. *Assert* se koristi u brojnim bibliotekama kako bi se utvrdilo da je stanje programa onakvo kako je programer očekivao i ako to nije slučaj, *assert* će srušiti proces. Na brojnim mjestima unutar *firmwarea* moguće je pronaći pozive na *assert* koji dolaze iz vanjskih biblioteka ili iz internih implementacija dijelova *firmwarea*. Jako je korisna informacija koja se nalazi u svakom *assert* pozivu puni naziv datoteke u kojoj se *assert* nalazi. Jedan takav primjer za biblioteku *openssl* može se vidjeti ovdje 3.2. Taj je primjer prikaz iz Ghidrinog dekompajliranog pogleda na kod.

```
fprintf(&PTR_stderr_013df0d4,(byte *)"Failed assertion `%s\' at line %d of `%s\'.\n",
        "in && out && key",0x592,
        "C:/Sources/fwk_4.5.2/S7JCYP_S7PFW__SRC/openssl/crypto/aes/aes_core.c");
```

Slika 3.2. Primjer asserta u kodu

Vidimo da je u ovom assertu dostupan puni put do datoteke pa se može vidjeti da se ovaj poziv nalazi unutar datoteke aes_core.c unutar biblioteke *openssl*. *Openssl* je biblioteka otvorenog tipa koda koja implementira velik broj kriptografskih algoritama. Ako posjetimo službeni repozitorij biblioteke i navigiramo se u datoteku aes_core.c, vrlo brzo ćemo pronaći funkciju koja točno odgovara našoj funkciji unutar *firmwarea*. Ovim procesom može se znatno smanjiti posao prilikom reverzanja jer se može znati točno koja

funkcija obavlja koji zadatak ako ima ovakav assert. Ovim procesom uspješno je otkriveno da PLC koristi biblioteku *openssl*. No, možemo i više od toga. Ako otvorimo u Ghidri neku drugu verziju *firmwarea*, primijetit ćemo da se u toj verziji ne nalaze isti *assertovi*. Moguće je da je za neke specifične dijelove koda taj kod stvarno promijenjen i ne koristi više istu biblioteku, ali vjerojatnije je da je samo *assert* maknut iz koda, a biblioteka se i dalje koristi. Tako primjerice ako se vratimo u verziju 4.1.2 možemo pronaći reference na datoteke iz InterNiche TCP/IP mrežnog stoga napravljenoga specifično za ARM arhitekturu. Korištenje tog mrežnog stoga može se potvrditi i time što među opisnicima za dnevnik postoji opisnik s imenom [TCIP_INICHE].

Ovo su neke od vanjskih biblioteka koje je moguće detektirati korištenjem *assert stringova*. Ali, moguće je detektirati i interne elemente PLC-a. Ako pogledamo sljedeći primjer 3.3. iz Ghidrinog dekompajlerskog prikaza, vidjet ćemo referencu na neku datoteku koja je dio *pdcfs-a*. Iz ostalih dijelova *stringa*, ali i istraživanjem, moguće je zaključiti da je PDCFS datotečni sustav. Specifično, to je vlasnički datotečni sustav koji je Siemens razvio za svoje PLC-ove i zamišljen je za rad u stvarnom vremenu.

```
iVar2 = FUN_00c3fdfc("C:/Sources/fwk_4.5.2/s7p.fileservices/pdcfs/FAT/src/pdcfs_fdlc.c",0x3c6,
                     param_4,&local_20,uVar5 + 0x48);
```

Slika 3.3. Primjer internog asserta u kodu

Koristeći isti pristup, zaključuje se i da za implementaciju c++ standardne biblioteke koriste Dinkumware verziju c++ standardne biblioteke.

Sljedeći način kako su korišteni *stringovi* jest pomoću c++ imena klasa. Opisnici klasa u c++-u imaju u sebi zapisano ime tipa podatka kako bi određeni *castovi* funkcionirali unutar jezika. To je za nas odlično jer možemo detektirati korištenje određenih biblioteka unutar koda. Ghidra je također tu od velike pomoći jer automatski dodaje ime klase u ime funkcije kako bi nam posao bio lakši. Tako se primjerice u deskriptorima klasa mogu pronaći imena klasa koja u sebi spominju biblioteke ACE i TAO.

ACE i TAO su biblioteke koje se koriste u mrežnom programiranju i o njima će se više govoriti u sekciji o mrežnom stogu PLC-a.

```
                     class_descriptor_00fb4f3c                      XREF[2]:     00fb4668(*)
    00fb4f3c 00 fc f3        class_de…
             e0 00 fb
             4f 4c 02 …
       00fb4f3c 00 fc f3 e0     uint      FCF3E0h                  constant
       00fb4f40 00 fb 4f 4c     char *    s_ACE_6_5_0::ACE_Timer… name            = "ACE_6_5_0::ACE
       00fb4f44 02 4f 99 0f     uint      24F990Fh                 class_uid
       00fb4f48 00 fb 50 0c     void *    PTR_class_descriptor_0… class_parent… = 00fb4ea0

                     s_ACE_6_5_0::ACE_Timer_Queue_T<ACE_00fb4f4c     XREF[1]:     00fb4f40(*)
    00fb4f4c 41 43 45        ds         "ACE_6_5_0::ACE_Timer_Queue_T<ACE_6_5_0::ACE_E…
             5f 36 5f
             35 5f 30 …
```

Slika 3.4. Primjer Ghidra prikaza c++ deskriptora klase

3.1.3. Otkrivanje informacija pomoću memorijske mape

S obzirom na raspoloživost memorijske mape PLC-a, moguće je zaključiti što radi koja funkcija ako pristupa memoriji unutar memorijske mape. Primjerice, vrlo često prilikom pokretanja operacijskog sustava ili prilikom pokretanja *bootloadera* PLC često pristupa memoriji koja pripada nekom od *timera*. Iz toga se može zaključiti da PLC čeka na neku potvrdu od neke periferije, na primjer da je RAM spreman za korištenje. Vjerojatno se može zaključiti i koja je to periferija ako pogledamo malo kod prije i primijetimo kojoj periferiji je PLC pristupao. Na ovaj se način može zaključiti puno o procesu pokretanja operacijskog sustava ili samog PLC-a, što će biti jako korisno u poglavlju o dinamičkoj analizi jer će pomoći izgraditi vjerniji emulator PLC-a.

