VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

POUŽITIE STATICKEJ ANALÝZY PRE DETEKCIU CHÝB V OBSLUHE SIGNÁLOV

A STATIC ANALYSIS TOOL DETECTING BUGS IN SIGNAL HANDLERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **DANIEL KOZOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE **Dr. Ing. PETR PERINGER,**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Kozovský Daniel**
Program: Informační technologie
Název: **Použití statické analýzy pro detekci chyb v obsluze signálů**
A Static Analysis Tool Detecting Bugs in Signal Handlers
Kategorie: Formální verifikace

Zadání:

1. Prostudujte problematiku signálů používaných pro komunikaci mezi procesy (IPC) na systémech kompatibilních s normou POSIX. Zaměřte se na pravidla pro bezpečné provedení obsluhy signálů. Seznamte se s interní reprezentací kódu (napsaného v C a C++) používaného v překladači GCC a s rozhraním pro zásuvné moduly GCC.
2. Navrhněte zásuvný modul pro GCC, který ověřuje vybraná pravidla pro bezpečnou obsluhu signálů v C/C++ kódu. Návrh řádně zdokumentujte.
3. Implementujte modul v C++ a experimentálně ověřte jeho funkčnost na dostatečně velkém vzorku volně dostupných zdrojových kódů v C/C++ (cca 1 milion řádků).
4. Zhodnoťte kvalitu získaných výsledků a použitelnost tohoto nástroje pro vývoj programů v C a C++.

Literatura:

- Stevens R., Rago S.: *Advanced Programming in the Unix Environment*. Second Ed., Pearson Education, 2005
- Další dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění prvních dvou bodů zadání.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Peringer Petr, Dr. Ing.**
Konzultant: Dudka Kamil, Ing., RedHatCZ
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Táto práca sa zaoberá zásuvným modulom *csigsafe* pre prekladač GCC. Používa statickú analýzu programov na odhalenie chýb v obsluhe signálov podla normy POSIX. Tento nástroj analyzuje zdrojové súbory v jazyku C a C++. Tento analyzátor je vytvorený pre firmu Red Hat, ktorá ho používa na testovanie sRPM balíkov určených do ich Linuxových distribúcii. Nástroj bol testovaný na vzorku 37 projektoch s volne šíriteľnými zdrojmi. Z testovania sa ukázala užitočnosť nástroja pri vyhľadávaní chýb spojených s porušením pravidiel na správnu obsluhu signálov.

Abstract

This work is about the plugin *csigsafe* for the GCC compiler. It uses static code analysis to detect bugs in signal handlers according the POSIX norm. This tool analyzes the source files written in C and C ++. This analyzer is created for the Red Hat, which uses it to test sRPM packages used in their Linux distributions. The tool has been tested on a sample of 37 Open Source projects. Testing has shown the utility of the tool to search for errors associated with violation of rules for proper signal handling.

Kľúčové slová

Signál, Obsluha signálov, GCC, Zásuvný modul, Statická analýza

Keywords

Signal, Signal handler, GCC, Plug-in, Static analysis

Citácia

KOZOVSKÝ, Daniel. *Použitie statickej analýzy pre detekciu chýb v obsluhe signálov*. Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dr. Ing. Petr Peringer,

Použitie statickej analýzy pre detekciu chýb v obsluhe signálov

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením Dr. Ing. Petra Peringera. Ďalšie informácie mi poskytol konzultant z firmy Red Hat Ing. Kamil Dudka. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Daniel Kozovský
13. mája 2019

Poďakovanie

Chcel by som sa poďakovať vedúcemu práce Dr. Ing. Petrovi Peringerovi, ako aj konzultantovi z firmy Red Hat, Ing. Kamilovi Dudkovi za ich pomoc pri vypracovaní tejto práce.

Obsah

Kapitola 1

Úvod

Skoro každá zložitejšia aplikácia používa na svoje fungovanie signály. Signály sú asynchrónna komunikácia, a preto je nutné dbať na určité zásady pri programovaní ich spracovania. Keďže programy sú zložité a s chybami, ručné hľadanie týchto chýb je náročné a vyžaduje automatizáciu. Okrem toho chyby spojené s nesprávnou obsluhou signálov sa neprejavujú deterministicky, preto nie je zaručené, že budú odhalené pri testovaní. Z toho dôvodu je vhodnejšie sa snažiť tieto chyby objaviť pomocou statickej analýzy. Preto sa táto práca zaoberá nástrojom, ktorý by mal pomocou statickej analýzy tieto chyby odhaliť.

Nástroj na svoju činnosť používa statickú analýzu, to znamená že prechádza zdrojové kódy programu, a prostredníctvom nich sa snaží nájsť chybu, prípadne rizikový kód, ktorý by mohol za istých okolností spôsobiť zlyhanie systému. Analyzátor bude realizovaný ako zásuvný modul do prekladača GCC. Modul dostal meno csigsafe, a primárne bude používaný vo firme Red Hat, pre ktorú je aj realizovaný. Keďže firma Red Hat používa prekladač GCC primárne, tak pre ňu bolo vhodné aby bol zásuvný modul postavený práve na tomto prekladači. Prekladač pre analyzátor nahrádza lexikálnu syntaktickú a sémantickú analýzu jazyka C, respektíve C++. Nástroj *csigsafe* by mal uľahčiť hľadanie chýb spojených s nesprávnou obsluhou signálov. Keďže bude postavený na prekladači GCC, tak bude možné jednoducho ho použiť spolu s nástrojom csmock, ktorý slúži na analýzu sRPM balíkov používaných v linuxových distribúciách firmy Red Hat. Keďže csmock obsahuje zásuvný modul na analýzu sRPM balíkov pomocou prekladača GCC, nie je rozšírenie o analyzátor *csigsafe* vôbec zložité.

Kapitola 2 sa zaoberá prehľadom teórie a súčasne dostupnými analyzátormi, ktoré sa zaoberajú podobným problémom. V kapitole 3 je priblížený návrh modulu a popis algoritmu. Kapitola 4 obsahuje popis implementácie a testovania modulu.

Kapitola 2

Prehľad súčastného stavu

V tejto kapitole je mierne priblížené ako funguje analýza programov a sú v nej spomenuté dostupné riešenia na analyzovanie obsluhy signálov. Ďalej sa táto kapitola zaoberá tým, čo sú to signály a ich spôsoby obsluhy, príklady použitia a možné chyby plynúce zo zlej obsluhy signálov. Na koniec kapitoly je priblížený prekladač GCC a jeho vnútorná štruktúra, ako aj spôsob akým fungujú zásuvné modulu v tomto prekladači.

2.1 Analýza programov

Analýza programu slúži na automatizované odhalenie chýb, ktoré daný program môže obsahovať. Existujú dva druhy analýzy, a to statická a dynamická. Statická analýza funguje tak, že sú prehľadávané zdrojové kódy na rôzne chyby. Dynamická analýza naopak analyzuje spustiteľný program. Statická aj dynamická analýza majú svoje výhody a nevýhody, a preto sa každá hodí na iné veci. Najčastejšie sa na hľadanie chýb v programoch používajú obe tieto analýzy. Použitie nástrojov na analýzu programu nemôže dokázať neprítomnosť chýb v kóde. Pomocou týchto nástrojov je možné iba potvrdenie chýb, nie ich vyvrátenie. Táto práca sa zameriava výhradne na statickú analýzu [6].

2.1.1 Dynamická analýza

Dynamická analýza vyžaduje spustiteľný program, na ktorom vykonáva určité pozorovania. Nevýhoda je to, že vždy pozoruje v jednom čase, len jeden beh programu z pevne danými argumentami. Tento typ analýzy je vhodný na hľadanie chýb, ktoré sa prejavujú deterministicky, čiže pri tých istých podmienkach sa prejaví vždy rovnako [6].

2.1.2 Statická analýza

Statická analýza vyžaduje prístup k zdrojovým kódom programu, ktoré následne analyzuje bez ich spustenia. Preto nie je podstatné či analyzovaný program je, alebo nie je škodlivý. Nevýhodou statickej analýzy je to, že ak sú v kóde použité externé funkcie, čiže funkcie, ktoré nie sú definované v daných zdrojových kódoch, tak analyzátor nevie čo dané funkcie vykonávajú. Narozdiel od dynamickej analýzy, je pomocou statickej analýzy možné analyzovať do určitej miery aj nedeterministické chovanie programu, ako je aj napríklad obsluha signálov [6].

Control Flow analýza

Táto analýza používa algoritmy na zostavenie takzvaného CFG (Control Flow Graph). Ide o orientovaný graf, ktorý sa skladá z blokov a orientovaných prechodov medzi týmito blokmi. Dôležité je identifikovať slučky a volané funkcie, čiže takzvaný Call Graph. Dôsledkom toho sa dozvedáme, ktoré funkcie sú volané a za akých podmienok. Tento druh analýzy sa často používa na optimalizáciu [6]. Nástroj spomínaný v tejto práci používa práve túto analýzu.

2.1.3 Prehľad existujúcich analyzátorov

Dobrý statický analyzátor by mal byť ľahko použiteľný. To znamená, že ich výstup by mal byť ľahko pochopiteľný pre bežného programátora. Tieto nástroje nie sú schopné nahradiť potrebu prezerania kódu človekom, ale môžu pomôcť vyhľadať miesta v kóde, ktoré vyžadujú bližšie preskúmanie skúsenejším programátorom.

Existuje veľké množstvo programov vykonávajúcich statickú analýzu. Dokonca veľké množstvo z nich je voľne dostupných. Väčšina týchto nástrojov sa ale zameriava na detekciu chýb ako sú pamäťová bezpečnosť, neinicializované premenné, nedefinované chovanie a ďalšie typické chyby. Len malé množstvo analyzátorov obsahuje kontrolu obslužných funkcii, pričom táto kontrola býva často len čiastočná. Pre toto chcela firma Red Hat vytvoriť voľne dostupný nástroj zameraný len na kontrolu obslužných funkcii [7]. Krátky prehľad je možné vidieť v tabuľke 2.1.

Astrée	Axivion Bauhaus Suite
Compass/ROSE	LDRA tool suite
Parasoft C/C++test	Polyspace Bug Finder
PRQA QA-C	RuleChecker
Splint	

Tabuľka 2.1: Prehľad dostupných analyzátorov kontrolujúcich pravidlo volanie len reentrantných funkcii [7].

Tabuľka 2.1 obsahuje len základný prehľad nástrojov, ktoré by mali byť schopné skontrolovať aspoň čiastočne volanie nereentrantných funkcii. Nie všetky tieto nástroje sú voľne dostupné, čo je dôležitý faktor pre firmu Red Hat, ktorá ponúka voľne dostupné riešenia.

2.2 Signály v Unixovom prostredí

Signály sú spôsob, ktorým informujeme proces, že došlo k nejakej udalosti. Zväčša sa používajú ako jednoduchá forma medziprocesovej komunikácie. Udalosti, ktoré spôsobujú zaslanie signálu procesu nastávajú asynchrónne v závislosti na vykonávaní kódu daného procesu. Z tohoto dôvodu je aj obsluha signálu spustená asynchrónne. Dá sa povedať, že signál je istá forma softvérového prerušenia. Existuje viac druhov signálov. Táto práca sa ale zameriava iba na signály podľa normy POSIX [8].

Norma POSIX.1-2008 definuje štandardizované rozhranie a prostredie operačného systému vrátane rozhrania príkazového riadku. Súčasťou tejto normy je definícia signálov, ich spracovania, a definuje funkcie, ktoré je možné považovať za reentrantné, alebo asynchrónne bezpečné. Taktiež tvrdí, že funkciu je možné považovať za bezpečnú, len ak je definovaná ako bezpečná. Zoznam bezpečných funkcií je možné nájsť v manuálových stránkach pomocou volania `man signal-safety`[4].

Každý signál má meno, ktoré vždy začína tromi znakmi SIG a následne identifikácia signálu(napríklad SIGSEGV, čiže signál spojený s príchodom takzvanej chyby segmentácie (segmentation fault), ktorý značí zásah do neoprávneného miesta v pamäti). Okrem toho sú reprezentované kladnou číselnou konštantou. Žiaden signál nemá konštantu 0, tá je rezervovaná na špeciálne účely [8]. Prehľad signálov je možné vidieť v prílohe A v tabuľkách A.1, A.2 a A.3.

