ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Hledaní zranitelností nad LLVM mezikódem
Student:	Bc. Jan Brož
Vedoucí:	Ing. Josef Kokeš
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra informační bezpečnosti
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Analyzujte běžné bezpečnostní chyby v programech psaných v kompilovaných jazycích.
Seznamte se s mezikódy používanými v překladačích.
Nastudujte mezikód LLVM, jeho instrukce a programové konstrukce. Prozkoumejte možnosti jeho statické analýzy s cílem detekce zranitelností.
Navrhněte algoritmy pro detekci vybraných zranitelností v LLVM mezikódu.
Naimplementujte vybrané algoritmy a otestujte jejich efektivitu na vhodných vzorcích.
Diskutujte své výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 12. února 2019

Diplomová práce

Hledání zranitelností nad LLVM mezikódem

Bc. Jan Brož

Katedra informační bezpečnosti
Vedoucí práce: Ing. Josef Kokeš

9. května 2019

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2019 ....

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Jan Brož. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Brož, Jan. *Hledání zranitelností nad LLVM mezikódem.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce rozebírá možnosti statické analýzy programů za účelem hledání zranitelností, které vznikají častými programátorskými chybami. Nejprve jsou popsány tyto chyby, jejich dopady a světové statistiky. Jsou také shrnuty současné existující postupy na odhalování těchto chyb a jejich nedostatky. Následně je navržena metoda zpětného hledání původu dat sledováním datových závislostí v LLVM mezikódu, která je poté aplikována na detekování injekce příkazů či formátovacího řetězce a některé případy přetečení bufferu. Jsou rozebrány nejčastější typy falešných poplachů, které tato metoda produkuje, a navrženy možnosti jejich eliminace. Metoda je implementována v C++ pomocí LLVM frameworku a otestována na vybraných vzorcích zranitelných programů.

Klíčová slova detekce zranitelností, injekce příkazu, přetečení bufferu, statická analýza, LLVM, analýza datových toků

Abstract

Thesis examines possibilities of static analysis for the purpose of searching vulnerabilities, which arise from common programming mistakes. Firstly we describe those mistakes, their impacts and worldwide statistics. We also summarize existing methods for detection of such mistakes and their shortcomings. Thereafter we suggest a method of backward finding a data origin by following data dependancies in LLVM intermediate code, which is then applied on detection of injection of commands or format strings and certain cases of buffer overflow. We examine types of false positives this method produces and we suggest ways to mitigate them. The method is implemented in C++ using LLVM framework and tested on selected samples of vulnerable programs.

Keywords vulnerability detection, command injection, buffer overflow, static analysis, LLVM, data flow analysis

Obsah

Seznam ukázek kódu **1**

Úvod **3**

1 Popis problému **5**
- 1.1 Softwarové chyby a jejich dopady . . . 5
- 1.2 Statická vs. dynamická analýza . . . 10
- 1.3 Možnosti statické analýzy . . . 11
- 1.4 Zdrojový kód, strojový kód a meziformy . . . 14

2 Volba meziformy pro analýzu **17**
- 2.1 Tříadresový kód a SSA . . . 17
- 2.2 GENERIC a GIMPLE . . . 20
- 2.3 LLVM . . . 21
- 2.4 Rozhodnutí . . . 23

3 Analýza datových toků **25**
- 3.1 Metoda zpětného hledání původu dat . . . 25
- 3.2 Detekce injekce příkazu či formátovacího řetězce . . . 32
- 3.3 Detekce přetečení bufferu . . . 32

4 Omezení falešných poplachů **35**
- 4.1 Pořadí provádění . . . 35
- 4.2 Sanitace vstupních dat . . . 37
- 4.3 Podmínky vylučující provedení . . . 38

5 Implementace **41**
- 5.1 Využití LLVM API pro hledání původu dat . . . 41
- 5.2 Pole na zásobníku, na haldě a globální . . . 43
- 5.3 Znovupoužití výsledků . . . 43

5.4 Intermodulární vyhledávání 44
5.5 Konfigurační soubor 44

6 Testování a výsledky **45**
6.1 Juliet Test Suite 45
6.2 C Test Suite for Source Code Analyzer v2 46
6.3 Zhodnocení 47

7 Možnosti budoucího rozšíření **49**
7.1 Analýza spustitelných souborů pomocí dekompilace do LLVM . 49
7.2 Abstrakce nad operacemi datových toků 50

Závěr **53**

Literatura **55**

A Seznam použitých zkratek **59**

B Obsah přiloženého CD **61**

Seznam obrázků

1.1 Příklad přetečení bufferu (převzato z [10]) 7
1.2 Formátovací řetězec a argumenty 9
1.3 Příklad datového toku pro délku pole 12
1.4 Příklad symbolického provádění 14
1.5 Architektura kompilátorů 15

2.1 Podoba výrazu v paměti 23

3.1 Příklad datových toků v kódu 3.1 26
3.2 Ukázka zpětného průchodu (fáze 1) 27
3.3 Ukázka hledání míst zápisu (fáze 2) 28
3.4 Příklad běhu zpětného hledání zdrojů 31
3.5 Příklad kontroly správné velikosti pole 33

4.1 Basic bloky funkce sum_abs 36
4.2 Automat pro konstantu 37
4.3 Automat pro vstup 37
4.4 Automat po nahrazení znaků a spojení 38

7.1 Společný abstraktní datový tok 51

Seznam tabulek

1.1 25 nejhorších softwarových zranitelností podle MITRE [4] 6
1.2 10 nejhorších softwarových zranitelností podle OWASP [5] 6

6.1 Výsledky pro sadu Juliet Test Suite 46
6.2 Matice záměn pro sadu C Test Suite v2 47
6.3 Matice záměn pro sadu C Test Suite v2 - v procentech 47

7.1 Porovnání existujících dekompilátorů do LLVM 50

Seznam ukázek kódu

1.1 Příklad přetečení bufferu 7
1.2 Příklad injekce příkazu 8
1.3 Příklad injekce formátovacího řetězce 9
1.4 Příklad problémového datového toku 12
1.5 Příklad analýzy rozsahů 13
2.1 LLVM kód výrazu 22
2.2 Konstrukce výrazu 22
3.1 Příklad datových toků při práci s řetězci 25
3.2 Příklad mezifunkčních datových závislosti 27
3.3 Příklad mezifunkčních datových závislosti 28
3.4 Příklad běhu zpětného hledání zdrojů 30
3.5 Příklad kódu s posunutým ukazatelem 33
7.1 Kód C s datovými závislostmi 51
7.2 Kód C++ s datovými závislostmi 51

Úvod

Software je všude kolem nás, v našich telefonech, automobilech, domácích spotřebičích, ale také na daleko kritičtějších místech jako jsou letadla, elektrárny či nemocnice. Na správné a bezpečné funkci softwaru mohou v krajních případech záviset lidské životy.

Produkce kódu a složitost počítačových systémů rapidně roste. Bohužel ekonomický tlak a nedostatečná legislativa způsobují, že zabezpečení systémů se v rámci úspory nákladů podceňuje. Zároveň velká část programátorů nemá dostatečné vzdělání v oblasti počítačové bezpečnosti a bezpečného programování na to, aby dokázali rozpoznat rizikové situace a vymyslet správné a nenapadnutelné řešení. Dopady vidíme např. na statistikách od vyhledávače CVE Details [8], podle kterého počet objevených softwarových zranitelností za posledních 5 let vzrostl na dvojnásobek.