4. Dinamička analiza

Dinamička analiza je proces pokretanja programa koji se analizira, najčešće unutar specijaliziranih *debuggera*, s ciljem shvaćanja kako program točno funkcionira. To je u kontrastu sa statičkom analizom prilikom koje se pokušava dobiti šira slika o programu i shvatiti otprilike koji dio koda je zaslužan za koju funkcionalnost kako bi se moglo bolje fokusirati na specifičan kod od interesa. S obzirom na to da dinamička analiza zahtijeva pokretanje programa ili u ovom slučaju cijelog sustava, nailazimo na nekoliko problema koje nemamo prilikom statičke analize. Dva najveća problema s kojima se susreće ovaj rad jesu: pokretanje samog sustava, a kad smo ga pokrenuli, htjeli smo imati dobru vidljivost u to što točno sustav radi i htjeli smo imati finu kontrolu nad tim izvršavanjem.

Problem pokretanja proizlazi iz toga što je ovaj sustav na ARM arhitekturi i što je zamišljen za pokretanje na specifičnom sklopovlju. Ovo je riješeno izradom QEMU [11] emulatora. Prije ovog rada emulator je već bio donekle izrađen od prije, a više o početnom procesu izrade može se pročitati u diplomskom radu Ardiana Pantine [9]. Ovaj je rad emulator u izvođenju pogurao dalje nego što je prije bio tako što su u emulator implementirani novi periferni uređaji koji prije nisu postojali.

Problem vidljivosti i kontrole unutar sustava bio je problem kojim smo se više bavili. U standardim programima vidljivost i kontrolu nije teško postići jer *debugger* unutar kojeg se program pokreće ima implementiranu većinu željene funkcionalnosti. To su primjerice komande poput *step* za izvršavanje jedne instrukcije ili razne komande za pregled trenutne memorije na stogu, gomili ili vrijednosti unutar registara. QEMU je ponajprije emulator i neke od ovih mogućnosti nisu ni moguće direktno poput komande *step*, a druge, ako i jesu moguće, poput pregledavanja registara zahtijevaju drugačiji način rada nego što smo mi navikli vidjeti unutar *debuggera*. QEMU rješava ovaj problem tako što pruža sučelje za GDB, najčešći *debugger* za linux sustave. Nažalost, to suče-

lje na ovom sustavu nije funkcioniralo najbolje, GDB se može spojiti, ali kontrola nad emulatorom je vrlo ograničena. Specifično, jedino što je moguće jest mijenjati i čitati memoriju, kontrola izvršavanja nije nikako radila. Zašto se ovo točno događa, trenutno nije poznato, ali moguće je da ima veze s time da je PLC-ov sustav čisti ARM kod koji je razlomljen u *bootloader* i *firmware*, a ne jedinstveni program unutar izvršnog formata kao što je ELF. Moguće je da bi, kada bi se napravila specifična ELF datoteka koja bi imala oba elementa i *bootloader* i *firmware*, ta komunikacija s GDB-om radila bolje, ali takva datoteka nije uspješno izrađena. Umjesto toga napisan je *plugin* za GDB koristeći programski jezik Python jer GDB podržava pisanje *pluginova* u Pythonu. Time je postalo moguće implementirati većinu kontrole i vidljivosti i to na fleksibilan i proširiv način.

4.1. QEMU

QEMU (*Quick EMUlator*) je program otvorenog tipa koda koji podržava emulaciju i virtualizaciju većine procesorskih arhitektura i, što je za ovaj rad vrlo važno, podržava ručnu implementaciju perifernih uređaja kako bi se mogao izraditi emulator specifično za ovaj PLC. Također QEMU podržava izvršavanje komandi nad ovim emulatorom koje se izvršavaju preko QEMU-ovog monitor protokola (QMP). Te komande omogućavaju generičku kontrolu nad emulatorom i većinom omogućavaju pregled informacija o trenutnom stanju emulatora poput trenutnih vrijednosti registara ili trenutno učitanih perifernih uređaja. Nažalost, opet je puno tih komandi neiskoristivo jer su zamišljene specifično za pregled stanja sustava koji su bazirani na linux jezgri ili na x86 intel arhitekturi.

4.1.1. Pokretanje emulatora

Za uspješno pokretanje emulatora, potrebno je prvo odabrati na kojem će se procesoru pokretati kod. Originalni PLC ima u sebi ARM Cortex R4 procesor. Nažalost, QEMU nema službenu potporu za taj procesor, pa su prijašnji radovi [10] [9] odabrali modificiranu verziju QEMU-a tvrtke Xilinx [15] koja ima podršku za procesor ARM Cortex R5 koji je, iako nije isti kao ciljani procesor, dovoljno sličan da je moguće nastaviti dalje. Nakon odabira QEMU-a koji će se koristiti, potrebno je napraviti određene modifikacije *bootloaderu* i *firmwareu*. Prvi problem je da se PLC pokreće u *big endian* modu, ali QEMU podržava izvršavanje samo u *little endian* modu tako da je potrebno izmijeniti

sustav u *little endian* način rada. Nakon toga potrebno je maknuti instrukcije specifične za procesor s obzirom na to da procesor unutar emulatora nije točno isti kao procesor na PLC-u. To su instrukcije specifične za koprocesor *msr, mrs, mcr i mrc* i te su instrukcije zamijenjene instrukcijama *mov r0, r0, r0* koje ne rade ništa i efektivno su *nop* instrukcije. Također, u normalnom radu PLC-a *bootloader* kopira prvih 0x8000 bajtova *firmwarea* u RAM kojemu zatim prepušta kontrolu. Taj proces ručno simuliramo unutar emulatora i zato je potrebno pripremiti datoteku *firmwarea* tako da prvih 0x8000 bajtova stavimo u posebnu datoteku koju će emulator ručno učitati u memoriju na točnu adresu. Ovaj proces je prije opisan u diplomskom radu Ardiana Pantine [9]. Cijeli ovaj proces prilično je dugačak i podložan greškama i zato je u sklopu ovog rada izrađen set od 5 skripti koje olakšavaju cijeli proces od kompilacije QEMU-a, pripreme *bootloader* i *firmware* datoteka do pokretanja samog emulatora u normalnom načinu rada i u načinu rada koji je otvoren za *debugging*. Sami kod ovih skripti kao detaljniji opis cijelog procesa pokretanja i pripreme emulatora, a i uputa kako dodati novu periferiju u emulator, može se pronaći na github repozitoriju projekta [5].