Proces má tri možnosti ako obslúžiť signál [8]:

1. Ignorovať signál. Táto možnosť sa veľmi neodporúča, ak sa jedná o signál spojený s hardwarovou výnimkou. V prípade ignorovania takéhoto signálu hrozí, že sa aplikácia z tohto stavu nezotaví.

2. Nechať signál vykonať bežné chovanie. Sem najčastejšie patrí ukončenie procesu v prípade poruchy.

3. Obslúžiť signál pomocou vlastnej funkcie. V prípade použitia tejto možnosti si programátor, ktorý túto obslužnú funkciu implementuje, musí dať pozor, aby dodržal pravidlá na správnu obsluhu signálov. Ak tieto pravidlá nedodrží hrozí, že by mohol proces spôsobovať nedeterministické chovanie.

Trochu z histórie

Signály boli z počiatku takzvané nespoľahlivé. To znamená, že sa mohlo stať, že došlo ku generácii signálu, ale nie k jeho doručeniu procesu. Ďalšou vlastnosťou bolo to, že proces nemal veľkú kontrolu nad týmito signálmi. Jeho jediná možnosť bola buď signál obslúžiť, alebo ignorovať. Ďalšou z nevýhod bolo, že signál vždy po obslúžení bol nastavený na prednastavené obslúženie. Preto často tieto obslužné funkcie obsahovali opätovné nastavenie seba, ako obslužnú funkciu. Problémom však bola situácia, keď sa signál objavil pred týmto opätovným nastavením. Hlavne ak prednastavené chovanie je ukončenie procesu [8].

Pre toto došlo k zlepšeniu signálov. Keď dôjde ku generácii signálu, proces nastaví príznak v tabuľke procesu, že tento signál vznikol. To že signál je doručený procesu znamená, že bola vykonaná akcia, čiže volanie obslužnej funkcie. Medzi týmto časom je signál takzvaný čakajúci. Proces má možnosť blokovať signály. To znamená, že ak signál vznikne v čase keď je blokovaný, zostane čakajúci až do doby, kým buď proces signál neodblokuje, alebo nastaví jeho obsluhu na ignorovanie. Keď dôjde ku generácii viacerých tých istých signálov kým je signál blokovaný, tak sú dve možnosti. Systém si buď zapamätá len jeden výskyt signálu, alebo všetky. Väčšina systémov si pamätá len jeden výskyt toho istého signálu kým je blokovaný [8].

Signály v jazyku C

Signály v jazyku C sú definované v hlavičkovom súbore `<signal.h>`. Ich meno v tomto jazyku je len enumerácia, v skutočnosti ide len o číselné konštanty. Funkcia na zaslanie signálu sa volá `kill`. Táto funkcia je schopná zaslať ľubovolný signál ďalším procesom, ak má na to daný proces práva. Ak tieto práva nemá, táto funkcia zlyhá, a premennú `errno` nastaví na príslušnú hodnotu. Podobná funkcia ako `kill` je aj funkcia `raise`. Táto funkcia taktiež zasiela signál, ale príjemca je ten istý proces, ktorý túto funkciu zavolal. Čiže táto funkcia je schopná zaslať signál samému sebe [8].

Vlastnú funkciu na obsluhu signálu nastavuje programátor pomocou volania funkcie `signal` alebo `sigaction`. Táto funkcia dostane číslo signálu, ktorý chceme obslúžiť a ukazovateľ na funkciu, pomocou ktorej chceme tento signál obslúžiť. Funkcia `sigaction` dostáva namiesto ukazovateľa na funkciu štruktúru `struct sigaction`, v ktorej je tento ukazovateľ obsiahnutý. Podľa novej normy je odporúčané radšej používať na nastavenie vlastnej obsluhy signálu volanie funkcie `sigaction` [8].

2.2.1 Pravidlá na správnu obsluhu signálov

Medzi pravidlá na správnu obsluhu signálov patrí volanie len reentrantných funkcii. Reentrantné funkcie sú také funkcie, ktoré ak sú prerušené počas vykonávania svojho kódu a v obsluhe signálov sú opäť zavolané, tak po návrate z prerušenia dokončia svoju rutinu vždy rovnako, nezávisle na tom, v ktorom bode kódu došlo k ich prerušeniu [9]. Medzi nereentrantné funkcie patria napríklad také, ktoré používajú statické premenné, ktoré môžu meniť pri zavolaní svoju hodnotu. V prípade porušenia tohto pravidla môžu nastať viaceré problémy. V prípade funkcii, ktoré používajú zámky, môže dôjsť k večnému čakaniu na uvoľnenie zámku, v prípade statických štruktúr môžu funkcie vrátiť nesprávnu hodnotu. Toto pravidlo taktiež býva porušené najčastejšie.

Ďalším pravidlom je, že obsluha signálu nesmie zmeniť globálnu alebo statickú premennú, ak to vyslovene nevyžaduje. Sem najčastejšie patrí zmena premennej `errno`. Porušením tohto pravidla sa môže stať, že bude nesprávne klasifikovaná chyba získavaná z pre-

mennej `errno`. Toto môže nastať tak, že po zlyhaní funkcie, ktorá mení `errno`, ale pred testom hodnoty premennej `errno` príde signál, ktorý túto hodnotu zmení.

Taktiež zmena, alebo čítanie premennej by malo byť vždy atomické, to znamená, že nemôže dôjsť k prerušeniu pri zmene hodnoty premennej tak, že dáta v premennej budú nekonzistentné. Príkladom tohto porušenia je napríklad zápis do štruktúry obsahujúci dve čísla. Ak po zmene prvého nastane zachytenie signálu, tak v prípade, ak by obsluha signálu tieto hodnoty potrebovala, potom štruktúra neobsahuje správne hodnoty.

2.2.2 Príklady použitia obslužných funkcii

Celkom bežne používaným spôsobom obslúženia signálu je zapísanie do "rúry" (jednosmerná komunikácia, umožňujúca prenášať dáta z jedného procesu do druhého alebo aj v rámci procesu - navonok sa takáto "rúra" javí ako súbor). Potom slučka správ (run loop) dokáže pomocou funkcie `epoll` alebo `select` zistiť, že identifikátor súboru (file descriptor) identifikujúci danú "rúru" je pripravený na čítanie. Program si môže doobsluhovať rozpracované úlohy alebo obslúžiť čakajúcich klientov, ale neprijímať nové požiadavky, alebo šetrne skončiť operáciu, upratať si alokované zdroje a ukončiť sa. V prípade že pri obsluhe signálu dôjde k ďalšej obsluhe signálu, volanie `write` zlyhá a zmení premennú `errno` na hodnotu `EAGAIN`. Môže sa stať že sa zmenila premenná `errno`. Ak programátor nezachytí danú zmenu a neobnoví pôvodnú hodnotu, môže dôjsť k zmene premennej `errno` aj po návrate z obslužnej funkcie. Zdrojový kód je možné vidieť v kóde 2.1.

Na príklade je vidieť, že volanie funkcie `write`, ktoré môže zmeniť premennú `errno`, je ošetrené prostredníctvom zálohovania hodnoty premennej `errno` do premennej `savedErrno`. Táto hodnota je následne obnovená tesne pred návratom z obslužnej funkcie.

```
       ...

static int pfd[2]; /* File descriptors for pipe */

static void
handler(int sig)
{
    int savedErrno; /* In case we change 'errno' */

    savedErrno = errno;
    if (write(pfd[1], "x", 1) == -1 && errno != EAGAIN)
        errExit("write");
    errno = savedErrno;
}

int
main(int argc, char *argv[])
{
       ...
    while ((ready = select(nfds, &readfds, NULL, NULL, pto)) == -1 &&
                errno == EINTR)
            continue; /* Restart if interrupted by signal */
       ...
}
```

Kód 2.1: Zdrojový kód ukazujúci použitie obslužných funkcii [5]

Majme situáciu, že obslužná funkcia uvedená v kóde 2.1 neobsahuje zálohu a obnovu premennej `errno` a ani ukončenie programu v prípade chyby. Taktiež v hlavnom tele programu obsahuje algoritmus podobný ako ten uvedený v kóde 2.2 a volanie operácie `op1` zlyhá, chybu nastaví prostredníctvom premennej `errno`. Následne tesne pred kontrolou premennej `errno` dôjde k príchodu signálu, a jeho obslúženie vyššie spomínanou chybnou obslužnou funkciou. V tejto obslužnej funkcii zlyhá z nejakých dôvodov volanie funkcie `write`. Táto funkcia tiež zmení premennú `errno`. Po návrate z obslužnej funkcie je hodnota premennej `errno` nastavená nie na pôvodnú hodnotu dôvodu zlyhania operácie `op1`, ale hodnotu z obslužnej funkcie. Toto môže viesť k nesprávnemu klasifikovaniu chyby, a tým pádom nesprávne zvolenej reakcii na túto chybu.

```
       ...
int ret = op1(...);
if (ret < 0) {
        switch (errno) {
       ...
        case EAGAIN: //zopakuj operaciu
               break;
       ...
        default: log_error("unspecified error");
}
       ...
```

Kód 2.2: Zdrojový kód ukazujúci dôsledok chybnej obslužnej funkcie

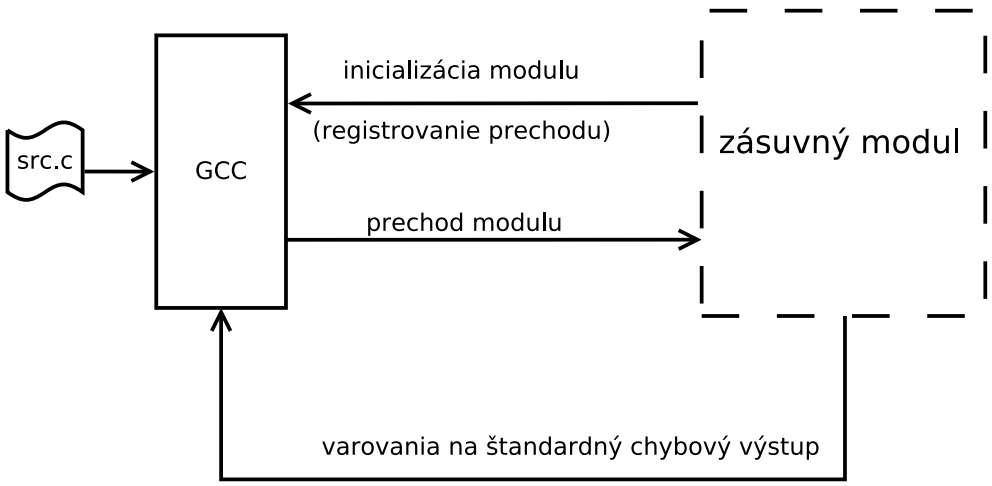
Takéto chyby spôsobujú nedeterministické správanie aplikácie a môže byť ťažké ich potom odhaliť. Následkom nesprávnej obsluhy signálu môže byť skončenie aplikácie, nepodarená aktualizácia update alebo prihlásenie, a podobne.

2.3 GCC, the GNU Compiler Collection

GCC je kolekcia prekladačov na rôzne jazyky. Zahŕňa prekladače na C, C++, Fortran, Ada, Go a ďalšie. Tento softvér je 100% voľne dostupný a dokonca voľne modifikovateľný. GCC ponúka rozhranie na tvorbu a zavádzanie zásuvných modulov [1] [2].

Text spomínaný v nasledujúcich odsekoch platí pre GCC verziu 8.1.0 až po verziu 8.3.1. Pretože GCC sa neustále mení, môže sa zmeniť spôsob registrovania prechodov, vnútorná štruktúra prekladača ako aj názvy premenných a metód. GCC bol najprv implementovaný v jazyku C, ale prebieha snaha o jeho prepísanie do jazyka C++ [1] [2].

Zásuvné moduly je možné použiť len na verzii GCC, pomocou ktorej boli zostavené. Zásuvný modul je zostavený ako dynamická knižnica, a následne sa zavádza pomocou argumentu `-fplugin=`, za ktorým nasleduje cesta ku knižnici. Tento modul musí vždy obsahovať funkciu `plugin_init`, ktorá inicializuje zásuvný modul. Všeobecnú blokovú schému je možné vidieť na obrázku 2.1 [1] [2].