V současné době se prevence proti chybám v kódu provádí především manuální kontrolou (známý anglický výraz „code review“). Bohužel tento způsob je velmi časově náročný a vyžaduje programátory s vysokou úrovní zkušeností. Často se tedy stává, že firmy při blížících se milnících tento krok opomíjejí, aby stíhaly dokončit slíbenou funkcionalitu.

Automatizované nástroje na detekci zranitelností v systémech sice existují (např. Metasploit), avšak ty většinou hledají již známé zranitelnosti objevené v konkrétní verzi konkrétního programu. Hledání nových ještě neobjevených zranitelností v programech je zatím ve fázi výzkumu a kvalitní prakticky použitelné nástroje nabízejí pouze drahá komerční řešení.

Ze všech těchto důvodů je na místě zabývat se automatickým hledáním zranitelností v programech, ať už ve zdrojovém kódu nebo ve spustitelných souborech.

Kapitola **1**

Popis problému

V této kapitole rozebereme stav softwaru z hlediska bezpečnosti a možnosti detekce zranitelností.

1.1 Softwarové chyby a jejich dopady

Kategorizací a ohodnocením programátorských chyb s bezpečnostními dopady se dlouhodobě zabývá organizace MITRE. Ta udržuje a aktualizuje seznam CWE („Common Weakness Enumeration") [3], který obsahuje formální definice zranitelností a jejich ohodnocení podle četnosti výskytů, jednoduchosti zneužití a kritičnosti následků. Přepis z jejich dokumentu „Top 25 Most Dangerous Software Errors" [4] vidíme v tabulce 1.1.

Pro srovnání ještě můžeme nahlédnout do „Top 10" od organizace OWASP [5]. Ten se sice zabývá pouze zranitelnostmi webových stránek, ale jsou zde vidět určité společné trendy. Jejich žebříček je v tabulce 1.2

Jak vidíme, uvedené chyby jsou v zásadě dvou typů:

- chyby v návrhu,
- chyby v implementaci.

Do návrhových chyb spadá chybějící nebo nedostatečná autentizace/autorizace, chybějící nebo slabé šifrování, spouštění s příliš velkým oprávněním apod. Tyto chyby jsou obecně velmi složité na detekci. Vyžadují zamyšlení nad problémem a vytušení problematických aspektů na základě zkušeností. Takové případy by šly hledat jedině pokročilými metodami strojového učení a umělé inteligence.

Typické implementační chyby jsou chyby při práci s buffery, chyby v aritmetických výpočtech (ztráta přesnosti nebo přetečení) a injekce příkazů, kódu, formátovacích řetězců či cest k souborům. Většina z těch z žebříčků je způsobena nedostatečným ošetřením vstupů a porušováním jednoho z hlavních pravidel bezpečného kódu – **nikdy nemíchat kód a data**. Toto pravidlo bude

#	ID	název
1	CWE-89	injekce SQL příkazu
2	CWE-78	injekce OS příkazu
3	CWE-120	kopírování bufferu bez kontroly hranic
4	CWE-79	Cross-Site Scripting
5	CWE-306	chybějící autentizace
6	CWE-862	chybějící autorizace
7	CWE-798	pevně daná hesla nebo klíče
8	CWE-311	chybějící šifrování citlivých dat
9	CWE-434	upload neověřeného souboru
10	CWE-807	bezpečnostní rozhodnutí podle nedůvěryhodného vstupu
11	CWE-250	spouštění se zbytečně velkým oprávněním
12	CWE-352	Cross-Site Request Forgery
13	CWE-22	neomezená cesta při přístupu do vyhrazené složky
14	CWE-494	stahování kódu bez ověření autentičnosti
15	CWE-863	nesprávná autorizace
16	CWE-829	vkládání funkcionality z nedůvěryhodného zdroje
17	CWE-732	nesprávné přiřazení oprávnění
18	CWE-676	použití potenciálně nebezpečné funkce
19	CWE-327	použití prolomeného nebo slabého šifrování
20	CWE-131	nesprávné určení velikosti bufferu
21	CWE-307	neomezené pokusy o přihlášení
22	CWE-601	neomezené přesměrování URL
23	CWE-134	neověřený formátovací řetězec
24	CWE-190	přetečení celých čísel
25	CWE-759	hashování hesel bez soli

Tabulka 1.1: 25 nejhorších softwarových zranitelností podle MITRE [4]

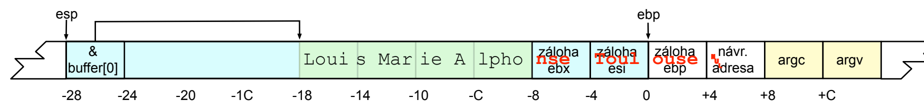
#	název
1	injekce (čehokoliv – kódu, příkazů, řetězců)
2	nesprávná autentizace
3	prozrazení citlivých dat
4	externí XML entity
5	nesprávné řízení přístupu
6	špatné nastavení zabezpečení
7	Cross-Site Scripting
8	nezabezpečená deserializace
9	používání komponent se známými zranitelnostmi
10	nedostatečné logování a monitorování

Tabulka 1.2: 10 nejhorších softwarových zranitelností podle OWASP [5]

základem pro naši navrženou metodu, kterou se pokusíme detekovat časté implementační chyby v kompilovaných jazycích. Tyto chyby si teď v krátkosti připomeneme.

1.1.1 Přetečení bufferu

Jedna z nejstarších chyb zneužívaná prvními viry a červy již v roce 1988 [2]. Využívá to, jakým způsobem procesor ukládá kontext běžícího procesu. Při zavolání funkce se adresa místa volání uloží na zásobník, aby procesor věděl, kam se má po skončení funkce vrátit. Stejný zásobník se ale bohužel používá i pro lokální proměnné funkce. Jelikož zásobník roste směrem k nižším adresám, zápis za hranici lokálního pole může způsobit přepsání návratové adresy, tak jak vidíme na obrázku, a následné neočekávané chování programu.



Obrázek 1.1: Příklad přetečení bufferu (převzato z [10])

V kompilovaných jazycích ke zranitelnosti může dojít buď používáním knihovních funkcí nekontrolujících hranice (jako např. v C `strcpy` nebo `gets`) nebo vymezením hranic nesprávně (`strncpy` s chybným argumentem délky). Nejtriviálnější příklad zranitelného kódu je ukázka 1.1

```
void vuln()
{
    char name[32];
    gets( name );
    printf( "Hello %s!", name );
}
```

Kód 1.1: Příklad přetečení bufferu

Následkem je, že pokud má uživatel pod kontrolou tento vstup, může v lepším případě způsobit pád programu skokem na neplatnou adresu, v horším kompletně ovládnout stroj spuštěním libovolného kódu.

Možností zneužití je několik. Pokud je buffer pro vstup dostatečně dlouhý a systém nemá aktivní ochranu proti spouštění dat na zásobníku, lze ve vstupu přímo specifikovat strojový kód a následně přepsat návratovou adresu tak, aby se na něj skočilo. Pokud se v paměti načtená knihovna s funkcí `system` nebo její alternativou, můžeme skočit do ní a na zásobník zapsat shellový příkaz. Typicky se snažíme vložit takový kód či příkaz, který spustí administrátorský shell, nebo v případě vzdáleného přístupu nějaký nástroj pro vzdálenou správu, např. telnet. Výsledkem je úplné ovládnutí počítače.