Izlaz emulatora

Kada se pokrene emulator, vidi se, prema zadanim postavkama pokretačke skripte, ARM instrukcije koje se trenutno izvršavaju. U tom istom prozoru u kojem se pokreće emulator moguće je upisivati QMP komande kako bi se promijenio primjerice prikaz instrukcija koje se ispisuju ili se dobiju informacije o stanju procesora kao što je trenutno stanje registara. Ograničenje je ovog QMP sučelja da ne dopušta dobru kontrolu nad tijekom izvršavanja emulatora. Jedine komande koje utječu na tijek izvršavanja su *stop* i *continue* koje mogu samo pauzirati i pokrenuti emulator. Ne postoji način za postavljanje *breakpointova* ili izvršavanje jedne po jedne instrukciju. Za takvu kontrolu treba nam GDB i zato je razvijen *plugin* za GDB koji omogućuje ovakvu kontrolu.

```
0x00c0ed04:  e7d40007  ldrb     r0, [r4, r7]

----------------
IN:
0x00c0ed08:  e3500000  cmp      r0, #0

----------------
IN:
0x00c0ed0c:  1a000006  bne      #0xc0ed2c

----------------
IN:
0x00c0ed10:  ea000014  b        #0xc0ed68

----------------
IN:
0x00c0ed68:  e1b00005  movs     r0, r5

----------------
IN:
0x00c0ed6c:  08bd84f0  popeq    {r4, r5, r6, r7, sl, pc}

----------------
IN:
0x00c0c05c:  e2501000  subs     r1, r0, #0

----------------
IN:
0x00c0c060:  0a000040  beq      #0xc0c168

----------------
IN:
0x00c0c168:  e5961000  ldr      r1, [r6]

----------------
IN:
0x00c0c16c:  ebffff51  bl       #0xc0beb8

stop
(qemu)
```

Slika 4.1. Prikaz izlaza emulatora

4.2. GDB Plugin

GDB podržava izradu *pluginova* koji mogu definirati nove komande pomoću sučelja u programskom jeziku Python. Prednost je takvih *pluginova* da ne moraju imati samo interakciju s GDB-om i procesom koji GDB vidi, nego mogu kontaktirati vanjske servise, u ovom slučaju kontaktirat će QEMU-ov QMP servis koji će omogućiti izvršavanje svih komandi koje može korisnik normalno izvršavati u prozoru QEMU-a, a s obzirom na to da je *plugin* unutar GDB-a, moguće je kontaktirati QEMU i preko GDB-a. Kako je prijašnje objašnjeno, postoji način da se kroz GDB mijenja memorija, ali ne može se ništa drugo. Dobra je stvar da je moguće mijenjati i memoriju u kojoj se nalaze instrukcije programa. Ovdje 4.2. je uključen dio koda koji pokazuje kako je implementirano postavljanje *breakpointova* unutar emulatora, ali postoji više komandi koje su implementirane i sve su vidljive u glavnom repozitoriju projekta [5].

```
1  breakpoints = {}
2  class PLC_SetBreakpoint(gdb.Command):
3      def __init__(self):
4          super(PLC_SetBreakpoint, self).__init__("plc-break", gdb.
       COMMAND_USER)
5  
6      def invoke(self, arg, from_tty):
7          qmp = QEMUMonitorProtocol(("localhost", 4444))
8          qmp.execute("qmp_capabilities")
9  
10         i = gdb.inferiors()[0]
11         m = i.read_memory(int(arg, 16), 4)
12         addr = m.tobytes()[::-1].hex()
13         gdb.write(addr + "\n")
14         breakpoints[arg[2:]] = m.tobytes()[::-1]
15 
16         i.write_memory(int(arg, 16), bytes.fromhex("feffffea"))
17         gdb.write("Breakpoint set.\n")
```

Listing 4..1: Implementacija *breakpoint* komande

Glavna je ideja iza ove implementacije *breakpointova* ta da ako se memoriju na mjestu gdje se nalaze instrukcije može mijenjati, može se i umjesto neke instrukcije upisati instrukciju koja će efektivno biti *breakpoint*. Normalni *debuggeri* rade na potpuno istoj ideji,

samo što oni instrukciju zamijene prekidom koji *debugger* može uloviti i ispisati trenutno stanje programa u tom trenutku. Budući da se GDB ne može dobro spojiti s QEMU-om, ne mogu se koristiti takvi prekidi jer ih GDB neće uloviti. Na sreću, ARM arhitektura podržava instrukciju **b #0** (u heksadekadskom prikazu ova instrukcija se kodira kao 0xeafffffe). Opkod **b** govori procesoru da skoči unaprijed za #broj bajtova. U ovom slučaju govorimo procesoru da skoči 0 bajtova unaprijed i procesor će samo opet skočiti na ovu istu instrukciju i biti zaglavljen u beskonačnoj petlji. Tada će naš emulator ostati na toj liniji i moguće ga je u tom trenutku pauzirati ili pregledavati memoriju ili što god jer je emulator stao točno na instrukciji na kojoj smo htjeli. Ovo je osnovni način kako je moguće implementirati kontrolu toka unutar ovog *debuggera*. Sada, ako primjerice želimo implementirati funkciju koja izvršava emulator instrukciju po instrukciju, dodamo naš "breakpoint" na adresu 4 bajta dalje (u ARM arhitekturi sve instrukcije su duge 4 bajta) i samo vratimo staru instrukciju, na mjesto trenutnog "breakpointa" tj. beskonačne petlje. Da bi se moglo nastavi s izvršavanjem tj. vratiti staru instrukciju, moraju se pratiti sve zamijenjene instrukcije. U ovoj je implementaciji za to korišten Python riječnik.

4.2.1. Pojašnjenje primjera koda

Ovdje je uključen samo isječak koda 4.2. koji direktno implementira GDB komandu i izostavljene su *import* naredbe i implementacija QEMUMonitorProtocol klase.

QEMUMonitorProtocol klasa služi za komuniciranje s QEMU emulatorom preko QMP-a. QMP koristi JSON format kako bi prenio komandu koju se želi izvršiti. Ti JSON podatci se prenose direktno preko TCP-a do vrata na kojima čeka QMP slušatelj unutar emulatora. Kako bi se osiguralo da je QEMU pokrenut s QMP slušateljem, potrebno je dodati zastavicu **-qmp tcp:localhost:4444,server,nowait** prilikom pokretanja QEMU emulatora. Ti JSON podatci generalno definiraju QMP komandu koju se želi izvršiti i potencijalne argumente koje se želi poslati skupa s komandom. Popis svih QMP komandi vidljiv je na službenim stranicama QEMU-a [12].