Obr. 2.1: Bloková schéma všeobecného zásuvného modulu.

Zásuvný modul musí taktiež obsahovať globálnu premennú typu `int`, ktorá sa volá `plugin_is_GPL_compatible`. Táto premenná nemá nejaké zvláštne použitie, ale slúži na upozornenie programátora, že zásuvný modul by mal byť kompatibilný s licenciou GPL. GCC je vydávané pod touto licenciou, a preto je možná jeho voľná modifikácia podľa potrieb. Táto licencia dovoľuje modifikáciu za cenu, že aj ďalšie programy z neho vychádzajúce budú vydané pod touto, alebo kompatibilnou licenciou. To platí aj pre zásuvné moduly [1] [2].

Následne je možné pomocou funkcie `register_callback` zaregistrovať ďalšie prechody. O to sa stará časť GCC, ktorá sa nazýva pass manager. Jeho prostredníctvom je možné pridávať, odoberať alebo nahrádzať všetky GCC prechody [2].

Na zaregistrovanie prechodu je predovšetkým dôležité tento prechod implementovať. To je možné napríklad vytvorením štruktúry, ktorá dedí z `gimple_opt_pass` pre prechod používajúci reprezentáciu Gimple. Táto štruktúra musí následne implementovať metódu `execute`, ktorá bude volaná pre každú funkciu z prekladaného zdrojového súboru. Táto metóda dostáva ako parameter ukazovateľ na štruktúru `function`, prostredníctvom ktorého je možné pristupovať ku kódu aktuálne prechádzanej funkcie [3] [2].

2.3.1 Vnútorná štruktúra prekladača GCC

GCC, ako vnútornú reprezentáciu kódu používa RTL a Gimple. Gimple slúži na jazykovo nezávislú optimalizáciu a RTS slúži na optimalizáciu na danú platformu, a následnú generáciu kódu. Front end reprezentácia kódu pre C a C++ sa nazýva GENERIC. Táto práca sa zaoberá reprezentáciou Gimple [3] [2].

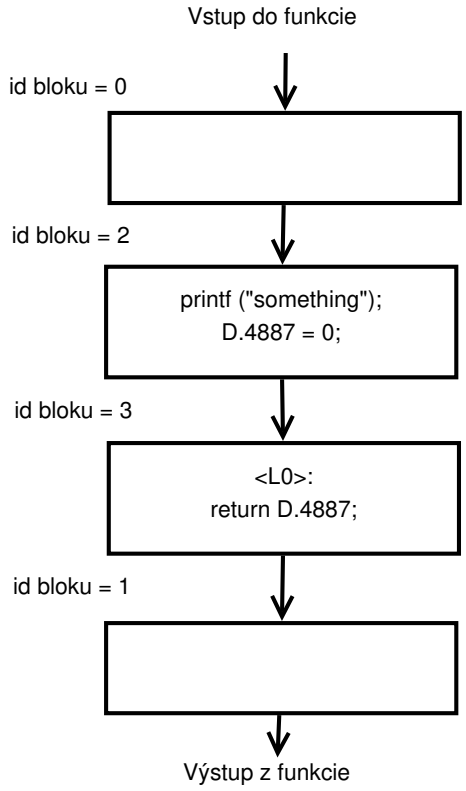
Gimple je kód, ktorý pozostáva z identifikačného čísla a maximálne troch operandov. Identifikačné číslo určuje, čo za kód dané Gimple tvrdenie reprezentuje. Sem patrí napríklad priradenie(`GIMPLE_ASSIGN`) alebo volanie funkcie(`GIMPLE_CALL`). Prístup k tejto identifikácií sa vykonáva prostredníctvom volania funkcie `gimple_code`, ktorý dostane ako parameter príslušné Gimple tvrdenie. Operandy sú typu tree, ktoré môžu byť jednoduché alebo komplexnejšie. Tree obsahuje tiež identifikáciu podľa toho, čo určuje. Napríklad identifikácia premennej je `VAR_DECL` a identifikácia funkcie je `FNC_DECL`. Komplexnejšie tree môžu byť zložené z viacerých tree operandov [2].

Funkcie sú reprezentované prostredníctvom tree typu `FNC_DECL`. Tu je možné zistiť o funkcii všetky informácie. Samotný kód je ale dostupný prostredníctvom ukazovateľa na štruktúru `function`. Telo funkcie je zložené z častí, ktoré sa nazývajú basic block. Každý tento blok je sekvenčná časť kódu, ktorá má len jeden vstup, a len jeden výstup. To znamená, že v rámci jedného bloku nedochádza k rozhodovaniu podmienok, ani k cyklom. Tieto rozhodnutia sa vždy vykonávajú na hranách, ktoré tieto bloky spájajú. Tieto hrany môžu byť podmienené určitou podmienkou, a jeden blok môže obsahovať rôzny počet výstupných hrán. Blok okrem hrán a samotného kódu obsahuje identifikáciu bloku. Blok obsahujúci ID 0 je vždy počiatočný blok vo funkcii, a blok obsahujúci ID 1 je vždy blok výstupný. Tieto dva bloky neobsahujú žiaden kód. Takéto riešenie je pre to, aby v prípade vetvenia nedochádzalo k tomu, že by funkcia obsahovala viacej výstupných blokov. Spojením týchto informácii dostávame takzvaný Control Flow Graph, skrátene CFG [2].

Grafické znázornenie príkladu CFG je možné vidieť na obrázku 2.2. Tento CFG reprezentuje funkciu, ktorá nerobí nič iné, iba volá funkciu `printf`, za ktorou nasleduje volanie `return 0`. Zdrojový kód tejto funkcie písaný v jazyku C je možné vidieť v kóde 2.3.

```
int fnc()
{
    printf("something");
    return 0;
}
```

Kód 2.3: Zdrojový kód odpovedajúci grafickej reprezentácii CFG z obrázku 2.2



Obr. 2.2: Príklad grafickej reprezentácie CFG.

Prechod cez všetky Gimple tvrdenia je možné vykonať pomocou makier a štruktúr, uľahčujúcich tento prístup. Makro `FOR_ALL_BB_FN` prejde všetky bloky vo funkcii. Toto makro berie dva parametre, a to ukazovateľ na štruktúru `function` a štruktúru `basic_block`, do ktorej postupne vkladá údaje o samostatných blokoch. Prechod Gimple tvrdeniami v bloku je možné pomocou štruktúry `gimple_stmt_iterator`, ktorá obsahuje na toto slúžiaci iterátor. Následne pomocou makier `gsi_start_bb`, `gsi_end_p` a `gsi_next` je možné iterovať cez jednotlivé Gimple tvrdenia [2].

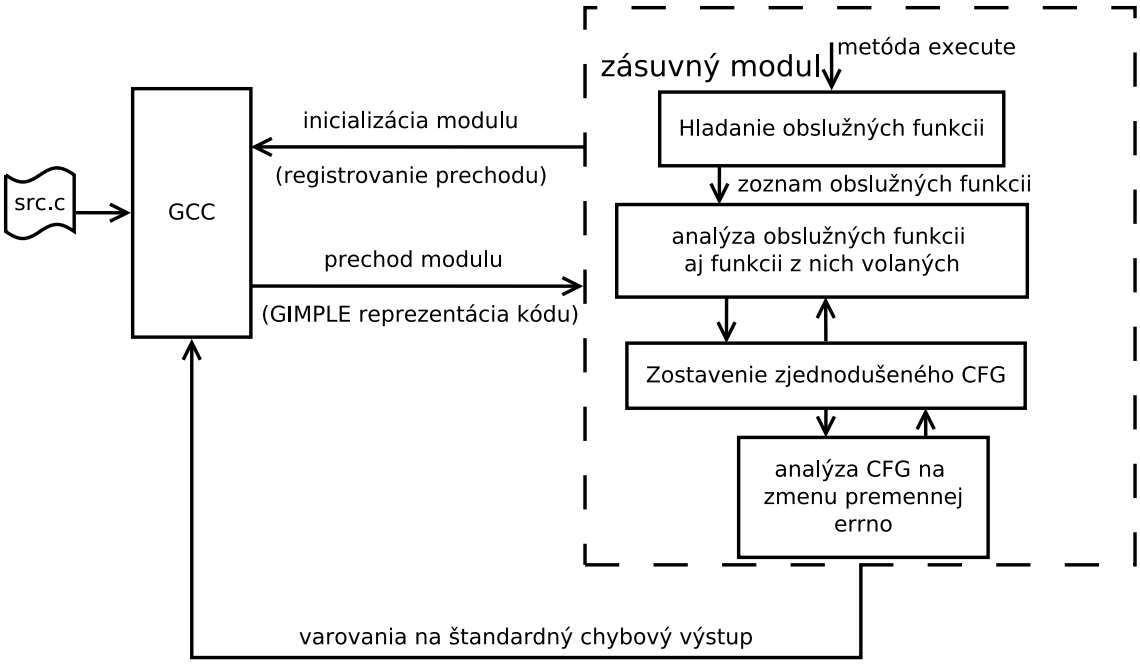
Kapitola 3

Objektovo orientovaný návrh zásuvného modulu

Keďže GCC je písané v jazyku C++, tak aj samotný analyzátor je písaný v tomto jazyku. C++ je objektovo orientovaný jazyk a preto som použil objektovo orientovaný návrh. Táto kapitola sa zaoberá blokovou schémou modulu, jeho diagramom tried a popisom samotného algoritmu použitého na tvorbu zásuvného modulu.