Přestože jde o velmi starou a velmi známou zranitelnost, na kterou existuje řada protiopatření (např. kanárci, ASLR), podle vyhledávače CVE Details [9] i v roce 2018 stále tvořila kolem 15 % všech nahlášených zranitelností a podle analýzy společnosti Sourcefire [6] je to dokonce „*zranitelnost čtvrtstoletí*“ (nejčastější zranitelnost posledních 25 let).

1.1.2 Injekce shellového příkazu

Zranitelnost vzniká tím, že se programátor pokouší v kódu spustit systémový příkaz, jehož část je uživatelským vstupem. Ve starších jazycích jako C, Fortran nebo Basic většinou existuje standardní knihovní funkce `system` nebo jinak pojmenovaný ekvivalent, který bere jediný argument, a to textový řetězec s shellovým příkazem, který se má spustit. Pokud tento argument vzniká spojením několika řetězců, z nichž jeden pochází ze vstupu do programu, dá se pomocí speciálních znaků sestavit jakýkoliv příkaz, který se spustí vedle toho původně zamýšleného.

Zranitelný kód je většinou ve stylu 1.2.

```
addr = getInputParam( "IP_addr" );
system( "ping -c 1 " + addr );
```

Kód 1.2: Příklad injekce příkazu

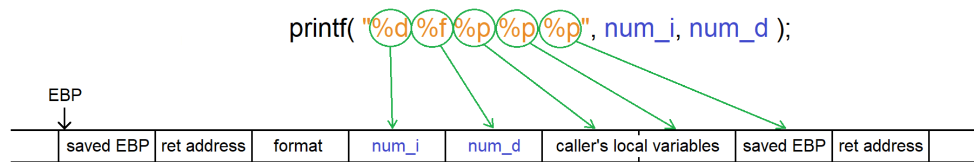
Při vstupu například `"; echo hacked"` vznikne `"ping -c 1; echo hacked"` a hned po ukončení neplatného příkazu ping se vypíše naše zpráva.

Stejně jako u přetečení bufferu, i zde můžeme dosáhnout vzdálené kontroly počítače tak, že například spustíme telnet démona bez hesla nebo pomocí utility wget stáhneme malware, který bude plnit příkazy získané z centrálního řídícího serveru. Výsledkem je opět úplné ovládnutí počítače.

Kromě toho, že v obou žebříčcích 1.1 a 1.2 je tato zranitelnost stále na prvních místech, v posledních letech nabývá na ještě větším významu díky rozmachu levných IoT zařízení a s tím souvisejícího rozšíření IoT botnetů. Podle bezpečnostních firem Kaspersky [11] a Netscout [12] se za poslední rok injekce příkazů, vedle defaultních hesel, stala jejich hlavní metodou šíření.

1.1.3 Injekce formátovacího řetězce

Kompilované jazyky, které neumějí variadické šablony, řeší proměnlivý počet argumentů při formátovaném výstupu za běhu, s pomocí informací ve formátovacím řetězci. Tento řetězec specifikuje, kolik argumentů a jakého typu se má ze zásobníku vyzvednout. Bohužel ani při kompilaci ani za běhu se nekontroluje, zda-li formátovací řetězec skutečné odpovídá předaným argumentům. Pokud tedy přidáme formátovací specifikátor, pro který neexistuje odpovídající argument, přečte se další hodnota na zásobníku následující za posledním argumentem. Použitím většího počtu specifikátorů typu `%x` nebo `%p` můžeme přečíst celý obsah zásobníku.



Obrázek 1.2: Formátovací řetězec a argumenty

Zranitelnost vzniká tehdy, když zprávu obsahující uživatelský vstup programátor použije jako formátovací řetězec místo dodatečného argumentu a příslušného formátovacího specifikátoru. Zranitelný kód může vypadat třeba jako 1.3

```
char message[64] = {'\0'};
strcat( message, user );
strcat( message, " connected to the server" );
printf( message );
```

Kód 1.3: Příklad injekce formátovacího řetězce

Pokud se uživatelský vstup nachází na zásobníku, s pomocí této chyby dokážeme číst a někdy i zapisovat na libovolné místo v paměti. Pro čtení stačí někam do vstupu zapsat adresu (většinou na konec, protože nuly v adrese by ukončily řetězec), ze které se má číst, potom pomocí specifikátorů `%p` `%x` nebo `%c` přeskočit všechny nezajímavé byty zásobníku, a na závěr použít `%s` pro přečtení obsahu. Pro zápis využijeme specifikátor `%n`, který do dané proměnné zapíše počet dosud vypsaných znaků. Opět použijeme stejný trik s přeskakováním na zásobníku a adresou na konci řetězce, jen na konci místo čtení přes `%s` zapíšeme hodnotu pomocí `%n`. Počet znaků snadno ovlivníme příslušným prodloužením formátovacího řetězce. Pokud bychom potřebovali zapsat velké číslo, a takový počet znaků by se do bufferu pro formátovací řetězec nevešel, můžeme číslo zapsat postupně po bytech s použitím `%hhn` (jedno-bytová verze `%n`).

V některých případech lze dokonce dosáhnout vzdáleného spuštění kódu přepsáním vhodných míst v paměti procesu. Buď můžeme klasicky přepsat návratovou adresu a okolní místa na zásobníku pro předání správných argumentů. Nebo jde přepsat položku v globální tabulce symbolů a změnit adresu funkce z externí knihovny, která se v programu volá někdy později. A nebo také lze přepsat adresu destruktorů volaných těsně před ukončením programu. Tyto metody jsou podrobně popsány v publikaci [13].

1.1.4 Shrnutí

Na základě popsaných problémů lze sestavit seznam kritických funkcí, do jejichž příslušných parametrů by se neměl dostat uživatelský vstup. Pro jazyk C by takový seznam mohl vypadat následovně:

- `system` (1. parametr)
- `strcpy` (2. parametr)
- `strcat` (2. parametr)
- `printf` (1. parametr) a její varianty (`sprintf`, `fprintf`, `vprintf`, ...)

Podle operačního systému ještě můžeme přidat

- Unix: `popen` (1. parametr)
- Windows: `_wsystem` (1. parametr)
- Windows: `_popen` (1. parametr)
- Windows: `_wpopen` (1. parametr)
- Windows: `wsprintf` (2. parametr)
- Windows: `wvsprintf` (2. parametr)
- Windows: `StringCbPrintf` (3. parametr) a její varianty (`StringCbPrintfEx`, `StringCbVPrintf`, `StringCchPrintf`, ...)

Tento seznam využijeme při testování našeho nástroje na vybraných příkladech zranitelných programů napsaných v C.

1.2 Statická vs. dynamická analýza

Existují dva hlavní přístupy ke hledání zranitelností – analýza statická a dynamická.