Za komunikaciju s GDB-om koristi se Python gdb paket koji pruža službenu potporu za komunikaciju s GDB-om. Kako bi se definirala nova komanda, potrebno je napraviti klasu koja nasljeđuje klasu gdb.Command i definirati konstruktor i metodu *invoke* kao u primjeru. U konstruktoru se definira ime komande koja se implementira, u ovom

slučaju **plc-break**. U metodi *invoke* definira se željeni efekt kada se komanda izvrši. U ovom slučaju želi se pročitati trenutna instrukcija na memorijskoj lokaciji koja je dana kao argument u metodu *invoke* kako bi ju se moglo spremiti i zatim na mjesto te instrukcije upisati beskonačnu petlju. Kako bi se moglo čitati memoriju procesa, prvo treba dobiti referencu na objekt procesa unutar GDB-a, što se dobiva tako da se uzme prvi član polja kojeg vraća funkcija *gdb.inferiors()* jer ona vraća polje svih trenutno otvorenih procesa u GDB-u. Nakon toga može se pročitati instrukcija na toj memorijskoj lokaciji sa read_memory() i spremiti se u naš rječnik svih *breakpointova*. Zatim se na to mjesto s funkcijom write_memory() može upisati naša beskonačna petlja kodirana kao 4 bajtni heksadekadski broj. Treba primijetiti da je kodirani broj pretvoren u *little endian* format iz kodiranja napisanog iznad jer iako ARMv7 radi u *big endian* načinu rada, ovaj se emulator mora pokretati u *little endian* načinu rada.

Na ovaj način, korištenjem QMP komandi i izmjenom i čitanjem memorije gdje je potrebno, moguće je implementirati komande koje idu instrukciju po instrukciju, ispisuju stanje stoga, pregledavaju stanje registara ili većinu drugih komandi koje su moguće u normalnom *debugging* okruženju. Jedno trenutno ograničenje jest to da kod koji implementira alociranje memorije na gomili ne radi unutar emulatora, a nije ni reverzano točno kako algoritam alokacije i dealokacije radi pa je prikaz gomile na smislen način trenutno nemoguć (memorija se može gledati, ali tamo se ništa ne alocira).

5. Mrežni stog

Koristeći tehnike koje su opisane dosad, pokušali smo saznati što je više moguće o mrežnom stogu operacijskog sustava ADONIS na primjeru PLC-a S7-1200. Također, valja primijetiti da se opisane tehnike mogu koristiti za reverzanje bilo kojeg aspekta operacijskog sustava ADONIS ili PLC-a S7-1200. Mrežni stog je širok pojam i u ovom će se poglavlju opisati mrežne sposobnosti koje su pronađene i povezati biblioteke otvorenog ili zatvorenog tipa koda. Također, ako postoji dio koda za koji vjerujemo da nije dio neke od prije poznate biblioteke, nego ga je dodao Siemensov tim postojat će napomena. Važno je napomenuti da je emulator trenutno u razvoj nije još dovoljno dobar da možemo zaključiti jako puno o mrežnom stogu iz njega. Trenutno stanje emulatora je takvo da gotovo završi proces pokretanja operacijskog sustava, ali ne može alocirati memoriju za gomilu. Zbog ovoga emulator ne uspijeva doći do većine koda koji ima veze s mrežnim stogom. Ipak, emulator je za ovo istraživanje bio jako bitan za razumijevanje općenitog načina rada operacijskog sustava ADONIS, a sigurno će budućim istraživačima biti vrlo koristan. Nažalost, trenutno još nije na razini ovog zadataka pa je većina spoznaja o mrežnom stogu dobivena statičkom analizom i istraživanjem prijašnjih radova.

5.1. TCP/IP stog

Kako je opisano u prijašnjem poglavlju o statičkoj analizi, pomoću *stringova* moguće je, ako se pogledaju *stringovi* starije verzije *firmwarea*, među *stringovima* koji se koriste za *debugging* vidjeti *stringove* koji referenciraju InterNiche TCP/IP stog. U tom je stogu 2021. godine otkriven skup od 14 ranjivosti nazvan INFRA:HALT[16]. Stara verzija *firmwarea* u kojima postoje reference direktno na InterNichev stog verzija je 4.1.2 i dolazi iz vremena prije 2021. kada su te ranjivosti popravljene. U novijoj verziji 4.5.2 koja je najviše proučavana u sklopu rada ne postoji tako direktna referenca na InterNiche datoteke,

ali ako se pogledaju opisnici za dnevnički sustav, moguće je pronaći opisnik s imenom [TCIP_INICHE] koji indicira da se i dalje koristi InterNiche mrežni stog, ali nakon ažuriranja koje je otklonilo ranjivosti, više nema tih *debug stringova*. Također, u Siemensovu službenom izvješću [14] spominju se samo 4 uređaja na koje ranjivosti utječu i svi su uređaji dio SENTRON linije uređaja koja je zamišljena kao linija sigurnosnih uređaja. Moguće je da ni jedna ranjivost koja je nađena u InterNiche stogu ne utječe na PLC jer je konkretno mjesto gdje se koristi taj stog kao podloga za Siemensov vlasnički S7Comm protokol koji se koristi za komunikaciju s radnom stanicom i moguće da način na koji koriste stog nije podložan tim ranjivostima.

Što se tiče samog TCP/IP stoga, postoje i drugi opisnici za dnevnik vezani za TCP/IP, a to su [TCIP_UPPER], [TCIP_OBSD] i [TCIP_MEM]. Interesantan je opisnik [TCIP_OBSD] jer sugerira korištenje nekog koda iz OpenBSD operacijskog sustava. Mi nismo pronašli kod koji koristi taj opisnik, ali ako postoji, moguće da je vezan za implementaciju *socket* sučelja za koje InterNiche stog ima samo minimalnu implementaciju.