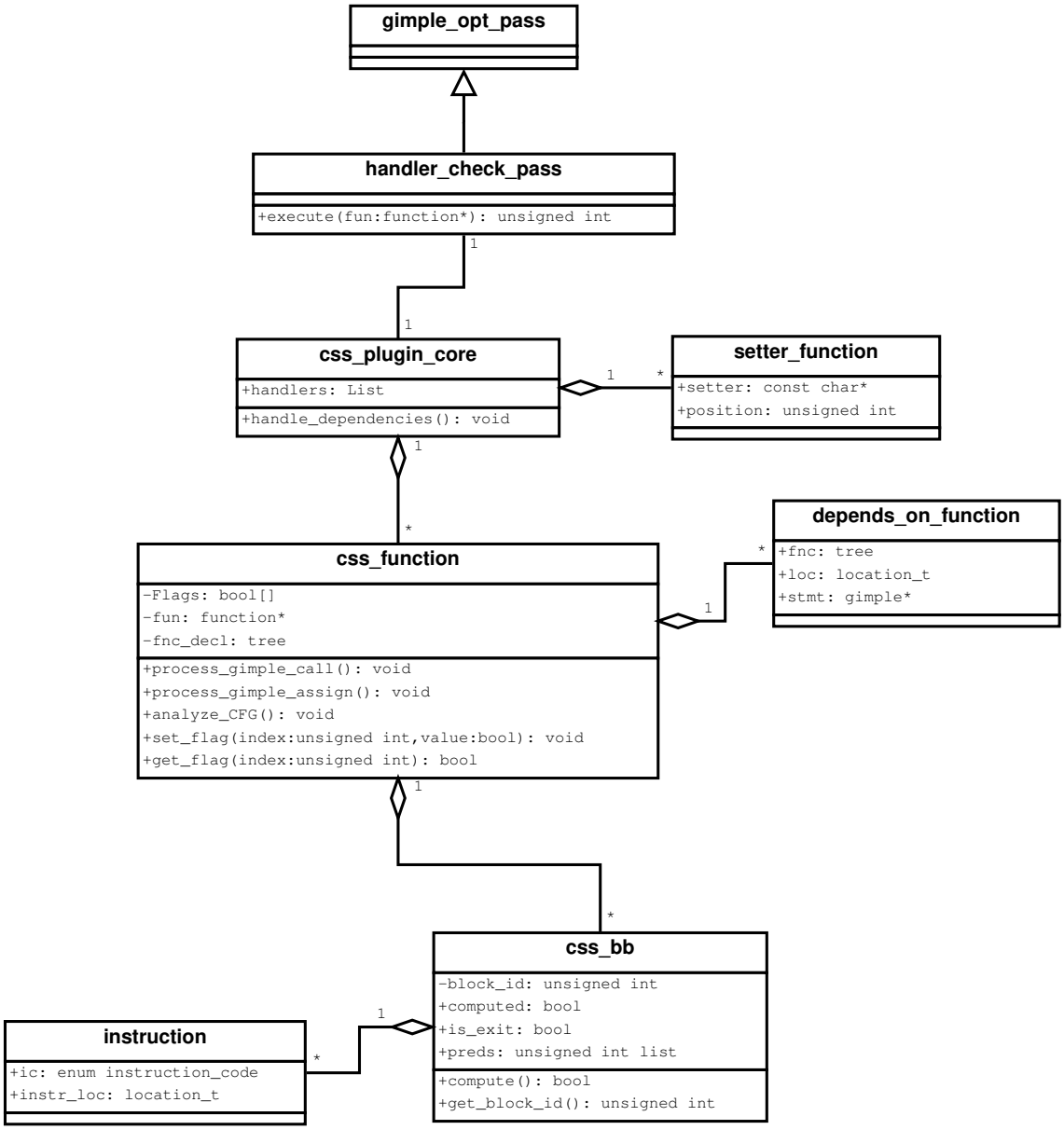
3.1 Bloková štruktúra zásuvného modulu

Na obrázku 3.1 je možné vidieť blokovú schému zásuvného modulu. Prvé k čomu dochádza, je inicializácia zásuvného modulu, pri ktorej dôjde k registrácií prechodu. Následne GCC vykonáva svoje prechody, až pokým nenastane čas na prechod zásuvného modulu. V tomto prechode GCC postupne volá funkciu `execute` registrovaného prechodu pre každú funkciu, ktorá sa nachádza v zdrojovom súbore. V tejto časti zásuvný modul prechádza zdrojový kód a snaží sa identifikovať funkcie, ktoré slúžia na obsluhu signálov. Po tejto časti zásuvný modul začne prechádzať obslužné funkcie pre porušenia volania len reentrantných funkcií. Pri tejto časti prechádza aj volané funkcie definované v danom zdrojovom súbore, ktoré preskenuje tiež pre toto porušenie. Tým sa docieli, že nedôjde k tranzitívnemu volaniu nereentrantnej funkcie. Popri tejto analýze zostaví zásuvný modul zjednodušený CFG. Následne prechádza modul tento CFG, aby zistil, či daná obslužná funkcia nemôže zmeniť premennú `errno`. Všetky porušenia hlási prostredníctvom GCC na štandardný chybový výstup.



Obr. 3.1: Bloková schéma zásuvného modulu.

Na obrázku 3.2 je možné vidieť diagram tried zásuvného modulu. Najdôležitejšou časťou je samotný prechod, ktorý obsahuje metódu `execute`. Ďalšou, taktiež dôležitou časťou je trieda `css_plugin_core`, ktorá obsahuje všetky zistené informácie o každej funkcii nachádzajúcej sa v analyzovanom zdrojovom súbore. Tieto informácie sú uložené prostredníctvom triedy `css_function`. Tá obsahuje informácie o jednej funkcii. Táto trieda obsahuje zoznam informácii o blokoch, ktoré sú uložené v triede `css_bb`, ktorá obsahuje zoznam inštrukcii v usporiadanom poradí. Inštrukcia je tiež uložená prostredníctvom triedy, ktorá sa nazýva `instruction`.



Obr. 3.2: Diagram tried zásuvného modulu.

Nástroj klasifikuje funkcie primárne do piatich skupín:

1. Funkcie ktoré sú bezpečné na použitie v obslužnej funkcii, a ich použitím nemôže dôjsť k zmene premennej `errno`. Knižničné funkcie, ktoré spadajú do tejto kategórie je možné vidieť v tabuľke 3.1.

2. Funkcie, ktoré sú bezpečné na použitie v obslužnej funkcii, ale pri ich volaní môže dôjsť k zmene premennej `errno`. Knižničné funkcie, ktoré spadajú do tejto kategórie je možné vidieť v tabuľke 3.2.

3. Funkcie, ktoré spôsobujú ukončenie aplikácie a je bezpečné ich volať v obslužnej funkcii. Knižničné funkcie, ktoré spadajú do tejto kategórie je možné vidieť v tabuľke 3.4.

4. Funkcie, ktoré nie je bezpečné volať v obslužnej funkcii, a preto by sa v takej funkcii nemali nachádzať. Týchto funkcií je obrovské množstvo, a preto bolo potrebné vybrať len určitú množinu funkcií, ktoré často porušujú túto zásadu. Prehľad takýchto knižničných funkcií je možné vidieť v tabuľke 3.3.

5. Funkcie, ktoré nespadajú do žiadnej z vyššie uvedených skupín.

aio_error	alarm	cfgetispeed
cfgetospeed	getegid	geteuid
getgid	getpgrp	getpid
getppid	getuid	htonl
htons	longjmp	memccpy
memchr	memcmp	memcpy
memmove	memset	ntohl
ntohs	posix_trace_event	pthread_kill
pthread_self	pthread_sigmask	raise
sigfillset	siglongjmp	sleep
stpcpy	stpncpy	strcat
strchr	strcmp	strcpy
strspn	strlen	strncat
strncmp	strncpy	strnlen
strpbrk	strchr	strspn
strstr	strtok	umask
wcpcpy	wcpncpy	wcscat
wcschr	wcscmp	wcscpy
wcscspn	wcslen	wcsncat
wcsncmp	wcsncpy	wcsnlen
wcspbrk	wcsrchr	wcsspn
wcsstr	wcstok	wmemchr
wmemcmp	wmemcpy	wmemmove
wmemset		

Tabuľka 3.1: Zoznam knižničných funkcii, ktoré nástroj vyhodnotí ako bezpečné.

accept	access	aio_return
aio_suspend	bind	cfsetispeed
cfsetospeed	clock_gettime	close
connect	creat	dup
dup2	execl	execle
execv	execve	faccessat
fcntl	fdatasync	fexecve
fchdir	fchmod	fchmodat
fchown	fchownat	fork
fstat	fstatat	fsync
ftruncate	futimens	getgroups
getpeername	getsockname	getsockopt
chdir	chmod	chown
kill	link	linkat
listen	lseek	lstat
mkdir	mkdirat	mkfifo
mkfifoat	mknod	mknodat
open	openat	pause
pipe	poll	pselect
read	readlink	readlinkat
recv	recvfrom	recvmsg
rename	renameat	rmdir
select	sem_post	send
sendmsg	sendto	setgid
setpgid	setsid	setsockopt
setuid	shutdown	sigaction
sigaddset	sigdelset	sigemptyset
sigismember	signal	sigpause
sigpending	sigprocmask	sigqueue
sigset	sigsuspend	sockatmark
socket	socketpair	stat
symlink	symlinkat	tcdrain
tcflow	tcflush	tcgetattr
tcgetpgrp	tcsendbreak	tcsetattr
tcsetpgrp	time	timer_getoverrun
timer_gettime	timer_settime	times
uname	unlink	unlinkat
utime	utimensat	utimes
wait	waitpid	write

Tabuľka 3.2: Zoznam knižničných funkcii, ktoré nástroj vyhodnotí ako bezpečné, ale môžu meniť premennú `errno`.

aligned_alloc	atexit	___builtin_putchar
___builtin_puts	calloc	err
error	error_at_line	errx
exit	fclose	fcloseall
fgetc	fgetc_unlocked	fgetwc
fgetwc_unlocked	fopen	fopen64
fprintf	fputc	fputc_unlocked
fputs	fputs_unlocked	fputwc
fputwc_unlocked	fputws	fputws_unlocked
free	freopen	freopen64
fscanf	fwprintf	fwscanf
getc	getc_unlocked	getchar
getchar_unlocked	getw	getwc
getwc_unlocked	getwchar	getwchar_unlocked
grantpt	mallinfo	malloc
mallopt	memalign	mtrace
muntrace	on_exit	perror
posix_memalign	printf	ptsname
ptsname_r	putc	putc_unlocked
putchar	putchar_unlocked	puts
putw	putwc	putwc_unlocked
putwchar	putwchar_unlocked	realloc
reallocarray	scanf	sem_close
sem_open	sem_unlink	shm_open
shm_unlink	sigsetjmp	snprintf
sprintf	sscanf	strerror
strerror_r	swprintf	swscanf
tempnam	tmpfile	tmpfile64
tmpnam	unlockpt	valloc
verr	verrx	vwarn
vwarnx	warn	warnx
wprintf	wscanf	

Tabuľka 3.3: Zoznam knižničných funkcii, ktoré nástroj vyhodnotí ako nebezpečné na použitie v obsluhe signálov.

_exit
_Exit
abort

Tabuľka 3.4: Zoznam knižničných funkcii, ktoré nástroj vyhodnotí ako bezpečné, a zároveň spôsobia ukončenie aplikácie.

3.2 Popis algoritmu zásuvného modulu

V prvom momente dôjde k volaniu funkcie `plugin_init`, ktorá zaregistruje prechod modulu, ktorý sa volá `handler_check_pass`. Tento prechod obsahuje metódu `execute`. Na začiatku táto metóda vytvorí nový objekt typu `css_function`. Ten ukladá do zoznamu `fnc_list` uloženého v objekte `plugin_core`, ktorý je typu `css_plugin_core`. Tento objekt `plugin_core` je globálne prístupný objekt, uchovávajúci informácie o všetkých informáciach, ktoré nástroj zisťuje. Následne postupne prechádza všetky Gimple tvrdenia nachádzajúce sa v aktuálne prechádzanej funkcii zo zdrojového súboru. Pri tomto prechode ho ale zaujímajú iba Gimple tvrdenia typu `GIMPLE_CALL` a typu `GIMPLE_ASSIGN`. Pseudokód metódy `execute` je možné vidieť na v kóde 3.1.

```
virtual unsigned int execute(function * fun) override
{
    plugin_core.fnc_list.push_front(new_fnc);
    tree handler=nullptr;

    Gimple stmt;
    for_each_Gimple_stmt(stmt)
    {
        if (gimple_code(stmt)==GIMPLE_CALL)
            if(handler=get_handler(stmt)!=nullptr)
            {
                plugin_core.handlers.push_front(handler);
                handler=nullptr;
            }
        else if (gimple_code(stmt)==GIMPLE_ASSIGN)
        {
            tree lValue,rValue;
            lValue = get_lValue(stmt);
            rValue = get_rValue(stmt);
            if(is_struct_sigaction(lValue))
                Possible_handler_remember(rValue);
        }
        if (handler!=nullptr)
    }
    for_each_handler(handler)
        scan_own_function(get_name(handler),...);
    return 0;
}
```

Kód 3.1: Pseudokód metódy `execute`

Tvrdenia typu `GIMPLE_ASSIGN` kontroluje na všetky priradenia do štruktúr typu `struct sigaction`, a to konkrétne dosadenia do premenných `sa_handler` a `sa_sigaction` nachádzajúcich sa v tejto štruktúre. Následne je uložená dvojica štruktúra a k nej prislúchajúca obslužná funkcia uložené do zoznamu `possible_handlers` nachádzajúcom sa v objekte `plugin_core`. Tento zoznam obsahuje štruktúry `handler_in_var`, ktoré slúžia práve na uloženie jednej dvojice štruktúra a k nej prislúchajúca obslužná funkcia.

Gimple tvrdenia typu `GIMPLE_CALL` kontroluje na volania funkcii `sigaction`, alebo `signal`. V prípade použitia funkcie `signal` získava obslužnú funkciu priamo z parametru, kde naopak u funkcie `sigaction` je tento postup zložitejší. U tohto typu sa získava iba názov štruktúry `struct sigaction`, ktorá je následne vyhľadávaná v zozname `possible_handlers`. V prípade nájdenia príslušnej štruktúry v tomto zozname je zistený názov obslužnej funkcie. Ak dôjde k tomu, že táto štruktúra nebola nájdená v spomínanej štruktúre, tak dôjde ešte k prístupu k inicializácii premennej. K tomuto môže dôjsť tak, že táto premenná je statická alebo globálna a tým pádom je vykonaná inicializácia mimo samotný kód funkcie. Ak sa nepodarí nájsť príslušnú obslužnú funkciu, tak je toto volanie funkcie `sigaction` ďalej ignorované.