Dynamická přistupuje k problému prakticky. Program se spustí, předloží se mu určitý uživatelský vstup a následně se sleduje jeho chování. S pomocí metod ladění (anglicky „debugging“) můžeme sledovat využití paměti, volání systémových funkcí, vyhozené výjimky, dobu běhu atd. Výhoda dynamické analýzy je, že není vyžadován zdrojový kód ani nejsou kladeny požadavky na výsledný spustitelný soubor (používání base pointer registru, nepřítomnost obfuskací atd.), a také že její nálezy jsou potvrzené. Nevýhody jsou především nutnost připravit běhové prostředí a neschopnost pokrýt všechny případy vzhledem k obrovskému počtu možností vstupů. Metody dynamické analýzy jsou dlouho známé a hojně používané, např. v Unixových nástrojích Valgrind,

Mudflap a do jisté míry v každém debuggeru. Pro hledání zranitelností se často používá technika zvaná „fuzzing“ popsaná v práci [15]. Ta generuje velké množství různých vstupů, buď zcela náhodných nebo v určitém formátu, a když program spadne nebo vyprodukuje nesmyslný výstup, pak jsme narazili na chybu.

Statická analýza je teoretičtější a snaží se naopak spouštění programu vyhnout. Systematickým zkoumáním kódu (ať už zdrojového či strojového) se pokouší zjistit, zda-li existuje možnost, že nastane nějaký zakázaný stav. Výhodou je, že analýza může poskytnout důkaz o absenci určitého typu zranitelnosti. Nevýhodou naopak, že typicky produkuje velké množství falešných poplachů, které musí ručně prověřit člověk. Využití statické analýzy pro hledání zranitelností je stále ve fázi aktivního výzkumu, příkladem jsou práce [17], [18] nebo [20].

1.3 Možnosti statické analýzy

Zde stručně popíšeme některé typické metody statické analýzy.

1.3.1 Hledání vzorů (pattern matching)

Nejstarší a nejjednodušší způsob je vyhledávat v kódu konkrétní nebezpečné vzory, jako například

```
input = getInput(); ... system( input );
```

Hledat vzory se dá na třech úrovních porozumění kódu. Můžeme vyhledávat čistě textově, např. pomocí regulárních výrazů. Taková metoda je velice nepřesná, snadno ji zmate i komentář nebo bílé znaky. O něco sofistikovanější je lexikální analýza, která transformuje text na posloupnost tokenů. Ta se sice odstraní bílé znaky a rozpozná komentáře, ale stále nedokáže detekovat jakýkoliv komplikovanější případ obsahující např. makra nebo složené datové typy. Nejefektivnější je analyzovat abstraktní syntaktický strom (dále bude používána jen anglická zkratka AST) vzniklý kompletním naparsováním kódu. To má výhodu, že makra už jsou expandovaná a navíc vidíme vztahy mezi proměnnými. Celkově ale hledání vzorů není schopné postihnout zranitelnosti přesahující do více funkcí nebo ohraničené relevantními podmínkami. Navíc aby metoda fungovala, je potřeba mít a udržovat databázi známých nebezpečných vzorů, což snižuje její praktickou použitelnost. [16]

Mezi existující nástroje pokoušející se o hledání vzorů patří Flawfinder, RATS a ITS4.

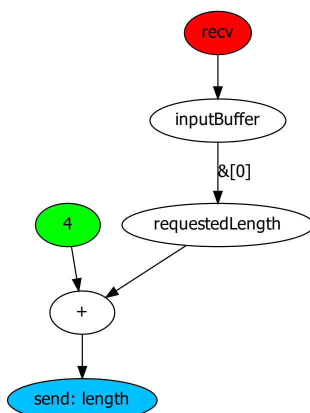
1.3.2 Analýza datových toků (data flow analysis)

Analýza datových toků je obecný pojem zahrnující zkoumání vztahů mezi proměnnými, vzájemných datových závislostí a způsobů propagace dat. Pro hledání zranitelností se může použít tak, že vyhledáme místa uživatelských vstupů a propagujeme informaci o tom, co všechno může být jimi ovlivněno.

Například datový tok vyvozený z kódu 1.4 značí zranitelnost, protože velikost odesílaných dat závisí na vstupu a nikoliv na skutečné délce pole.

```
recv( socketFd, inputBuffer, sizeof(inputBuffer) );
requestedLength = *(int*)&inputBuffer[0];
send( socketFd, outputBuffer, 4 + requestedLength );
```

Kód 1.4: Příklad problémového datového toku



Obrázek 1.3: Příklad datového toku pro délku pole

Varianta tohoto přístupu je v této práci implementována.

1.3.3 Analýza rozsahů hodnot (value range analysis)

Analýza rozsahů hodnot je metoda, která se snaží nalézt všechny možné hodnoty, kterých daná proměnná v daném místě programu může nabývat. Tato metoda původně vznikla pro predikci podmíněných skoků pro účely optimalizace. Avšak při hledání zranitelností je užitečná např. pro odhalení možného přetečení bufferu zadáním špatné velikosti. Pokud proměnná délky může obsahovat hodnotu vyšší než je skutečná velikost bufferu, potom máme potenciální zranitelnost. Dále je možno zjistit splnitelnost podmínky, která podmiňuje vznik zakázaného stavu způsobujícího zranitelnost, například nějakou kont-

rolu před nebezpečným použitím uživatelského vstupu, nebo kontrolu velikosti vstupu před kopírováním.

Algoritmus pro určení rozsahů hodnot byl popsán v práci [21]. V zásadě jde o zobecnění známého algoritmu propagace konstant. Kód se rekurzivně prochází od vstupní funkce. Každá deklarovaná proměnná dostane prázdnou množinu možných hodnot. Při přiřazení konstanty do proměnné se konstanta přidá do množiny možných hodnot. V případě přiřazení proměnné do jiné proměnné se množina té zdrojové zkopíruje do cílové. V případě přiřazení vnějšího vstupu do proměnné se proměnná označí jako libovolná hodnota. Pokud dvě proměnné vstupují do binární operace, výsledek operace se určí pro každý prvek kartézského součinu jejich množin. A při procházení větve podmíněné hodnotou proměnné se množina omezí na hodnoty splňující podmínku.

Příklad analýzy rozsahů můžeme vidět na kódu 1.5:

```
uint a = 10;          // a={10};
uint b = readInt();   // a={10}; b=ANY;
if (b < 3) {
    a = b * 2;        // a={0,2,4}; b={0,1,2};
}
uint c = a + 1;       // a={0,2,4,10}; b=ANY; c={1,3,5,11};
if (c < 5) {
    print("lower");   // a={0,2,4,10}; b=ANY; c={1,3};
} else {
    print("higher"); // a={0,2,4,10}; b=ANY; c={5,11};
}
```

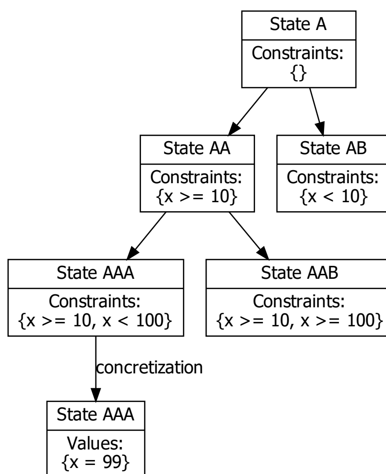
Kód 1.5: Příklad analýzy rozsahů

1.3.4 Symbolické provádění (symbolic execution)

Symbolické provádění je hybridní staticko-dynamická metoda podobná analýze rozsahů hodnot. Název metody je odvozen od toho, že program se od určitého bodu interpretuje speciálním způsobem, při kterém proměnné mají místo konkrétních hodnot jen hodnoty abstraktní (symbolické). Při každém rozvětvení programu se vždy procházejí obě větve, jenom si pro každou z nich poznamenáme omezení pro tu symbolickou hodnotu, které se týká podmínka větve. Když dojdeme do místa, které nás zajímá, vyřešíme soustavu nerovnic, abychom zjistili jestli je dané místo dosažitelné, případně pro jaký vstup. Jednoduchý příklad je na následujícím obrázku 1.4 (převzato z [23]).