5.2. S7Comm i S7CommPlus Protokoli

S7Comm protokol je vlasnički protokol koji Siemens koristi za komunikaciju s radnom stanicom, a stanica može konfigurirati i upravljati PLC-om. To se upravljanje na radnoj stanici obavlja većinom u okviru programa TIAPortal, također vlasničkog programa tvrtke Siemens. S obzirom na to da se protokol koristi za komunikaciju s konfiguracijskim portalom, sigurnost ovog protokola jako je važna. Ta konfiguracija uključuje paljenje i gašenje PLC-a, mijenjanje programa koji se pokreće na PLC-u, mijenjanje stanja memorije PLC-a i slično. Stara verzija S7Comm protokola, korištena na S7-300 i S7-400 PLC-ovima, bila je nekriptirana i zato je u modernim verzijama PLC-ova S7-1200 i S7-1500 zamijenjena sa S7CommPlus protokolom. S7CommPlus ima nekoliko prije pronađenih ranjivosti iako je napredak naspram S7Comm barem u tome da je kriptiran. Lei et al. [7] su 2017. godine reverzali velik dio protokola i njihovo su istraživanje 2019. godine iskoristili Biham et al. [4] kako bi implementirali lažnu stanicu za konfiguraciju PLC-a. Ta lažna stanica iskorištava grešku u protokolu koja je u tome da se TIA ne mora uspješno identificirati PLC-u, samo se PLC mora identificirati TIA portalu. Takva postava protokola omogućila je stvaranje lažne stanice preko koje je moguće zaustaviti PLC

i čak promijeniti program koji se vrti na PLC-u. Ovaj problem može se riješiti tako da se pristup PLC-u zaštiti lozinkom, ali postoje brojna postrojenja koja lozinke ne koriste ili im je sigurnost tih lozinki jako loša. Također, 2021. Hui et al. [6] pokazali su greške u anti-replay mehanizmu protokola koji omogućava slanje lažnih paketa i time između ostalog omogućava gašenje PLC-a. Ove greške postoje u *firmwareima* koji su verzija 4 i vjerojatno postoji određeni broj postrojenja koja još uvijek koriste starije verzije tih *firmwarea* i ne koriste dobre lozinke (ili ne koriste lozinke uopće), zbog čega su njihovi PLC-ovi ranjivi.

5.3. ACE

ACE (engl. *Adaptive Communication Environment*) je radni okvir otvorenog tipa koda za c++ koji koristi objektno orijentirani dizajn kako bi olakšao mrežno programiranje. Otkriveno je u sklopu rada da PLC koristi ACE pomoću gledanja *stringova* koji postoje u opisnicima c++ klasa. Također iz pregleda koda ACE razvojnog okvira uključenog u *firmware* vidi se da je kod izmijenjen kako bi mogao funkcionirati sa ADONIS operacijskim sustavom, primjerice izmijenjena je logika za upisivanje u dnevnik. ACE je jako velik i nije previše vremena provedeno proučavajući sve izmjene koje su dodane u kod, ali ako je potrebno bolje razumijevanje funkcioniranja operacijskog sustava, moguće je koristiti te izmjene kako bi se nešto zaključilo budući da je originalni ACE otvorenog tipa.

5.4. Web Server

Web server je jedna od velikih mogućnosti koje PLC podržava. Web server je zamišljen kao zamjena za TIAPortal koja se može pokrenuti na bilo kojem uređaju. Web server ima manje mogućnosti od samog TIAPortala, ali i dalje ima dovoljno mogućnosti da pristup serveru mora biti potupuno siguran. Primjerice, moguće je zaustaviti i pokrenuti PLC, mijenjati stanje varijabli u memoriji PLC-a, pregledavati dijagnostiku o radu i slično. Osnovni kod web servera koristi MiniWeb kao bazu. To se može zaključili preko opisnika za dnevnik koji referenciraju MiniWeb ([MINIWEB_CORE], [MINIWEB_PROACTORIF], [MINIWEB_S7WEB], [MINIWEB_AWP] i [MINIWEB_DIAGBUFF]). Najzanimljiviji od tih opisnika je [MINIWEB_S7WEB] jer se taj opisnik koristi za dijelove web servera koje je implementirao Siemens. Ti dijelovi dodaju nove funkcionalnosti koje MiniWeb nema,

kao što je komunikacija preko JSON RPC protokola, ali u tim dijelovima je implementirana i dodatna funkcionalnost specifična za PLC kao što je pisanje i čitanje varijabli u memoriji PLC-a. U *firmwareu* se koristi *string* S7PWEB za tu dodatnu funkcionalnost, za razliku od opisnika koji koristi *string* S7WEB. Ako se potraži S7PWEB, moguće je vidjeti *stringove* koji referenciraju datoteke izvornog koda, slično kako je opisano u sekciji o dobivanju *stringova* iz *assertova*.

Stranice koje web server prikazuje implementirane su na jedan od dva načina. Prvi način je korištenje normalnih HTML datoteka unutar kojih se piše JavaScript, a drugi način je korištenje *MiniWeb Scripting Languagea (MWSL)*. Stranice s kojima web server dođe, prilično je lagano izvući van iz *firmwarea* jer su unutra pohranjene kao običan tekst.

```
<a href="/basic/Portal.mwsl?PriNav=Bgz">Index<a class="Header_Link" href="/MiniWebCA_Cer.crt">
<div>
$('#restore_pw_field').prop('disabled', false); webpw: $('#restore_pw_field').val(),
$('#restore_pw_field').prop('disabled', true);
submit_form();
return state_load_config;<meta http-equiv="x-ua-compatible" content="ie=edge">
<meta http-equiv="content-type" content="text/html; charset=utf-8">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="expires" content="-1">
<meta name="viewport" content="width=device-width, initial-scale=1, maximum-scale=1.0">
<link rel="stylesheet" type="text/css" href="/CSS/S7Web.css" media="all">
<link rel="stylesheet" type="text/css" href="/CSS/S7WebPrint.css" media="print">
<link rel="shortcut icon" href="/favicon.ico">
<script src="/Scripts/jquery.min.js" type="text/javascript"></script>
<script src="/Scripts/popup.js" type="text/javascript"></script>No invalid,id="varstate_newvar" class="active" disabled<a href="/basic/Portal.mwsl?PriNav=Index"><img
src="/Icons/HOME.gif" alt="Home"></a>File fail-safe-markerDir
```

Slika 5.1. Primjer koda iz početne stranice web servera

Iz primjera koda koje je izvučen iz *firmwarea* može se dobiti nekoliko zaključaka. Prvi je da je moguće pisati i HTML i MWSL i da HTML poveznice mogu voditi do stranica pisanih u MWSL-u. Drugi je da stranica koristi jQuery. Konkretno se u verziji *firmwarea* 4.5.2 koristi jQuery 3.6.6 koji u trenutku pisanja nema poznatih ranjivosti.

```
1 <MWSL>
2     WriteXMLData("<EXTERNAL SRC=\"" + GetVar("XML", "URL") + "\"/>",
3     "<TEMPLATES><EXTERNAL SRC=\"" +
4     GetVar("TEMPLATE", "URL") + "\"/></TEMPLATES>");
5 </MWSL>
```

Listing 5..1: Primjer MWSL programa

Kao što je vidljivo na primjeru iz 5.4., MWSL izgleda slično kao PHP i koristi XML kako bi komunicirao s PLC-om. MWSL omogućava priličnu kontrolu nad PLC-om i u tim stranicama pogotovo treba biti oprezan jer one mogu pristupati memoriji i stopirati

PLC. MWSL se nakon što je napisan provede kroz kompajler i nakon toga pokreće se direktno na PLC-u.