Všetky nájdené obslužné funkcie sú ukladané v zozname `handlers`, uložených v objekte `plugin_core`. Tento zoznam je postupne prechádzaný prvok po prvku a následne dochádza k skenovaniu obslužnej funkcie pomocou funkcie `scan_own_function`.

scan_own_function

Táto funkcia prechádza funkcie zo zoznamu `fnc_list` a hľadá zhodu s funkciou, ktorú má skenovať. Po nájdení zhody skontroluje všetky potrebné príznaky. V prípade že funkcia ešte nebola preskenovaná, nastáva jej skenovanie. To pozostáva v prechode všetkých Gimple tvrdení v kóde. Gimple tvrdenia typu `GIMPLE_CALL` sa spracúvajú pomocou metódy `process_gimple_call` a Gimple tvrdenia, ktoré sú typu `GIMPLE_ASSIGN` sa spracúvaju pomocou metódy `process_gimple_assign`.

Tieto dve metódy popri svojej činnosti ešte postupne vytvárajú zoznam inštrukcii, ktoré slúžia na ďalšiu analýzu, a to konkrétne na analýzu, či nedošlo k zmene premennej `errno`. Táto analýza využíva CFG (Control Flow Graph). Tieto inštrukcie slúžia na zjednodušenie analýzy. To funguje tak, že sú zapamätané, len tie Gimple tvrdenia, ktoré sú pre túto analýzu potrebné, a to v zjednodušenej forme. Pseudokód funkcie `scan_own_function` je možné vidieť v kóde 3.2.

```
return_code scan_own_function (const char* name,...)
{
    if(undirect_recurse(name))
        return RC_CYCLIC;

    return_code rc;
    css_function fnc;
    fnc=get_function_from_fnc_list(name);

    if(is_scaned(fnc))
    {
        rc=check_flags(fnc);
    }
    else
    {
        bb block;
        for_each_block(block)
        {
            css_bb plugin_block=build_block(bb);
            Gimple stmt;
            for_each_Gimple_stmt(stmt)
            {
                if (gimple_code(stmt)==GIMPLE_CALL)
                    fnc.process_gimple_call(plugin_block,stmt,...);
                else if (gimple_code(stmt)==GIMPLE_ASSIGN)
                    fnc.process_gimple_assign(plugin_block,stmt,...);
            }
            fnc.push_block(plugin_block);
        }
        obj.analyze_CFG();
        rc=check_flags(fnc);
    }
    if(fnc.is_handler)
        print_warnings(fnc);
    return rc;
}
```

Kód 3.2: Pseudokód funkcie `scan_own_function`

process_gimple_call

V metóde `process_gimple_call` je získaná volaná funkcia, ktorá je ďalej kontrolovaná, či je definovaná v danom module, alebo nie. To sa získava prostredníctvom makra `DECL_INITIAL`, ktoré vracia hodnotu true, v prípade že táto funkcia je definovaná v aktuálnom module.

Ak je definovaná v aktuálnom module, dochádza k jej kontrole, tiež prostredníctvom volania funkcie `scan_own_function`. Táto funkcia je tým pádom rekurzívna a zavolá sa toľko krát, koľko je treba. Problém tejto implementácie je cyklické volanie funkcii. Pre tento prípad dostáva funkcia `scan_own_function` ako parameter zoznam funkcii, ktoré

aktuálne skenuje. Každá funkcia na začiatku svojho skenovania skontroluje, či sa v zozname nenachádza. Ak sa v spomínanom zozname nachádza, skenovanie končí s návratovým kódom indikujúcim cyklickú závislosť. Ak sa tam nenachádza, tak pridá do tohto zoznamu záznam o svojom skenovaní.

Ak funkcia nie je definovaná v aktuálne skenovanom module, najprv dôjde ku kontrole, či volaná funkcia nie je `__errno_location`. Toto volanie je v podstate prístup k premennej `errno`. Ak to nieje toto volanie, následne dochádza na kontrolu pomocou funkcie `is_handler_ok_fnc`, ktorá kontroluje názov funkcie so zoznamom bezpečných funkcii podľa normy POSIX.

Ak sa v tomto zozname nenachádza, dôjde ešte ku kontrole prostredníctvom funkcie `is_handler_wrong_fnc`. Táto naopak hľadá zhodu s najčastejšími porušeniami ako sú napríklad funkcie `malloc`, `free`, `exit`, `printf` a podobne. Toto je taktiež riešené pomocou konštantného zoznamu, ako aj pri funkciách bezpečných. Pseudokód prislúchajúci k metóde `process_gimple_call` je možné vidieť v kóde 3.3.

```
void process_gimple_call(css_bb &plugin_block, Gimple stmt...)
{
    tree fn_decl = gimple_call_fndecl(stmt);
    const char* called_function_name=get_fnc_name(fn_decl);
    return_code rc;
    if (DECL_INITIAL (fn_decl))
    {
        rc=scan_own_function(called_function_name,...);
    }
    else
    {
        rc=is_handler_ok_fnc(called_function_name);
    }
    this->set_flags_according_rc(rc);
    if(rc_is_relevant_to_CFG_analysis(rc))
        plugin_block.add_instruction(instruction_from_stmt(stmt));

}
```

Kód 3.3: Pseudokód metódy `process_gimple_call`

process__gimple__assign

Metóda `process_gimple_assign` kontroluje priradenia premenných na kontrolu zálohovania a obnovy premennej `errno`. Udržiava si zoznam premenných, do ktorých bolo v nejakom čase uložená premenná `errno`. To slúži na udržiavanie možných úložísk premennej `errno`. Následne kontroluje priradenia rôznych variant ako sú z premennej do premennej, alebo prostredníctvom referencii na premenné a `errno`. Pseudokód metódy `process_gimple_assign` je možné vidieť v kóde 3.4.

```
void process_gimple_call(css_bb &plugin_block, Gimple stmt...)
{
    if(is_errno_assign(stmt))
    {
        plugin_block.add_instruction(instruction_from_stmt(stmt));
    }
}
```

Kód 3.4: Pseudokód metódy `process_gimple_assign`

analyze_CFG

Analýza CFG je vykonaná prostredníctvom metódy `analyze_CFG`. Táto metóda prechádza zoznam `block_status`, ktorý obsahuje objekty `css_bb`. Tieto objekty uchovávajú informácie o jednotlivých blokoch vo funkcii. Taktiež obsahujú dve množiny, jednu výstupnú a jednu vstupnú. Tieto množiny uchovávajú informáciu o premenných, v ktorých je buď na vstupe, alebo na výstupe uložená počiatočná hodnota premennej `errno`.

Taktiež tento objekt obsahuje metódu `compute`, ktorá vypočíta zo vstupnej množiny množinu výstupnú. To funguje tak, že prechádza jednotlivé inštrukcie v bloku a postupne upravuje kópiu vstupnej množiny až na konci bloku bude táto množina reprezentovať výstupnú množinu. Táto metóda taktiež zisťuje či sa výstupná množina zmenila od predchádzajúcej hodnoty výstupnej množiny. Keď dôjde k návratu z funkcie s možnou zmenou premennej `errno` metóda `compute` na toto upozorní.

Výpočet vstupnej množiny naopak realizuje samotná metóda `analyze_CFG`. To realizuje tak, že prejde prepočítané predchádzajúce bloky k aktuálne počítanému bloku a spraví prienik ich výstupných množín. Metóda `analyze_CFG` postupne prepočítava množiny blokov, až po kým sa nedostane do situácie, že sa pri prechode nezmení hodnota žiadnej z výstupných množín. Tento ustálený stav znamená, že analýza CFG bola dokončená. V takom stave stačí už len skontrolovať, či blok s ID s hodnotou 1 obsahuje vo svojej výstupnej množine samotnú premennú `errno`. Pseudokód metódy `analyze_CFG` je možné vidieť v kóde 3.5 a pseudokód metódy `compute` je možné vidieť v kóde 3.6.

```
void analyze_CFG()
{
    this->check_if_is_exit();

    do_while_something_changed
    {
        css_bb block;
        this->list_of_css_bb.for_all_bb(block)
        {
            block.input_set=intersection_of_all_predecessors_output_sets;
            if(block.imput_set.changed || !block.computed)
            {
                if(block.compute(...))
                {
                    this->set_flag_errno_changed();
                    return;
                }
            }
        }
    }
}
```

Kód 3.5: Pseudokód metódy `analyze_CFG`

```
bool compute(...)
{
    this->computed=true;
    instruction instr;
    for_all_instructions_in_block(instr)
    {
        switch(instr.ic)
        {
            ...
            change_output_set_according_instruction
            ...
        }
    }
    if(this->is_return_block)
    {
        if(!set_contain_errno(this->output_set))
            return true;
    }
    return false;
}
```

Kód 3.6: Pseudokód metódy `compute`

print_warning a print_errno_warning

Samotné chyby sú vypísané pomocou funkcii `print_warning` a `print_errno_warning`. Funkcia `print_warning` vypíše varovanie ohľadom nereentrantnej funkcie. Toto môže byť s prívlastkom, že ide o možnú nereentrantnú funkciu,ak táto funkcia nie je definovaná v aktuálnom module, a zároveň nie je v zozname nevhodných funkcii.

Následne táto funkcia zavolá funkciu `print_note`, ktorá ďalej doplní tranzitívne funkcie, ktoré sú dôvodom pre túto chybu. Funkcia `print_errno_warning` vypisuje chybu spôsobenú možnou zmenou premennej `errno`. Taktiež na získanie elementárneho volania, ktoré spôsobilo túto chybu, zavolá funkciu `print_errno_note`, ktorá podobne ako funkcia `print_note` tranzitívne prejde všetky funkcie, ktoré môžu premennú `errno` zmeniť. Pseudokód funkcie `print_warning` je možné vidieť v kóde 3.7, pseudokód funkcie `print_note` v kóde 3.8, pseudokód funkcie `print_errno_warning` v kóde 3.9 a pseudokód funkcie `print_errno_note` je možné vidieť v kóde 3.10 . Výpis týchto funkcii je možné vidieť na obrázku 4.1 pre výskyt nereentrantnej funkcie, a na obrázku 4.2 pre možnú zmenu premennej `errno`.

```
void print_warning(tree handler,tree fnc,location_t loc,bool fatal)
{
    string msg=generate_warning_msg(handler,fnc,fatal);
    warning_at(loc,msg);
    print_note(fnc,loc,fatal);
}
```

Kód 3.7: Pseudokód funkcie `print_warning`

```
void print_note(tree fnc, location_t loc, bool fatal)
{
    css_function function=plugin_core.fnc_list.get_fnc(fnc);
    if(function==nullptr)
    {
        string msg=generate_note(fnc);
        inform(loc,msg);
        return;
    }
    for_all_warnings(warning)
    {
        string msg=generate_warning_msg(fnc,warning.fnc,fatal);
        inform(warning.loc,msg);
        print_note(warning.fnc,warning.loc,fatal);
    }
}
```

Kód 3.8: Pseudokód funkcie `print_note`

```
void print_errno_warning(tree handler, tree fnc, location_t loc)
{
    string msg=generate_errno_msg(handler);
    warning_at(loc,msg);
    print_errno_note(fnc);
}
```

Kód 3.9: Pseudokód funkcie `print_errno_warning`

```
void print_errno_note(tree fnc)
{
    css_function function=plugin_core.fnc_list.get_fnc(fnc);
    if(function==nullptr)
        return;
    string msg=generate_errno_msg(function.errno_fnc);
    warning_at(function.errno_loc,msg);
    print_errno_note(function.errno_fnc);
}
```

Kód 3.10: Pseudokód funkcie `print_errno_note`

Kapitola 4

Implementácia a testovanie zásuvného modulu

Táto kapitola sa zaoberá implementačnými detailami a použitými nástrojmi na tvorbu a testovanie zásuvného modulu. Približuje vývoj aplikácie, ako aj postup a výsledky jej testovania. Ukazuje príklady použitia nástroja, ako aj príklady výstupu.

4.1 Implementácia zásuvného modulu

Nástroj je implementovaný v jazyku C++ a následne zostavený ako dynamická knižnica, aby mohol slúžiť ako zásuvný modul. Na zostavenie modulu je vytvorený Makefile. Nástroj bol počas svojho vývoja skúšaný na GCC verzii 8.1.0 až po verziu 8.3.1. Na testovanie sa používa nástroj ctest, ktorý je súčasťou nástroja cmake.