Výsledkem je informace o tom, které vstupy způsobí průchod kterou cestou v programu. Metoda byla původně navržena pro generování testovacích vstupů pro ověření správné funkčnosti programu [22], ale dá se použít na detekci zranitelností typu obcházení autentizace/autorizace. Nejdříve v programu identifikujeme místo, kde je vybraná privilegovaná akce povolena (tam

```
x = readInt();
// Branch A
if (x >= 10) {
    // Branch AA
    if (x < 100) {
        // Branch AAA
        print("You win");
    } else {
        // Branch AAB
        print("You lose");
    }
} else {
    // Branch AB
    print("You lose");
}
```



Obrázek 1.4: Příklad symbolického provádění

se chce útočník dostat). Potom označíme všechna místa, kde do programu vchází nějaký vstup, a necháme symbolicky interpretovat program z míst vstupu do místa povolení akce. Pro každou nalezenou cestu od vstupu k povolení vznikne posloupnost podmínek, které musí být pro tuto cestu splněny. Vyřešením soustavy podmínek zjistíme, zdali existuje, vstup vedoucí do bodu povolení, případně jaký. V případě, že je nalezen jiný vstup než jaký byl programátorem zamýšlen, našli jsme zranitelnost.

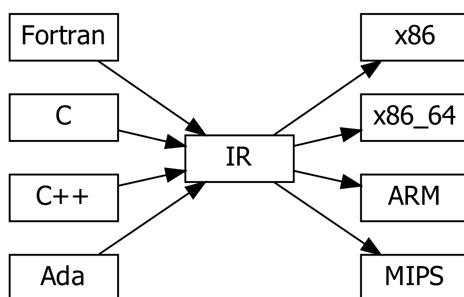
1.4 Zdrojový kód, strojový kód a meziformy

Statická analýza se typicky provádí na zdrojovém kódu. To má tu výhodu, že analýza má k dispozici všechny informace jako typy proměnných a signatury funkcí. Můžeme tak hledat chyby související s typy a přesností, např. ztrátu informace v důsledku oříznutí na kratší číslo nebo chybné statické přetypování reference. Nevýhoda je samozřejmě, že je vyžadován zdrojový kód, takže nelze skenovat spustitelné soubory či zkompilované knihovny třetích stran. Také je analyzátor potřeba napsat pro každý programovací jazyk zvlášť, a to obnáší jak parser a AST, tak analýzu samotnou, i když ta bude ve většině případů podobná a tudíž snadno přenositelná.

Alternativou je analyzovat spustitelné soubory. V mnoha případech zdrojové kódy nejsou k dispozici, například když do svého produktu linkujeme proprietární knihovnu třetí strany, nebo když analyzujeme firmware nějakého IoT zařízení. V takovém případě nám nezbývá než provádět analýzu nad strojovým kódem. Taková analýza má tu výhodu, že může detekovat i chyby vy-

generované kompilátorem i když je zdrojový kód v pořádku. Cena je ale ta, že se musíme obejít bez některých informací. Například pokud byly staticky přilinkovány knihovny standardního běhového prostředí, bez identifikátorů nepoznáme, že se volají standardní funkce daného programovacího jazyka. Existují metody, jak staticky linkované knihovny rozpoznávat, pokouší se o to třeba nástroj IDA FLIRT nebo BinDiff, ty ale nefungují vždy.

Existuje ale ještě třetí možnost – analýza nad nějakým druhem meziformy (anglicky „intermediate representation"). Meziformy se běžně používají v kompilátorech pro optimalizace a zjednodušení generování kódu pro různé procesory. Architektura typického kompilátoru pak vypadá takto.



Obrázek 1.5: Architektura kompilátorů

Pro každý vyšší programovací jazyk se napíše tzv. frontend, který naparsuje zdrojový kód a přeloží ho do mezikódu. Veškeré optimalizace a analýzy se provádí nad tímto mezikódem, takže je stačí napsat jednou a fungují jak pro všechny programovací jazyky, tak pro všechny architektury procesorů. Na závěr se výsledný transformovaný kód zvoleným backendem přeloží do cílového strojového jazyka. Tento způsob je velmi atraktivní pro provádění statické analýzy protože veškeré procedury jsou nezávislé na jazyku a navíc zde existuje určitý potenciál pro analýzu spustitelných souborů, protože dekompilace do mezikódu je mnohem jednodušší než dekompilace do zdrojového jazyka. Tento aspekt bude podrobněji rozebrán v poslední kapitole.

Kapitola 2

Volba meziformy pro analýzu

Jak bylo vysvětleno na konci předchozí kapitoly, nejvýhodnější je analyzovat nějaký druh meziformy, která je společná pro více programovacích jazyků a pro více procesorových architektur. Zde probereme základní vlastnosti meziforem používaných v kompilátorech a popíšeme konkrétní meziformy dominantních kompilátorů ze světa open-source. Informace o meziformách používaných v komerčních kompilátorech jako kompilátor z Visual Studia nebo kompilátor od Intelu nejsou k dispozici.

2.1 Tříadresový kód a SSA

Tříadresový kód je meziforma, která splňuje podmínku, že má maximálně 3 operandy – 2 vstupní a 1 výstupní. Většinou se zapisuje v podobě přiřazení unární či binární operace do proměnné, např. `x = a + b`. Složitější aritmetické výrazy se musí rozdělit do těchto elementárních tříadresových operací. Například výraz

$$x = a + b * -c * (d + 1)$$

bude vypadat takto

```
r1 = - c;
r2 = d + 1;
r1 = b * r1;
r1 = r1 * r2;
x = a + r1;
```

kde `r1` a `r2` jsou dočasné proměnné mající podobný účel jako registry.

Lze i zapisovat do paměti, a to pomocí ukazatelů. Například zápis hodnoty do elementu pole (`a[i] = x`) se provede následovně:

```
p = a + i
*p = x
```

Podmínky a cykly se provádějí pomocí příkazů podmíněných skoků, které opět mohou mít nejvýše 2 vstupní operandy, např.:

```
if x > y jump label1 else jump label2;
```

Cíl skoku je určen jménem tzv. basic bloku, což je úsek kódu, který obsahuje maximálně jeden cíl skoku, a to na začátku úseku, a maximálně jeden skok, a to na konci úseku. Jednotlivé basic bloky mají všechny vlastní název a skoky do nich jsou explicitní i tehdy, když cílový blok následuje hned za příkazem skoku. Zjednodušuje to optimalizace vyžadující přeskládání nebo odstranění bloků.