Web server također dopušta pisanje vlastitih web stranica, a ne samo korištenje postojećih. Moguće je da u nekim postrojenjima te web stranice imaju greške koje su unijeli programeri stranica, koje mogu utjecati na sigurnost samog sustava. Što se tiče sigurnosti, bitno je napomenuti i to da PLC ne prisiljava korištenje https protokola iako za to ima mogućnost pa je promet između servera i korisničkog računala po početnim postavkama nekriptiran.

5.5. Ostale mrežne sposobnosti

PLC podržava i implementira još puno protokola osim onih navedenih gore. Svi protokoli koje podržava navedeni su u tablici 5.5.

Protokol	Firmware verzija
TCP	1.0
ISO on TCP (RFC 1006)	1.0
PROFINET RT	2.0
PROFINET IO Device	4.0
S7Comm	1.0
Modbus TCP	2.1
HTTP	2.0
HTTPS	2.0
SNMP	2.0
LLDP	2.0
DCP	2.0
NTP	2.0
ARP	2.0

U ovom radu spomenute su samo neke od ovih mrežnih sposobnosti, naročito zato što ih ima previše za proučavanje odjednom. Fokus je bio stavljen na one protokole koji imaju najviše potencijala da izazovu sigurnosne greške ili koji su već prije izazvali sigurnosne greške.

6. Zaključak

Tehnološki napredak često znači tehnološke komplikacije. Kako sustavi industrijskog upravljanja postaju sve kompliciraniji, tako i PLC-ovi koji njima upravljaju moraju postati sve kompliciraniji. U ovom radu pokušali smo reverzati mrežni stog operacijskog sustava ADONIS koji je izrađen specifično za S7-1200 i S7-1500 PLC-ove. Također smo pokazali našu metodologiju tog reverzanju i kako smo izradili alate koji pomažu u reverzanju. Kako sustavi postaju kompliciraniji, a proizvođači tih sustava i dalje drže sve dijelove jako zatvorene, alati poput emulatora, *debuggera* i raznih skripti postat će sve važniji. Za statičku analizu pokazali smo skriptu koja automatski detektira funkcije uključene u podsustave unutar ADONIS-a. S druge strane, za dinamičku analizu smo pokazali QEMU emulator koji pokreće kod operacijskog sustava i pokazali smo kako je, čak i kada *debugger* ne radi, moguće ručno napraviti puno funkcionalnosti koje očekujemo od *debuggera*. Nakon toga smo primijenili te alate na reverzanje mrežnog stoga ADONIS-a i S7-1200 PLC-a. Zaključili smo da na mnogo mjesta postoji puno prostora za napredak jer su ne tako davno postojale kritične ranjivosti u određenim dijelovima mrežnog stoga. Naročito su ranjivosti postojale u TCP/IP stogu i S7CommPlus protokolu koji su vlasnički i zatvoreni za istraživanje te ako postoji još ranjivosti u njima, istraživači moraju uložiti puno vremena kako bi ih pronašli. Također, sam web server interesantna je meta za istraživanje koja još nije jako istraživana, a omogućava potencijalno veliku kontrolu nad PLC-om.

Bibliografija

[1] Ali Abbasi, Tobias Scharnowski i Thorsten Holz. "Doors of durin: The veiled gate to siemens S7 silicon". *BlackHat Europe* (2019.).

[2] ARM. `https://www.arm.com/`. [stranica posjećena: svibanj 2024.]

[3] Marie Baezner i Patrice Robin. *Stuxnet*. Teh. izv. ETH Zurich, 2017.

[4] Eli Biham i dr. "Rogue7: Rogue engineering-station attacks on s7 simatic plcs". *Black Hat USA* 2019 (2019.).

[5] Delimar Danko. `https://github.com/D4ntae/s7-plc-qemu`. [stranica posjećena: svibanj 2024.]

[6] Henry Hui, Kieran McLaughlin i Sakir Sezer. "Vulnerability analysis of S7 PLCs: Manipulating the security mechanism". *International Journal of Critical Infrastructure Protection* 35 (2021.), str. 100470.

[7] Cheng Lei, Li Donghong i Ma Liang. "The spear to break the security wall of S7CommPlus". *Blackhat USA* 17 (2017.), str. 1–12.

[8] NSA. `https://ghidra-sre.org/`. [stranica posjećena: svibanj 2024.]

[9] Pantina Ardian. `https://www.zemris.fer.hr/~sgros/students/diploma_thesis/pantina_ardian_diplomski.pdf`. [stranica posjećena: svibanj 2024.]

[10] Kovač Petar. "Izvršavanje firmwarea PLC-a korištenjem alata QEMU". *FER Diplomski rad* (2022.).

[11] QEMU. `https://www.qemu.org/`. [stranica posjećena: svibanj 2024.]

[12] QEMU Project. `https://qemu-project.gitlab.io/qemu/interop/qemu-qmp-ref.html`. [stranica posjećena: svibanj 2024.]

[13] Siemens official website. `https://www.siemens.com/global/en/products/automation/systems/industrial/plc.html`. [službena stranica siamatic PLC-ova; stranica posjećena: svibanj 2024.]

[14] SiemensCERT. https://cert-portal.siemens.com/productcert/pdf/ssa-789208.pdf. [stranica posjećena: svibanj 2024.]

[15] Xilinx. https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/821395464/QEMU+User+Documentation. [stranica posjećena: svibanj 2024.]

[16] Žorž Željka. https://www.helpnetsecurity.com/2021/08/04/vulnerabilities-nichestack/. [stranica posjećena: svibanj 2024.]

Sažetak

Reverzanje mrežnog stoga operacijskog sustava ADONIS

Danko Delimar

U ovom radu pokazat ćemo kako smo izradili specifične alate za reverzanje ADONIS operacijskog sustava i kako smo ih primijenili na reverzanje mrežnog stoga ADONIS-a i S7-1200 PLC-a. Pokazat, ćemo skriptu u Ghidri koja automatski detektira funkcije koje pripadaju određenom podsustavu OS-a. Također, pokazat ćemo kako smo riješili problem da ne možemo koristiti *debugger* nad emulatorom koji je napravljen specifično za S7-1200 PLC. Na kraju, pokazat ćemo naše spoznaje o mrežnom stogu ADONIS-a i S7-1200 PLC-a.