Nástroj sa skladá z dvoch súborov a to csigsafe.cc a csigsafe.hh. Hlavičkový súbor obsahuje deklarácie všetkých štruktúr, funkcii a makier použitých v projekte, až na triedu `handler_check_pass`. Táto trieda obsahuje len definíciu, nie osamostatnenú deklaráciu. taktiež neobsahuje deklaráciu funkcie `plugin_init`, ktorá je deklarovaná v samotnom rozhraní pre vytváranie zásuvných modulov pre GCC. Základnou časťou externých hlavičkových súborov ktoré nástroj používa sú gcc-plugin.h a plugin-version.h. Tieto hlavičkové súbory sú jadrom pri vytváraní zásuvných modulov. Okrem toho nástroj používa hlavičkové súbory pre prácu so štruktúrami gimple a tree. Z bežných knižničných hlavičkových súborov nástroj používa <iostream> <list> a <set>.

Súbor csigsafe.cc obsahuje všetky používané globálne premenné. Sem patrí najdôležitejší objekt s menom `plugin_core`, ktorý je inštanciou triedy `css_plugin_core`. Zároveň je tento objekt definovaný ako static, čo zabraňuje jeho viditeľnosť mimo tento modul. Okrem toho obsahuje nástroj v tomto súbore radu konštantných premenných ako sú meno zásuvného modulu `plugin_name` a verziu `release_version`, ktoré sú typu static const char * const, čiže konštantný ukazovateľ na konštantný reťazec. Ďalej obsahuje konštantu `pseudo_errno`, ktorá je na reprazentáciu premennej `errno` prostredníctvom štruktúry `errno_var`.

Príznaky u triedy `css_function` sú realizované ako pole hodnôt typu bool. Prístup je realizovaný pomocou dvojice metód `get_flag` a `set_flag`. Tieto metódy vyžadujú index príznaku, ku ktorému majú pristúpiť. Tento index je neznamienkové celé číslo, čiže dátový typ unsigned int. Na špecifikovanie príznaku je preto vytvorená sada makier, ktoré reprezentujú číselné konštanty týchto makier. Tieto makrá je možné identifikovať pomocou ich

prefixu, ktorým je FLG_. Túto sadu makier aj s vysvetlením jednotlivých príznakov je možné vidieť v tabuľke 4.1.

Názov makra	číselná hodnota	Popis
FLG_SCANED	0	Tento príznak indikuje že funkcia už bola preskenovaná
FLG_IS_HANDLER	1	Tento príznak indikuje že funkcia bola detekovaná ako obslužná funkcia
FLG_IS_OK	2	Tento príznak indikuje že funkcia je bezpečná aby bola volaná z obsluhy signálov
FLG_NOT_SAFE	3	Tento príznak indikuje že došlo k detekovaniu nevhodnej funkcie, a preto nieje vhodné ju volať v obsluhe signálov
FLG_WAS_ERR	4	Tento príznak indikuje, že už došlo k detekcii a výpisu chyby.
FLG_FATAL	5	Tento príznak rozširuje príznak FLG_NOT_SAFE, a to tým, že určuje či volaná funkcia volá jednu z funkcii definovanou v zozname nevhodných funkcii
FLG_IS_EXIT	6	Tento príznak indikuje že funkcia končí bezpečným (z pohľadu obsluhy signálov) ukončením programu
FLG_CAN_BE_SETTER	7	Tento príznak indikuje že funkcia ešte môže nastavovať premennú `errno` zo svojho parametra (tento príznak je na začiatku nastavený ako pravda), to znamená že má stále zmysel to kontrolovať
FLG_IS_ERRNO_SETTER	8	Tento príznak indikuje že funkcia mení premennú `errno`, a dosádza do nej hodnotu z jedného parametra, funkcia tým pádom nahrádza priradenie do premennej `errno`
FLG_ERRNO_CHANGED	9	Tento príznak indikuje, že funkcia môže svojím zavolaním zmeniť premennú `errno`
FLG_OUT_OF_RANGE	10	Toto makro indikuje pretečenie rozsahu príznakov, to znamená, že číslo príznaku väčšie alebo rovné tomuto makru je neexistujúci príznak. Toto makro slúži len na kontrolu rozsahu

Tabuľka 4.1: Sada makier používaných na prístup k príznakom u funkcie `css_function` a význam týchto príznakov.

Ako je spomenuté v sekcii 3, nástroj klasifikuje funkcie do piatich skupín. Podľa týchto skupín musí nástroj nastavovať návratovú hodnotu u funkcii `scan_own_function` a funkcii `is_handler_ok_fnc`. Tieto návratové hodnoty sú typu int8_t. Aby bolo jednoduchšie reprezentovať čo ktorá konštanta znamená, nástroj používa sadu makier na reprezentáciu týchto hodnôt. Tieto makrá je možné identifikovať podla prefixu RC_. Túto sadu je možné vidieť v tabuľke 4.2.

Názov makra	číselná hodnota	Popis
RC_NOT_ASYNCH_SAFE	0	Funkcia nie je v zozname bezpečných, ale nie je ani v zozname nebezpečných funkcii
RC_ASYNCH_SAFE	1	Funkcia je bezpečná na volanie z obsluhy signálov
RC_ERRNO_CHANGED	2	Funkcia je bezpečná, ale môže zmeniť hodnotu premennej `errno`
RC_SAFE_EXIT	4	Funkcia je bezpečná, a zároveň nedôjde k návratu z nej (ukončenie procesu)
RC_ERRNO_SETTER	8	Funkcia funguje tak, že z parametra nastavuje hodnotu premennej `errno` (Niečo ako priradenie do premennej `errno`)
RC_ASYNCH_UNSAFE	-1	Funkcia je nevhodná aby bola volaná z obsluhy signálov
RC_CYCLIC	110	Špeciálny návratový kód indikujúci cyklické volanie funkcii

Tabuľka 4.2: Sada makier reprezentujúca návratové hodnoty z funkcii `scan_own_function` a `is_handler_ok_fnc` a ich význam.

Zoznamy knižných funkcii, ktoré nástroj pozná a vie rozlíšiť do skupín, sú implementované ako konštantné reťazce. Tieto reťazce sú pristupované prostredníctvom polí ukazovateľov na konštantný znak, čiže const char*. Vždy jedno pole reprezentuje jednu skupinu knižničných funkcii podľa rozdelenia uvedeného v kapitole 3. Z toho vyplýva, že nástroj obsahuje 4 takéto polia. Funkcia `is_handler_ok_fnc` obsahuje 3 z týchto polí. Štvrté pole je definované vo funkcii `is_handler_wrong_fnc`. Tieto polia sú definované ako statické, a ďalej sú počas analýzy nemenné. Veľkosť poľa pri prechádzaní nie je udaná konštantou, ale je dopočítaná zo skutočného počtu prvkov. Toto riešenie umožňuje jednoduchú zmenu samotných reťazcov, v prípade toho, že by bolo nutné nejakú ďalšiu funkciu do zoznamu pridať, alebo nejakú funkciu zo zoznamu odobrať.

Nástroj používa vlastnú štruktúru na identifikáciu premennej pri analýze CFG. Na to slúži štruktúra `errno_var`. Táto štruktúra obsahuje dve položky, a to `id` typu unsigned int a `name` typu const char *. Premenná je tým pádom identifikovaná vždy dvojicou svoje meno a svoje ID.

Na preklad premennej z typu `tree` na typ `errno_var` slúži funkcia `tree_to_errno_var`. Táto funkcia berie jeden parameter typu `tree` a vracia hodnotu typu `errno_var`. ID funkcie je možné zistiť pomocou volania funkcie `DECL_UID` a meno pomocou funkcie `get_name`.

Špeciálnym prípadom je uloženie premennej `errno` do tejto štruktúry. To je realizované tak, že ukazovateľ `name` obsahuje hodnotu `nullptr`. V prípade premennej `errno` slúži položka `id` na identifikovanie dosadenia hodnoty parametru funkcie do premennej `errno`. To znamená že ak po výstupe z funkcie pri CFG analýze dôjde k tomu, že táto reprezentácia premennej `errno` obsahuje v položke `id` číslo rôzne od 0, tak je funkcia považovaná za "errno setter", čiže nastavuje premennú `errno` z parametra. Pozícia parametra, ktorý túto hodnotu nastavuje je vždy daná hodnotou položky `id` - 1. Keďže nástroj využíva množinu týchto štruktúr na svoju prácu, tak používa objekt ktorý je inštanciou triedy set. Táto trieda je presnejšie usporiadaná množina, čiže nezávisle v akom poradí boli prvky pridané, budú vždy v rovnakom poradí. Z tohto dôvodu je nutné vedieť určiť, ktorá hodnota v tejto štruktúre je "menšia" pri porovnávaní dvoch hodnôt. To znamená, že bolo nutné definovať bool operátor menší nad touto štruktúrou. Hodnota reprezentujúca premennú `errno` je vždy menšia ako ľubovolná ďalšia premenná. Premenné sú následne usporiadané primárne podla hodnoty id, a sekundárne podľa výsledku z funkcie `strcmp`.

Nástroj je vytvorený pre firmu Red Hat, ktorá mu dala názov *csigsafe*. Primárne bude používaný prostredníctvom nástroja csmock, ktorý slúži na testovanie sRPM balíkov, nachádzajúcich sa v linuxových distribúciach RHEL, Fedora, a Centos. Pre toto bol nástroj *csigsafe* primárne vyvíjaný a testovaný na distribúcii Fedora 28. Následne vznikol repozitár pre distribúcie Fedora 28, Fedora 29, Fedora 30 a Fedora Rawhide do ktorého bol pridaný aj nástroj *csigsafe*. Tento repozitár je verejný a voľne dostupný, čo umožňuje ľahké použitie analyzátora *csigsafe*. Nástroj csmock obsahuje profil na použitie analyzátora *csigsafe*.

Nástroj csmock pri analýze CFG používa vlastné inštrukcie. Bolo by možné vykonávať CFG analýzu aj nad samotnými Gimple tvrdeniami, ale keďže CFG analýza môže nad jedným blokom prechádzať viac ako jeden krát, bolo by nutné prechádzať a analyzovať samotné Gimple tvrdenia viac ako jeden krát. Keďže vlastné inštrukcie sú jednoduchšie ako Gimple tvrdenia, ktoré reprezentujú, je ľahšie nad nimi vykonávať analýzu. Analýza Gimple tvrdení dokonca prebieha ešte pred tým, ako je vôbec CFG analýza zahájená. Preto v čase, keď sú tieto tvrdenia analyzované, tak sú prevedené do zjednodušenej formy, a tou sú práve inštrukcie zásuvného modulu uložené v štruktúre `instruction`. Tieto inštrukcie sa rozlišujú pomocou prvku ic (anglicky insturction code, v preklade kód inštrukcie). Tento prvok v štruktúre `instruction`, viď kapitola 3, je typu enum `instruction_code`. Určuje presný typ inštrukcie, a na základe tohoto prvku sa môže meniť aj význam ďalších prvkov v tejto štruktúre. Tieto hodnoty kódu inštrukcie je možné vidieť v tabuľke 4.3.