Striktnější meziformou je „Single Static Assignment“ forma (nemá vhodný český překlad, budeme používat jen zkratku SSA). Ta přidává omezení, že do každé dočasné proměnné lze přiřadit pouze jednou, a to v místě její deklarace. Předchozí příklad by v SSA podobě vypadal takto:

```
t1 = - c;
t2 = d + 1;
t3 = b * t1;
t4 = t3 * t2;
x = a + t4;
```

Tato vlastnost ale přináší jeden problém, a to, že v případě rozvětvení se na základě podmínky mohou přiřadit 2 různé hodnoty, např. `x = a < 5 ? a : c;`. Řešením jsou tzv. Φ-funkce, které obchází pravidlo jediného přiřazení formálním konstruktem, který z několika hodnot vybere tu správnou na základě toho, která větev byla provedena. Náš příklad bude tedy v SSA vypadat takto:

```
    if a < 5 jump less else jump grteq;
less:
    x1 = a;
    jump after;
grteq:
    x2 = c;
    jump after;
after:
    x = phi( x1, x2 );
```

Taková forma byla poprvé navržena v roce 1988 v [25] a dnes ji využívají všechny moderní kompilátory. Výhoda této formy je, že u každé proměnné jednoznačně víme, z čeho vznikla, což zjednodušuje některé optimalizace jako propagace kopií hodnot, eliminace nedosažitelného kódu nebo slučování společných podvýrazů [27].

Bohužel ne vždy lze kompletně vyloučit klasické proměnné s přiřazenou pamětí, do které lze opakovaně zapisovat, například u globálních proměnných nebo lokálních, na které potřebujeme vzít adresu. Takový případ se v SSA řeší

pomocí ukazatelů na explicitně alokovanou paměť, byť jde ve finále o statickou alokaci na zásobníku. Lokální proměnnou potom vyjadřujeme následujícím způsobem.

```
int* a = local_alloc( sizeof int );
```

A používáme ji přes explicitní operace čtení a zápisu.

```
*a = tmp1;
tmp2 = *a;
```

Vše dohromady lze vidět na následujícím o něco složitějším příkladu kódu, který sečte absolutní hodnoty čísel v poli.

```
int sum_abs( int* nums, int len )
{
    int sum = 0;
    for (int i = 0; i < len; i++) {
        int val;
        if (nums[i] < 0) {
            val = -nums[i];
        } else {
            val = nums[i];
        }
        sum += val;
    }
    return sum;
}
void main()
{
    int nums[] = {1,-2,3,-4};
    int sum = sum_abs( nums, 4 );
}
```

Jeho SSA podoba bude vypadat následovně:

```
int sum_abs( int* nums, int len )
{
    sum1 = 0;
    i1 = 0;
    jump cond;
 cond:
    sum = phi( sum1, sum2 );
    i = phi( i1, i2 );
    if i < len jump loop else jump exit;
 loop:
    tmp1 = nums[i];
    if tmp1 < 0 jump neg else jump pos;
 neg:
    val1 = -tmp1;
    jump endif;
```

```
pos:
   val2 = tmp1;
   jump endif;
endif:
   val = phi( val1, val2 );
   sum2 = sum + val;
   i2 = i + 1;
   jump cond;
exit:
   return sum;
}
void main()
{
   int* nums = local_alloc( 4 * sizeof int );
   ... init array ...
   sum = sum_abs( nums, 4 );
}
```

Nelze si nevšimnout podobnosti se strojovým kódem, kde dočasné proměnné připomínají registry a lokální alokované proměnné zase proměnné na zásobníku. Nicméně tato forma kompletně abstrahuje nad paměťovým modelem, například vůbec nepředpokládá zásobník, a tím pádem je přenositelná i na architektury ukládající stav jiným způsobem nebo používající více oddělených zásobníků pro proměnné a návratové adresy.

2.2 GENERIC a GIMPLE

Obě tyto formy jsou součástí kolekce kompilátorů GCC od verze 3. Vznikly na základě potřeby nějaké formy na vyšší úrovni než původní RTL (Register Transfer Language), ale společné pro všechny vstupní jazyky.

GENERIC je rozšířením původního AST pro Javu a jeho účelem je poskytnout na jazyce nezávislou stromovou reprezentaci celých funkcí. Základní typy uzlů jsou navrženy tak, aby byly použitelné pro téměř jakýkoliv procedurální jazyk, nicméně každý frontend (parser pro konkrétní jazyk) si může dodefinovat vlastní typy uzlů, pokud specifikuje, jak se mají přeložit do GIMPLu. Mezikód ve formě GENERIC je stále poměrně vysokoúrovňový, dokáže reprezentovat složitější výrazy v jednom přiřazení a podmíněné příkazy, ale například všechny typy cyklů jsou převedené do podoby návěští a skoků. [29]

GIMPLE je zjednodušením formy GENERIC. Splňuje podmínky SSA, tzn. každý příkaz má max. 3 operandy a každá proměnná má právě jedno přiřazení, kromě těch, které mají alokované místo v paměti. Avšak při přiřazení do složeného typu jako pole či struktura (`a.b[2] = 42`) se levá strana dále neštěpí, protože některé optimalizátory by ztratily potřebné informace. Podmínky jsou převedeny na podmíněné skoky a návěští a všechny typy cyklů jsou reprezentované jediným konstruktem nekonečné smyčky, kde podmínka ukončení je explicitně uvedena jako příkaz uvnitř těla cyklu. Volání funkcí

zůstává ve stejné podobě jako ve zdrojovém kódu a abstrahuje nad volacími konvencemi. Tato forma vzniká překladem z GENERICu a probíhá nad ní většina optimalizací a analýz. Po nich se ještě pro některé architektury GIMPLE přeloží do RTL, který už explicitně operuje s registry, ale neomezuje jejich počet. Přiřazení konkrétních registrů proběhne ve fázi překladu do finálního strojového kódu podle zvolené architektury CPU. [29]

Z programátorského hlediska jsou obě formy tvořeny uzly typu union propojené ukazateli. Typ uzlu je dán proměnnou typu enum, které se v dokumentací říká kód stromu (tree code). S uzly a stromy se pracuje přes čistě procedurální API a k položkám uzlu se přistupuje pomocí maker. Uzel funkce si uchovává typ návratové hodnoty, seznam parametrů[1] a odkaz na blok těla funkce. Blok (ne basic blok, ale blok ve významu jazyka C) se chová jako kontejner. Kromě toho, že drží svoje deklarace typů a proměnných, obsahuje spojový seznam příkazů. Bloky je možné vnořovat stejně jako v jazyce C. Příkaz může být přiřazení výrazu do proměnné, volání procedury, nebo složený příkaz jako podmínka nebo cyklus. Podmínky mají odkazy na boolovský výraz a blok kódu a cykly mají jen blok kódu, protože podmínka ukončení je uvnitř bloku. Výrazy jsou reprezentovány klasickým AST způsobem. Podle priority operátorů a závorek jsou rozčleněny do hierarchie binárních operací, kde každá operace je uzel binárního stromu, jehož potomkové jsou operandy. GENERIC i GIMPLE jsou stále (na rozdíl od RTL) typované, tzn. každá proměnná, výraz, funkce a parametr má svůj pojmenovaný typ. Ten je opět dán nějakým uzlem, uchovávajícím jméno a atributy typu, např. bitová délka. [28]

Tyto meziformy nelze používat mimo kompilátor GCC. Pro vytvoření vlastní analýzy je potřeba napsat modul do GCC a zaregistrovat patřičné funkce.