Ključne riječi: reverzanje, ADONIS, PLC, Ghidra, QEMU

Abstract

Uncomputable Computablity

Danko Delimar

In this paper we will show how we built specific tools to reverse engineering the ADONIS operating system and how we used those tools to reverse engineer the network stack of ADONIS and S7-1200 PLC. We will show a script made in Ghidra that automatically detects functions that belong to a specific subsystem of the OS. We will also show how we solved the problem of not being able to use a debugger on an emulator made specifically for the S7-1200 PLC. In the end we will show what we found out about the network stack of ADONIS and the S7-1200 PLC.

Keywords: reverse engineering, ADONIS, PLC, Ghidra, QEMU

Privitak A: Kod skripte za detekciju podsustava

```java
import ghidra.app.script.GhidraScript;
import ghidra.program.model.address.Address;
import ghidra.program.model.address.GenericAddress;
import ghidra.program.model.lang.Register;
import ghidra.program.model.listing.Function;
import ghidra.program.model.listing.FunctionManager;
import ghidra.program.model.listing.Instruction;
import ghidra.program.model.listing.Listing;
import ghidra.program.model.scalar.Scalar;
import ghidra.program.model.symbol.Reference;
import ghidra.program.model.symbol.ReferenceIterator;
import ghidra.program.model.symbol.ReferenceManager;
import ghidra.util.task.TaskMonitor;

import javax.swing.JFrame;
import javax.swing.JTable;
import javax.swing.WindowConstants;
import javax.swing.JScrollPane;
import javax.swing.table.AbstractTableModel;

import java.util.ArrayList;
import java.util.List;

public class FindReferencesScript extends GhidraScript {

  private ArrayList<Address> error_addrs = new ArrayList<>();

    @Override
    public void run() throws Exception {
```

```
30        String loggerVtable = "01e022a8";
31        String constant = askString("Enter a constant", "Enter the
    constant in hexadecimal form (0xab): ");
32        Address targetAddress = toAddr(loggerVtable);
33
34        getReferencesToAddress(targetAddress, constant);
35    }
36
37
38    private Address next(Address addr) {
39      return addr.add(4);
40    }
41
42    private Address previous(Address addr) {
43      return addr.subtract(4);
44    }
45
46    private Address sendBack(Address addr, int num) {
47      return addr.subtract(4 * num);
48    }
49
50    private Instruction getInstructionAtAddress(Address address) {
51      // Get the listing object from the current program
52      // Listing is an object to control all code level constructs (
    https://ghidra.re/ghidra_docs/api/ghidra/program/model/listing/
    Listing.html)
53      Listing listing = currentProgram.getListing();
54
55      // Get the instruction at the specified address
56        // Instruction object docs (https://ghidra.re/ghidra_docs/api/
    ghidra/program/model/listing/Instruction.html)
57      Instruction inst = listing.getInstructionAt(address);
58
59      return inst;
60    }
61
62    private String getConstantFromLoggerFunctionCall(Reference ref) {
63      // Get instruction referencing logger vtable
64      Address from_address = ref.getFromAddress();
```

```
65
66      Instruction instruction = getInstructionAtAddress(from_address);
67
68      if (instruction.getMnemonicString().startsWith("mov")) {
69        instruction = getInstructionAtAddress(next(from_address));
70        return "mov";
71      }
72
73      // Get register into which the logger vtable is loaded
74        // Register docs (https://ghidra.re/ghidra_docs/api/ghidra/
   program/model/lang/Register.html)
75      Register reg = instruction.getRegister(0);
76      println("Reg: " + reg.getName());
77      String reg_name = reg.getName();
78
79      Address temp_addr = next(from_address);
80
81        // Register from which the function is called. From here we're
    gonna look for the call instruction to find the constant.
82      Register call_reg = null;
83      Instruction next_instruction = null;
84      while (true) {
85        next_instruction = getInstructionAtAddress(temp_addr);
86
87        if (!next_instruction.getMnemonicString().equals("ldr")) {
88          temp_addr = next(temp_addr);
89          continue;
90        }
91
92        // Check to follow branches.
93        if (next_instruction.getMnemonicString().equals("b") &&
   next_instruction.getOpObjects(0)[0] instanceof GenericAddress) {
94          temp_addr = (GenericAddress) next_instruction.getOpObjects
   (0)[0];
95          continue;
96        }
97
98        try {
99            if (!(next_instruction.getOpObjects(1)[0] instanceof
```

```
  Register)) {
100             temp_addr = next(temp_addr);
101             continue;
102           }
103       } catch (Exception e) {
104         temp_addr = next(temp_addr);
105         continue;
106       }
107 
108 
109       if (!((Register)next_instruction.getOpObjects(1)[0]).getName()
  .equals(reg_name)) {
110         temp_addr = next(temp_addr);
111         continue;
112       }
113       call_reg = next_instruction.getRegister(0);
114       break;
115     }
116     // Used to track jumps when looking back
117     Address backJump = null;
118     Address jumpTo = null;
119     // Number of instructions to look ahead
120     int c = 24;
121     temp_addr = next(temp_addr);
122     println("CR1: " + call_reg.toString());
123     //Check if call_reg was copied
124     println("temp1: " + temp_addr.toString());
125     while (c != 0) {
126       try {
127         next_instruction = getInstructionAtAddress(temp_addr);
128 
129           if (next_instruction.getMnemonicString().equals("b") &&
  next_instruction.getOpObjects(0)[0] instanceof GenericAddress) {
130             backJump = temp_addr;
131             jumpTo = (GenericAddress) next_instruction.getOpObjects
  (0)[0];
132             temp_addr = (GenericAddress) next_instruction.
  getOpObjects(0)[0];
133             continue;
```

```
134           }
135 
136           if (next_instruction.getMnemonicString().startsWith("b")
    && next_instruction.getRegister(0).getName().equals(call_reg.
    getName()) ) {
137             break;
138           }
139 
140         if (next_instruction.getMnemonicString().equals("cpy")) {
141           if (next_instruction.getRegister(1).getName().equals(
    call_reg.getName())) {
142             call_reg = next_instruction.getRegister(0);
143             break;
144           }
145         }
146         temp_addr = next(temp_addr);
147         c -= 1;
148       } catch (Exception e) {
149         temp_addr = next(temp_addr);
150         c -= 1;
151       }
152 
153     }
154 
155 
156     // Number of instructions to look ahead
157     // Send back to the first instruction where CR was defined
158     temp_addr = next(sendBack(temp_addr, 24 - c));
159     c = 24;
160     // Check if call_reg was stored on the stack
161     println("CR2: " + call_reg.toString());
162     println("temp2: " + temp_addr.toString());
163 
164     try {
165       while (c != 0) {