Názov enum hodnoty	Popis
IC_CHANGE_ERRNO	Táto hodnota reprezentuje Gimple tvrdenie, ktoré môže, alebo zmení hodnotu premennej `errno`. Funkcia, ktorá môže zmeniť premennú je zapamätaná prostredníctvom prvku `var`.
IC_SAVE_ERRNO	Táto hodnota reprezentuje dosadenie hodnoty z premennej `errno` do nejakej ďalšej premennej. Premenná je zapamätaná prostredníctvom prvku `var`.
IC_SAVE_FROM_VAR	Táto hodnota reprezentuje Gimple tvrdenie, kde je dosadená hodnota z niektorej z premenných, do ktorých bola predtým uložená hodnota z premennej `errno`. Čiže zálohovanie hodnoty premennej `errno` z premennej, ktorá túto hodnotu už obsahuje. Premenná z ktorej je záloha prevedená je zapamätaná v prvku `from_var` a premenná do ktorej je hodnota uložená je identifikovaná prostredníctvom prvku `var`.
IC_DESTROY_STORAGE	Táto hodnota reprezentuje dosadenie nejakej hodnoty do premennej, do ktorej bola predtým zálohovaná hodnota premennej `errno`. Čiže inými slovami ide o znehodnotenie zálohy hodnoty premennej `errno`. Premenná, ktorá bola znehodnotená je zapamätaná v prvku `var`.
IC_RESTORE_ERRNO	Táto hodnota reprezentuje dosadenie hodnoty do premennej `errno`, z niektorej premennej, do ktorej bola premenná `errno` predtým zálohovaná. Čiže ide o obnovenie počiatočnej hodnoty premennej `errno`. Premenná, z ktorej bola záloha vykonaná je reprezentovaná v prvku `var`.
IC_SET_FROM_PARM	Táto hodnota reprezentuje dosadenie hodnoty do premennej `errno` z niektorého z parametrov funkcie. Táto inštrukcia je používaná na lokalizáciu funkcie, ktorá slúži ako dosadenie do premennej `errno`. Pozícia parametra, z ktorého je hodnota do premennej `errno` uložená je reprezentovaný v prvku `param_pos`.
IC_EXIT	Táto hodnota reprezentuje volanie niektorej z bezpečných ukončovacích funkcii. Môže ísť aj o vlastnú ukončovaciu funkciu, definovanú v aktuálnom module. K tejto inštrukcii nie sú potrebné ďalšie údaje.
IC_DEPEND	Táto hodnota slúži len ako takzvaný "placeholder", čiže inštrukcia, ktorá "drží miesto", ak by ju bolo neskôr nutné prepísať. Táto hodnota sa používa v prípade keď sa nastavuje závislosť na volanie nejakej funkcie, keďže zatiaľ nie je známe, čo za inštrukciu daná funkcia môže nastaviť. Ak sa táto hodnota vyskytuje v CFG analýze, tak sa ignoruje.

Tabuľka 4.3: Hodnoty typu enum reprezentujúce kód inštrukcie v štruktúre `instruction`

Nástroj sa používa ako každý iný zásuvný modul do GCC. Pri bežnom preklade zdrojových súborov pomocou prekladača GCC stačí uviesť argument `-fplugin=` a zaň zadať cestu k nástroju.
Napríklad je nástroj možné použiť takto:

```
gcc -c ./src.c -o src.o -fplugin=./csigsafe.so
```

Tento príklad prekladá zdrojový súbor src.c na obiektový súbor src.o. Zásuvný modul sa nachádza v aktuálnom adresáry zároveň zo zdrojovým súborom src.c.

Každá chyba lokalizovaná popri skenovaní je zapamätaná prostredníctvom zoznamu `err_log`, pre chybu spojenú z nereentrantnov funkciou, a prostredníctvom dvojice premenných `errno_loc` a `errno_fnc` pre zapamätanie si lokácie, a aj samotného Gimple tvrdenia ktoré môže zmeniť premennú `errno`. Príklad výstupu nástroja je možné vidieť na obrázku 4.1 pre prípad použitia nereentrantnej funkcie. Prvá časť ukazuje použitie funkcie printf, ktorá je považovaná za nevhodnú. V druhej časti obrázka je vidieť použitie externej funkcie, ktorá nieje definovaná v aktuálnom module. Nástroj na ňu aj tak upozorňuje, ale pridáva do výpisu reťazec “possibly”, ktorý indikuje že nie je isté či táto funkcia je nevhodná. Tento prípad vyžaduje ďalšiu pozornosť ľudského faktoru. Na obrázku 4.2 je možné vidieť príklad výstupu pre zmenu premennej `errno`. Taktiež je možné vidieť samotné volanie funkcie, ktoré túto zmenu spôsobilo tranzitívne. Pre tento prípad je to funkcia `kill`.

```
g++ -c -flto ./src.c -o /dev/null -fplugin=./../../csigsafe.so
./src.c: In function 'int main()':
./src.c:23:6: warning: asynchronous-unsafe function 'fnc1' in signal handler 'handler' [-fplugin=csigsafe]
  fnc1();
  ~~~~^~
./src.c:29:8: note: function 'fnc1' calls function 'fnc2'
    fnc2();
    ~~~~^~
./src.c:33:8: note: function 'fnc2' calls function 'fnc3'
    fnc3();
    ~~~~^~
./src.c:37:10: note: function 'fnc3' calls function 'printf'
    printf("text");
    ~~~~~~^~~~~~~~
./src.c:37:10: note: function 'printf' is not known to be async-signal-safe
./src.c:24:6: warning: possibly asynchronous-unsafe function 'fnc5' in signal handler 'handler' [-fplugin=csigsafe]
  fnc5();
  ~~~~^~
./src.c:46:8: note: function 'fnc5' calls function 'fnc6'
    fnc6();
    ~~~~^~
./src.c:50:14: note: function 'fnc6' calls function 'extern_fnc'
    extern_fnc();
    ~~~~~~~~~~^~
./src.c:50:14: note: function 'extern_fnc' is not known to be async-signal-safe
[dkozovsk@dkozovsk debug]$
```

Obr. 4.1: Príklad chybového výpisu zásuvného modulu na volanie nereentrantných funkcii.

```
g++ -c -flto ./src.c -o /dev/null -fplugin=./../../csigsafe.so
./src.c: In function 'int main()':
./src.c:19:6: warning: errno may be changed in signal handler 'handler' [-fplugin=csigsafe]
  fnc1();
  ~~~~^~
./src.c:24:8: note: function 'fnc1' may change errno
    fnc2();
    ~~~~^~
./src.c:28:8: note: function 'fnc2' may change errno
    fnc3();
    ~~~~^~
./src.c:32:8: note: function 'fnc3' may change errno
    kill(0,SIGKILL);
    ~~~~^~~~~~~~~~
```

Obr. 4.2: Príklad chybového výpisu zásuvného modulu na zmenu premennej `errno`.

Nástroj používa na svoju činnosť okrem vlastných štruktúr aj štruktúry zo samotnej vnútornej reprezentácie GCC. Toto riešenie spôsobuje to, že pri každej zmene vnútorných štruktúr GCC bude nutné upraviť samotné algoritmy na analýzu, na rozdiel od riešenia, kde najprv dôjde k prekladu zo štruktúr GCC do vlastných štruktúr. Toto riešenie je použité pre to, že prechod štruktúrami GCC je tak či tak nevyhnutný, a mal by znížiť pamäťovú náročnosť aplikácie. Pri dnešnom výkone počítačov sa toto môže javiť ako bezvýznamné, ale mohlo by sa to priaznivo prejaviť pri veľkých projektoch.

4.2 Testovanie zásuvného modulu

Nástroj bol popri vývoji testovaný prostredníctvom vlastných testov. Tieto testy boli navrhnuté podľa možných chýb, ktoré sa môžu v obslužných funkciách vyskytovať. Pre každý typ chyby a jej možný spôsob vykonania bol vždy vytvorený krátky zdrojový kód. Postupne bol kód vylaďovaný aby bol schopný úspešne prejsť všetkými z týchto navrhnutých testov.

Počiatočný algoritmus na výpočet zmeny premennej `errno` bol sekvenčný, a nebol schopný správne detekovať chybu spôsobenú touto zmenou. Pseudokód tejto testovacej obslužnej funkcie je možné vidieť v kóde 4.1. V tomto kóde môže dôjsť k návratu po zmene premennej `errno`, ktorú môže spôsobiť volanie funkcie `kill`. Vďaka tomuto testu bol algoritmus na detekciu zmeny premennej `errno` zmenený, a výsledný algoritmus je spomínaný v sekcii 3.2.

```
void handler(int signum)
{
   while(something)
   {
      if (something_else)
         return;
        ...
      kill(SIGKILL,0);
   }
   _exit(9);
}
```

Kód 4.1: Pseudokód testovanej obslužnej funkcie ktorá môže zmeniť premennú `errno`

Počas testovania na Open Source projektoch bolo zistené že použitie attribute cleanup na obnovenie premennej `errno` je nástrojom považované za chybu, aj keď v skutočnosti o chybu nešlo. Toto bolo spôsobené tým, že vnútorne sa toto javí ako volanie funkcie tesne pred návratom z funkcie. Preto bolo tým pádom nutné vedieť detekovať funkciu, ktorá nastavuje premennú `errno` zo svojho parametra. Táto funkcia tým pádom nahrádza priradenie do premennej `errno`. Riešením bolo vytvorenie štruktúry, `setter_function`, viď kapitola 3. Testovací kód na testovanie použitia attribute cleanup je možné vidieť v kóde 4.2. Použitie, ktoré je možné v tomto kóde vidieť je konkrétne prevzané z Open Source projektu systemd. Keďže je toto riešenie používané v praxi, bolo nutné vedieť sa s tým vysporiadať a správne určiť, či sa o chybu jedná, alebo nie.

```
#define _cleanup_(x) __attribute__((cleanup(x)))

static inline void _reset_errno_(int *saved_errno) {
       errno = *saved_errno;
}

#define PROTECT_ERRNO _cleanup_(_reset_errno_) \\
__attribute__((unused)) int _saved_errno_ = errno