2.3 LLVM

LLVM byla původně zkratka znamenající „Low Level Virtual Machine“ a šlo o projekt univerzitního původu pro zkoumání průběžného překladu dynamických jazyků. Poté, co byl ale převzat společností Apple, se projekt přetvořil v univerzální platformu pro kompilaci a optimalizaci, nahrazující starší a méně přizpůsobivou infrastrukturu GCC. Využívá ho například kompilátor clang, debugger LLDB nebo virtuální stroj pro symbolické provádění KLEE.

LLVM meziforma, stejně jako GENERIC a GIMPLE, jde reprezentovat dvěma způsoby – textovým kódem s určitou syntaxí a datovými strukturami v programu. Meziforma jako taková a její textová podoba je detailně definovaná v oficiální dokumentaci [31]. Důležité je, že splňuje podmínky SSA,

[1] Termíny „parametr“ a „argument“ se často pletou nebo používají nesprávně. V této práci budou použity ve významu tak, jak jej definuje prvotní návrh jazyka C++ [34], tzn. parametr je proměnná deklarovaná uvnitř deklarace funkce a argument je výraz použitý při volání funkce. Jinými slovy, argument je hodnota přiřazená do parametru.

je silně typovaná a abstrahuje nad konstrukty závislými na platformě jako zásobník nebo příznakové registry. Lokální proměnné, které potřebují mít přiřazenou paměť, se deklarují instrukcí alokace a pracuje se s nimi přes explicitní instrukce read a write. Tzv. Control Flow Graph (CFG) je v LLVM explicitní, všechny basic bloky jsou označeny názvem a všechny přechody mezi nimi jsou vyjádřeny instrukcí skoku, i když se skáče jen na následující instrukci. Volání funkce probíhá vždy v jediné instrukci, nezávisle na počtu parametrů, pomáhá to abstrahovat nad volacími konvencemi. Typový systém je nezávislý na zdrojovém jazyku a vyžaduje striktní dodržování typů včetně znaménkovosti. Např. operand podmíněného skoku musí mít typ boolean a znaménkové celé číslo lze sečíst pouze s celým číslem, pokud máme hodnotu jiného typu, musíme ji explicitně převést instrukcí konverze.

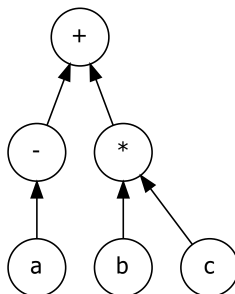
LLVM projekt obsahuje programátorský framework pro práci s mezikódem v objektové podobě. Ten modeluje mezikód jako AST, kde každý uzel je objekt s ukazateli na rodiče a potomky. Z každého souboru s mezikódem vznikne objekt `Module`, který obsahuje deklarace/definice konstant, globálních proměnných a funkcí. Funkce jsou reprezentovány objekty typu `Function`. Ty drží seznam svých parametrů, lokálních proměnných a graf basic bloků v podobě objektů typu `BasicBlock`. Každý `BasicBlock` má potom seznam instrukcí LLVM kódu modelovaných třídou `Instruction`. Podrobnou dokumentaci včetně obrázků hierarchie dědičnosti nalezneme na oficiálních stránkách [32]. Za zmínku ještě stojí třída `Value`, která je nadtřídou pro cokoliv, co lze použít jako operand instrukce. Zvláštností LLVM je, že každá instrukce je zároveň hodnotou (třída `Instruction` dědí of `Value`). To vypadá na první pohled zvláštně a pro začátečníka to může být matoucí, ale odpovídá to způsobu jakým je konstruován abstraktní syntaktický strom. Například LLVM kód 2.1 se v LLVM frameworku zkonstruuje kódem 2.2 a v paměti bude vypadat jako obrázek 2.1. Kód konstrukce tedy odpovídá mezikódu jedna ku jedné a při konstrukci instrukce můžeme předchozí instrukci předat jako argument té nové.

```
%neg = sub nsw i32 0, %a
%mul = mul nsw i32 %neg, %b
%add = add nsw i32 %mul, %c
```

Kód 2.1: LLVM kód výrazu

```
BinaryOperator* neg = BinaryOperator::Create(Instruction::
   Sub, const_int32_0, a, "neg", label_14);
BinaryOperator* mul = BinaryOperator::Create(Instruction::
   Mul, neg, b, "mul", label_14);
BinaryOperator* add = BinaryOperator::Create(Instruction::
   Add, mul, c, "add", label_14);
```

Kód 2.2: Konstrukce výrazu



Obrázek 2.1: Podoba výrazu v paměti

Každá instance `Value` má ještě odkaz na svůj typ v podobě třídy `Type`, která má podtřídy pro každý číselný a logický typ a pro složené typy pole a struktura.

Framework má podobu statických knihoven, které se dají přilinkovat k vlastnímu programu. Pro vytvoření analyzátoru tedy není potřeba zasahovat do kompilátoru a lze vytvořit samostatný program analyzující pouze LLVM mezikód.

2.4 Rozhodnutí

Existuje ještě několik dalších veřejně popsaných meziforem, ale ty nejsou příliš významné, protože jsou používány pouze pro jediný konkrétní jazyk a tudíž jejich univerzálnost není tak vysoká.

Při porovnání formy GIMPLE a LLVM vychází LLVM lépe v každém ohledu. LLVM je modernější, navržené pro současné potřeby a nezatížené historickými požadavky. LLVM framework je psaný v moderním C++ objektovým stylem (na rozdíl od čistě procedurálního C API v GCC) a je také mnohem lépe dokumentovaný. Jeho API se přesně hodí pro účely našeho algoritmu popsaného níže. S pomocí jeho staticky linkovaných knihoven lze snadno vytvořit samostatný nástroj pro analýzu nad LLVM kódem. Navíc, existující snahy o dekompilaci ze strojového kódu do LLVM teoreticky v budoucnu umožní analýzu spustitelných souborů.

Z těchto důvodů bylo LLVM vybráno jako meziforma pro naši implementaci.

Kapitola 3

Analýza datových toků

Zde se pokusíme pomocí analýzy datových toků detekovat zranitelnosti typu přetečení bufferu, injekce příkazu a injekce formátovacího řetězce. Společným postupem bude nalézt všechna volání kritických funkcí a pro každé z nich ověřit, jestli část dat v argumentu nepochází z uživatelského vstupu.

3.1 Metoda zpětného hledání původu dat

Pro hledání možných zdrojů dat použijeme metodu popsanou v následujícím textu. Při vysvětlování principu metody budeme pro názornost používat kód v jazyce C, protože odpovídající kód LLVM by byl příliš dlouhý a nepřehledný. Jak tyto principy převézt z C do LLVM bude popsáno v kapitole Implementace.

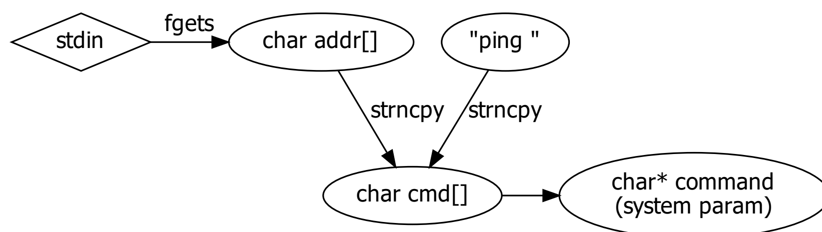
Představme si tento jednoduchý případ práce s řetězci:

```
void foo()
{
    char addr[16], cmd[32];
    fgets( addr, sizeof(addr), stdin );
    strncpy( cmd, "ping ", sizeof(cmd) );
    strncpy( cmd + 5, addr, sizeof(cmd) - 5 );
    system( cmd );
}
```

Kód 3.1: Příklad datových toků při práci s řetězci

Datové toky se nechají vizualizovat způsobem 3.1 Naším úkolem je nalézt všechny možné zdroje dat do argumentu volání funkce `system`. Argument takovéto řetězcově orientované funkce je adresa začátku lokálního pole. Do toho se ale zapisují data z více míst.