166           next_instruction = getInstructionAtAddress(temp_addr);
167 
168           if (next_instruction.getMnemonicString().equals("b") &&
    next_instruction.getOpObjects(0)[0] instanceof GenericAddress) {
```

```
169             backJump = temp_addr;
170             jumpTo = (GenericAddress) next_instruction.getOpObjects
    (0)[0];
171             temp_addr = (GenericAddress) next_instruction.
    getOpObjects(0)[0];
172             continue;
173           }
174 
175             if (next_instruction.getMnemonicString().startsWith("b")
     && next_instruction.getRegister(0).getName().equals(call_reg.
    getName()) ) {
176             break;
177           }
178           if (next_instruction.getMnemonicString().equals("str")) {
179             if (next_instruction.getRegister(0).getName().equals(
    call_reg.getName())) {
180               Register store_reg = (Register) next_instruction.
    getOpObjects(1)[0];
181               Scalar offset = (Scalar) next_instruction.getOpObjects
    (1)[1];
182               // Find where the register is popped of the stack
183               while (true) {
184                 if (next_instruction.getMnemonicString().equals("ldr
    ")) {
185                       if (((Register)next_instruction.getOpObjects
    (1)[0]).getName().equals(store_reg.getName()) && ((Scalar)
    next_instruction.getOpObjects(1)[1]).getValue() == offset.getValue
    ()) {
186                         call_reg = next_instruction.getRegister(0);
187                         break;
188                       }
189                 }
190                 temp_addr = next(temp_addr);
191               next_instruction = getInstructionAtAddress(temp_addr);
192               }
193             }
194           }
195           temp_addr = next(temp_addr);
196           c -= 1;
```

```
197         }
198
199       } catch (Exception e) {
200
201       }
202       println("CR3: " + call_reg.toString());
203
204       temp_addr = next(sendBack(temp_addr, 24 - c));
205       println("temp3: " + temp_addr.toString());
206
207       Address call_addr = null;
208       // Instruction from which to search backwards for the first
    change in r0, cuz thats the constant.
209
210       while (true) {
211         next_instruction = getInstructionAtAddress(temp_addr);
212         if (next_instruction.getMnemonicString().equals("b") &&
    next_instruction.getOpObjects(0)[0] instanceof GenericAddress) {
213           backJump = temp_addr;
214           jumpTo = (GenericAddress) next_instruction.getOpObjects(0)
    [0];
215           temp_addr = (GenericAddress) next_instruction.getOpObjects
    (0)[0];
216           continue;
217         }
218
219         // Ret 3
220       if (!(next_instruction.getMnemonicString().startsWith("bl") ||
    next_instruction.getMnemonicString().startsWith("bx"))) {
221         temp_addr = next(temp_addr);
222         continue;
223       }
224
225         try {
226           if (next_instruction.getOpObjects(0)[0] instanceof
    GenericAddress) {
227             temp_addr = next(temp_addr);
228             continue;
229           }
```

```
          if (next_instruction.getRegister(0) == null) {
            temp_addr = next(temp_addr);
            continue;
          }

        if (!next_instruction.getRegister(0).getName().equals(
call_reg.getName())) {
          temp_addr = next(temp_addr);
          continue;
        }
      } catch (Exception e) {
        temp_addr = next(temp_addr);
        continue;
      }

      call_addr = next_instruction.getAddress();
      break;
    }

    String con = "";
    println("CA1: " + call_addr.toString());
    temp_addr = previous(call_addr);
    while (true) {
      if (backJump != null && temp_addr.toString().equals(jumpTo.
toString())) {
        temp_addr = backJump;
      }

      next_instruction = getInstructionAtAddress(temp_addr);


      if (next_instruction.getOpObjects(0)[0] instanceof
GenericAddress) {
        temp_addr = previous(temp_addr);
        continue;
      }

      if (next_instruction.getRegister(0) == null) {
        temp_addr = previous(temp_addr);
```

```
266           continue;
267         }
268 
269         if (!next_instruction.getRegister(0).getName().equals("r0")) {
270           temp_addr = previous(temp_addr);
271           continue;
272         }
273 
274         con = next_instruction.getOpObjects(1)[0].toString();
275         return con;
276       }
277 
278 
279 
280     }
281 
282 
283     private Function getFunctionFromAddress(Address addr) {
284 
285         FunctionManager functionManager = currentProgram.
    getFunctionManager();
286         Function function = functionManager.getFunctionContaining(addr
    );
287         return function;
288     }
289 
290     private void getReferencesToAddress(Address addr, String con) {
291     ReferenceManager refMgr = currentProgram.getReferenceManager();
292     ReferenceIterator refs = refMgr.getReferencesTo(addr);
293     int c = 0;
294     for (Reference ref : refs) {
295       println("Ref: " + ref.toString());
296       String res = getConstantFromLoggerFunctionCall(ref);
297       println("---- Res: " + res);
298       println("Residual: " + c);
299       c++;
300     }
301 
302 
```

```
303     }
304 
305     private class TableEntry {
306         private Address address;
307         private Function function;
308 
309         public TableEntry(Address address, Function function) {
310             this.address = address;
311             this.function = function;
312         }
313 
314         public Address getAddress() {
315             return address;
316         }
317 
318         public String getFunctionName() {
319             return function != null ? function.getName() : "<no
    function>";
320         }
321     }
322 
323     private class CustomTableModel extends AbstractTableModel {
324         private List<TableEntry> entries;
325 
326         public CustomTableModel(List<TableEntry> entries) {
327             this.entries = entries;
328         }
329 
330         @Override
331         public int getColumnCount() {
332             return 2;
333         }
334 
335         @Override
336         public int getRowCount() {
337             return entries.size();
338         }
339 
340         @Override
```

```
341         public Object getValueAt(int rowIndex, int columnIndex) {
342             TableEntry entry = entries.get(rowIndex);
343             if (columnIndex == 0) {
344                 return entry.getAddress().toString();
345             } else if (columnIndex == 1) {
346                 return entry.getFunctionName();
347             } else {
348                 return "unknown";
349             }
350         }
351 
352         @Override
353         public String getColumnName(int column) {
354             if (column == 0) {
355                 return "Address";
356             } else if (column == 1) {
357                 return "Function";
358             } else {
359                 return "";
360             }
361         }
362     }
363 }
```