void handler(int signum)
{
   PROTECT_ERRNO;
   kill(signum,0);
}
```

Kód 4.2: Testovací kód na použitie attribute cleanup čiže volanie funkcie tesne pred návratom z funkcie

Testovanie je automatizované použitím Makefile. Na spustenie testov stačí v adresári obsahujúcom celý projekt použiť príkaz `make test`. Ten ďalej používa nástroj ctest, ktorý je rozšírenie nástroja cmake. Projekt obsahuje v priečinku tests súbor CMakeLists.txt, ktorý obsahuje ako, a ktoré testy sa majú vykonávať. Nástroj ctest následne na spustenie jednotlivých testov používa skript runtest.sh. Tento skript spustí požadovaný test, tým že spustí zavolá príkaz make v priečinku z príslušným testom. Následne upraví výstup testu, ktorý je uložený v súbore test.err. Po úprave na požadovaný formát porovná ďalej výstup z očakávaným výstupom. To je vykonané tak že sa použie nástroj `diff` na porovnanie súborou test.err a test.exp. Z toho vyplýva, že každý test obsahuje vo svojom adresári súbor Makefile, zdrojové súbory na testovanie a očakávaný výstup test.exp.

Následne bol nástroj testovaný na voľne dostupných projektoch ako sú systemd, nano, bash, dash, a ďalšie. Pomocou tohto nástroja boli detekované chyby v projekte systemd. Týmito chybami boli volania nereentrantných funkcii. V obslužnej funkcii `sig_alrm` bola použitá fukcia `exit()`. Táto funkcia nie je považovaná za bezpečnú pre jej vyprázdňovanie bufferov, a preto by nemala byť použitá v obslužnej funkcii. Táto funkcia bola nahradená bezpečnou variantou ukončovacej funkcie, a to funkciou `_exit()`. Taktiež v obslužnej funkcii `sigbus_handler` Je použitá funkcia `page_size`, ktorá pri prvom volaní volá nereetnrantnú funkciu `sysconf`, a výsledok si pamätá v statickej premennej. Pri každom ďalšom volaní vraciu už uloženú hodnotu. Táto chyba bola opravená tým, že sa zaručí volanie tejto funkcie ešte pred nastavením obslužnej funkcie. Toto riešenie síce chybu odstráni, ale táto chybu bude aj naďalej detekovaná pomocou nástroja spomínaného v tejto práci.

Nástroj sa ukázal byť docela spoľahlivý. Keď zanedbáme chyby označené ako “možné”, to znamená chyby spojené zo zmenou premennej `errno` alebo volaním nereentrantnej funkcie zo zoznamu nevhodných funkcii, tak chyby, ktoré nahlási sa tam často aj nachádzajú. V prípade použitia moc netriviálnych a nezvyčajných spôsobov je možné nástroj oklamať, a následne sa môžu chyby nahlásené nástrojom mýliť.

Názov projektu	Počet varovaní spôsobených volaním nereentrantnej funkcie	Počet varovaní spôsobených neznámou externou funkciou	Počet varovaní spôsobených zmenou premennej `errno`
arptools	0	0	0
bash	2	5	0
bsh	0	0	0
bzip2	3	5	0
coreutils	3	3	3
csmock	0	0	0
dash	0	0	0
dia	0	0	0
diffutils	0	1	0
drush	0	0	0
findutils	0	0	0
firewalld	0	0	0
flatpack	0	0	0
gawk	0	12	1
geany	0	0	0
gimp	0	2	0
git	0	0	1
glib	0	0	0
gmime	0	0	0
hexchat	0	0	0
hydra	4	4	0
kate	0	0	0
kcalc	0	0	0
kdelibs	1	35	4
kernel-tools	13	39	1
konsole	0	0	0
ksh	5	13	2
mock	0	0	0
nano	2	6	2
pidgin	3	8	1
readline	0	1	0
systemd	0	18	1
tsch	0	1	0
unzip	0	0	0
util-linux	9	16	7
zip	4	1	0
zsh	0	18	1

Tabuľka 4.4: Prehlad výsledkov testovania zásuvného modulu nad voľne dostupnými projektami

V tabuľke 4.4 je možné vidieť výsledky testovania rôznych projektov a nástrojov. Výsledky ukazujú, že z testovaných 37 projektov bola lokalizovaná chyba v 19 projektoch. Dokopy bolo na týchto projektoch lokalizovaných 49 chýb volania nereentrantnej funkcie, 22 chýb možnej zmeny premennej `errno` a 166 chýb spôsobených neznámou externou funkciou. Dokopy to činí 237 chýb lokalizovaných v týchto projektoch pomocou nástroja *csigsafe*. Z výsledkov vyplýva, že tieto chyby, spojené z obsluhou signálov, sa bežne nachádzajú aj v projektoch, ktoré sú dostupné a bežne používané. Preto je nástroj *csigsafe* použiteľný v bežnej praxi na nachádzanie týchto chýb.

Kapitola 5

Záver

Výsledkom tejto práce je zásuvný modul *csigsafe* pre prekladač GCC, ktorý používa metódu statickej analýzy programov control flow analýzu k detekcii chýb v obsluhe signálov podla normy POSIX. Tento nástroj slúži na analýzu zdrojových súborov v jazyku C, respektíve C++. Zaručene funguje na prekladači GCC verzii 8.1.0 až 8.3.1, na ktorých bol vytváraný a testovaný. Mal by fungovať aj na verzii 9.1.0. Z príchodom novších verzii môže byť nutné modul upraviť. Zásuvný modul *csigsafe* musí byť vždy prekladaný k príslušnej verzii prekladača GCC, na ktorú má byť používaný. Modul bol vyvinutý pre firmu Red Hat, ktorá ho už aj zahrnula do primárne používaného testovacieho nástroja csmock. Analyzátor *csigsafe* je aj dostupný v repozitári pre linuxové distribúcie Fedora 28, Fedora 29, Fedora 30 a Fedora Rawhide. Analyzátor *csigsafe* zjednodušuje hľadanie chýb v obsluhe signálov, keďže tieto chyby sa prejavujú nedeterministicky.

Nástroj bol testovaný pomocou vlastných testov, ktoré vyobrazujú bežné chyby spôsobené v obsluhe signálov, a na tieto testy bol primárne vyľaďovaný. V druhej fáze testovania bol nástroj použitý na niekoľkých projektoch z voľne dostupnými zdrojovými súbormi (Open Source). Analyzátor *csigsafe* bol ďalej prispôsobovaný podľa požiadaviek, ktoré vyplývali z reálnych spôsobov obsluhy signálov. Nástroj sa ukázal ako užitočný, keďže bol schopný odhaliť chyby v známych projektoch, ako je napríklad systemd. Analyzátor *csigsafe* je úspešný na odhalenie chýb spojených s volaním nereentrantných funkcii a zmenou premennej `errno`. Analyzátor je schopný analyzovať iba jeden modul. Pri použití externých funkcii nieje nástroj schopný určiť, či je funkcia vhodná, keďže nemá prístup k jej kódu. V takomto prípade nástroj *csigsafe* na funkciu upozorňuje ako na nevhodnú s prídavkom, že ide o možnú chybu. Keď zásuvný modul *csigsafe* objaví chybu spojenú so zmenou premennej `errno`, alebo volaním nereentrantnej knižničnej funkcie, čiže nie prípady keď upozorňuje na možnú chybu, tak sa poväčšine nemýli.

Testy zásuvného modulu *csigsafe* na voľne dostupných projektoch poukázali na 237 chýb v 37 projektoch. Z toho bolo vážnych chýb iba 71. Ostatné chyby boli označené ako možné, čiže 166 možných chýb, ktoré by vyžadovali ďalšiu pozornosť ľudského faktoru, aby boli buď potvrdené, alebo vyvrátené.

Do budúcna by bolo možné nástroj upraviť tak, aby bol schopný analyzovať celý program, nie len jednotlivé moduly. Táto úprava by však vyžadovala značnú zmenu koncepcie programu. Následne by bolo vhodné *csigsafe* lepšie prispôsobiť na analýzu C++ kódu. *csigsafe* je síce schopný analýzu kódu C++ vykonávať, ale keďže firma Red Hat primárne vyžadovala analýzu kódu jazyka C, tak je *csigsafe* určený hlavne na analýzu kódu jazyka C. Preto nebolo vykonané hĺbkové testovanie na jazyk C++. V budúcnosti by bolo možné rozšíriť *csigsafe* o ďalšie pravidlá týkajúce sa správnej obsluhy signálov.

Literatúra

[1] Free Software Foundation, Inc.: *GCC, the GNU Compiler Collection*. [Online; navštíveno 13.03.2019].
URL http://gcc.gnu.org/

[2] Free Software Foundation, Inc.: *GNU Compiler Collection (GCC) Internals*. [Online; navštíveno 13.03.2019].
URL https://gcc.gnu.org/onlinedocs/gccint/

[3] Ibáñez, R. F.: *A simple plugin for GCC – Part 2* . [Online; navštíveno 02.04.2019].
URL https://thinkingeek.com/2015/08/16/simple-plugin-gcc-part-2/

[4] IEEE; The Open Group: *Standardfor InformationTechnology—Portable OperatingSystem Interface (POSIX®)Base Specifications, Issue 7*. The Open Grou, 2016, ISBN 1-937218-81-2.

[5] Kerrisk, M.: *The Linux Programming Interface*. No Starch press, 2010, ISBN 978-1-59327-220-3.

[6] Nielson, F.; Nielson, H. R.; Hankin, C.: *Principles of Program Analysis*. 2005, ISBN 978-3-662-03811-6.

[7] Pincar, J.; Britton, J.: *SIG30-C. Call only asynchronous-safe functions within signal handlers*. [Online; navštíveno 10.04.2019].
URL https://wiki.sei.cmu.edu/confluence/display/c/SIG30-C.+Call+only+asynchronous-safe+functions+within+signal+handlers

[8] Stevens, R.; Rago, S.: *Advanced Programming in the UNIX Environment 3rd Edition*. 2013, ISBN 978-0-321-63773-4.

[9] VictorDev; Venki: *Reentrant Function*. [Online; navštíveno 13.03.2019].
URL https://www.geeksforgeeks.org/reentrant-function/

Príloha A

Prehľad signálov definovaných v manuálových stránkach

Názov signálu	Číselná hodnota	Norma	Bežné chovanie	Popis
SIGABRT	6	POSIX.1-1990	ukončenie + výpis jadra	signál vyvolaný volaním funkcie `abort`
SIGALRM	14	POSIX.1-1990	ukončenie procesu	vypršanie časovača nastaveného pomocou funkcie `alarm`
SIGBUS	7	POSIX.1-2001	ukončenie + výpis jadra	chyba zbernice, zlý pamäťový prístup
SIGCHLD	17	POSIX.1-1990	ignorovaný	synovský proces bol ukončený alebo skončil
SIGCLD	-	-	ignorovaný	alternatívny názov pre SIGCHLD, čiže synovský proces bol ukončený alebo skončil
SIGCONT	18	POSIX.1-1990	pokračovanie procesu	pokračovanie pozastaveného procesu
SIGEMT	-	-	ukončenie procesu	emulator trap
SIGFPE	8	POSIX.1-1990	ukončenie + výpis jadra	výnimka pri počítaní z pohyblivou rádovou čiarkou
SIGHUP	1	POSIX.1-1990	ukončenie procesu	riadiaci proces alebo terminál bol ukončený, čiže bolo detekované takzvané “zavesenie”
SIGILL	4	POSIX.1-1990	ukončenie + výpis jadra	nevyhodnotiteľná inštrukcia

Tabuľka A.1: Zoznam signálov a ich význam, časť 1.

Názov signálu	Číselná hodnota	Norma	Bežné chovanie	Popis
SIGINFO	-	-	ukončenie procesu	synonym pre SIGPWR, čiže chyba napájania
SIGINT	2	POSIX.1-1990	ukončenie procesu	prerušenie procesu spôsobené klávesnicou, čiže ide o užívatelom spôsobený signál
SIGIO	29	-	ukončenie procesu	vstupno-výstupná operácia je dostupná
SIGIOT	6	-	ukončenie + výpis jadra	synonym pre SIGABRT, čiže signál vyvolaný volaním funkcie `abort`
SIGKILL	9	POSIX.1-1990	ukončenie procesu	signál indikujúci "zabitie" aplikácie, bezvýhradné ukončenie aplikácie
SIGLOST	-	-	ukončenie procesu	Stratení zámok na súbore (nepoužitý)
SIGPIPE	13	POSIX.1-1990	ukončenie procesu	Rozbitá "rúra" (jednosmerná komunikácia, umožňujúca prenášať dáta z jedného procesu do druhého (navonok sa takáto "rúra" javí ako súbor)): zápis do "rúry", z ktorej nikto nečíta
SIGPOLL	29	POSIX.1-2001	ukončenie procesu	synonym pre SIGIO, čiže vstupno-výstupná operácia je dostupná
SIGPROF	27	POSIX.1-2001	ukončenie procesu	časovač profilovania uplinul
SIGPWR	30	-	ukončenie procesu	chyba napájania
SIGQUIT	3	POSIX.1-1990	ukončenie + výpis jadra	ukončenie aplikácie spôsobené klávesnicou, ide o signál spôsobený užívatelom
SIGSEGV	11	POSIX.1-1990	ukončenie + výpis jadra	neplatná referencia do pamäte, nevhodný prístup do pamäte
SIGSTKFLT	16	-	ukončenie procesu	porucha zásobníka na koprocesore (nepoužitý)
SIGSTOP	19	POSIX.1-1990	zastavenie procesu	pozastavenie procesu

Tabuľka A.2: Zoznam signálov a ich význam, časť 2.

Názov signálu	Číselná hodnota	Norma	Bežné chovanie	Popis
SIGTSTP	20	POSIX.1-1990	zastavenie procesu	stop spôsobený terminálom
SIGSYS	31	POSIX.1-2001	ukončenie + výpis jadra	zlé systémové volanie
SIGTERM	15	POSIX.1-1990	ukončenie procesu	signál na ukončenie
SIGTRAP	5	POSIX.1-2001	ukončenie + výpis jadra	breakpoint trap
SIGTTIN	21	POSIX.1-1990	zastavenie procesu	terminálový vstup pre proces na pozadí
SIGTTOU	22	POSIX.1-1990	zastavenie procesu	terminálový výstup pre proces na pozadí
SIGUNUSED	31	-	ukončenie + výpis jadra	synonymum k SIGSYS, čiže zlé systémové volanie
SIGURG	23	POSIX.1-2001	ignorovaný	urgentná podmienka na sokete
SIGUSR1	10	POSIX.1-1990	ukončenie procesu	signál pre užívateľské použitie číslo 1
SIGUSR2	12	POSIX.1-1990	ukončenie procesu	signál pre užívateľské použitie číslo 2
SIGVTALRM	26	POSIX.1-2001	ukončenie procesu	alarm spôsobený virtuálnym časovačom
SIGXCPU	24	POSIX.1-2001	ukončenie + výpis jadra	prekročený časový limit CPU
SIGXFSZ	25	POSIX.1-2001	ukončenie + výpis jadra	prekročený limit veľkosti súboru
SIGWINCH	28	-	ignorovaný	signál upozorňujúci na zmenu veľkosti okna

Tabuľka A.3: Zoznam signálov a ich význam, časť 3.