Hledání zdrojů má tedy 2 fáze:



Obrázek 3.1: Příklad datových toků v kódu 3.1

1. nalezení pole, jehož adresa se posílá do funkce
2. vyhledání všech možných vstupů do tohoto pole

3.1.1 Fáze 1

V tomto ukázkovém případě je první fáze triviální, protože argument je přímo adresou lokálního pole, bez jakékoliv mezihodnoty. Nalezení pole má tedy jediný krok – zjištění na co ukazuje argument volání [2]. Obecně ale mohou nastat tyto případy:

- adresa do pole je posunuta o nějaký offset
- adresa do pole je uložená v proměnné, do které se zapíše předtím
- ukazatel je parametrem funkce

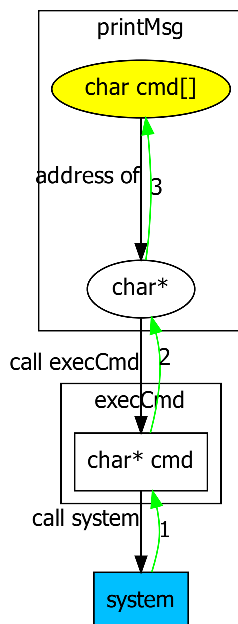
Když narazíme na binární operaci přičtení/odečtení offsetu od adresy, pouze vezmeme ten operand, který je typu ukazatel, a pokračujeme hledání s ním. V případě proměnné najdeme místa, kde se do ní zapisuje a pokračujeme se zdrojovými operandy. A pokud dojdeme k parametru, musíme najít všechna místa, odkud se funkce volá, a rekurzivně opakujeme postup, který začal argumentem volání funkce `system`.

Tato fáze lze chápat jako zpětné prohledávání grafu datových závislostí. Pokud se pole bude nacházet v jiné funkci, třeba jako v kódu 3.2, potom zpětný průchod datovými závislostmi bude vypadat jako na obrázku 3.2 (modrý uzel je počátek, žlutý je cíl a zelené šipky ukazují průchod):

[2] Jazyk C zavádí zmatek v tom, že proměnná typu pole se implicitně přetypuje na ukazatel na pole tam, kde se to hodí. Operace získání adresy lokální proměnné (tak jak bychom to udělali např. u struktury pomocí '&') je tedy v kódu skryta, ale přesto tam je.

```
void execCmd( char* cmd )
{
    system( cmd );
}
void printMsg( char* msg )
{
    char cmd[64];
    strncpy( cmd, "echo ", sizeof(cmd) );
    strncpy( cmd + 5, msg, sizeof(cmd) - 5 );
    execCmd( cmd );
}
```

Kód 3.2: Příklad mezifunkčních datových závislosti



Obrázek 3.2: Ukázka zpětného průchodu (fáze 1)

3.1.2 Fáze 2

Jakmile získáme cílové pole, nalezneme všechna místa, odkud se do něj zapisuje, a pro každé z nich rekurzivně použijeme fázi 1.

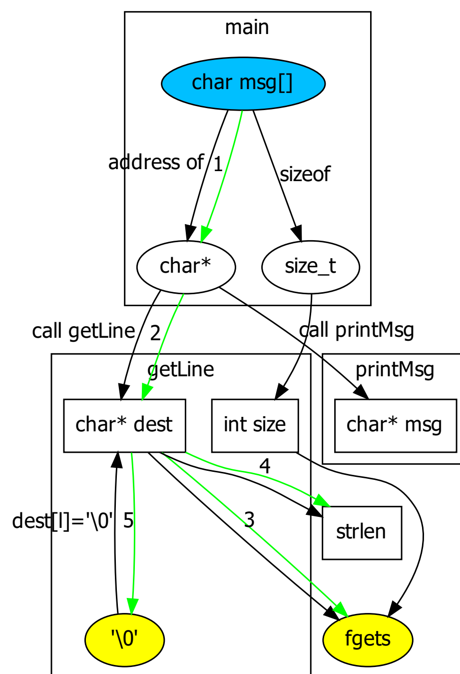
Místa zápisu ale opět nemusí být uvnitř stejné funkce, kde se nachází cílová proměnná, viz příklad 3.3.

Potom, pro každou neznámou funkci, které se předává adresa pole, musíme

```
void getLine( char* dest, int size )
{
    fgets( dest, size, stdin );
    int len = strlen( dest ) - 1;
    dest[ len ] = '\0';
}
void main()
{
    char msg[64];
    getLine( msg, sizeof(msg) );
    printMsg( msg );
}
```

Kód 3.3: Příklad mezifunkčních datových závislosti

projít tělo funkce a najít místa použití příslušného parametru (obrázek 3.3).



Obrázek 3.3: Ukázka hledání míst zápisu (fáze 2)

3.1.3 Ukončení rekurze

Rekurzivní prohledávání končí buď pokud narazíme na konstantu, a nebo pokud nalezneme funkci načítající data z uživatelského vstupu, jiná možnost ne-

existuje. Oba tyto případy přidáme do výsledného seznamu možných vstupů, s příslušným příznakem konstanta/vstup a vyřízení necháme na volajícím.

3.1.4 Výsledná podoba

Obě fáze dohromady se dají popsat pseudokódem 1.

Algorithm 1 Hledání zdrojů

```
function FindSources(arg)
    sources ← empty list
    Phase1(arg, sources)
    return sources
end function
function Phase1(arg, sources)
    current ← arg
    while true do
        if current is binary operator plus then
            current ← first operand of current
        else if current is parameter then
            for each call in calls to this function do
                Phase1(corresponding argument of call, sources)
            end for
        else if current is constant then
            append { current, CONSTANT } to sources
        else if current is pointer to array then
            Phase2(target of current, sources)
        else
            error “unexpected value type”
        end if
    end while
end function
function Phase2(array, sources)
    for each use in uses of array do
        if use is argument for a call then
            Phase2(use, sources)
        else if use is input from another variable then
            Phase1(use, sources)
        else if use is input from user then
            append { use, USERINPUT } to sources
        else
            error “unexpected use of array”
        end if
    end for
end function
```

Běh celého algoritmu demonstrujeme na příkladu s několika funkcemi (3.4).

```
void execCmd( char* cmd )
{
    system( cmd );
}
void printMsg( char* msg )
{
    char cmd[64];
    strncpy( cmd, "echo ", sizeof(cmd) );
    strncpy( cmd + 5, msg, sizeof(cmd) - 5 );
    execCmd( cmd );
}
void getLine( char* dest, int size )
{
    fgets( dest, size, stdin );
    int len = strlen( dest ) - 1;
    dest[ len ] = '\0';
}
void main()
{
    char msg[64];
    getLine( msg, sizeof(msg) );
    printMsg( msg );
}
```

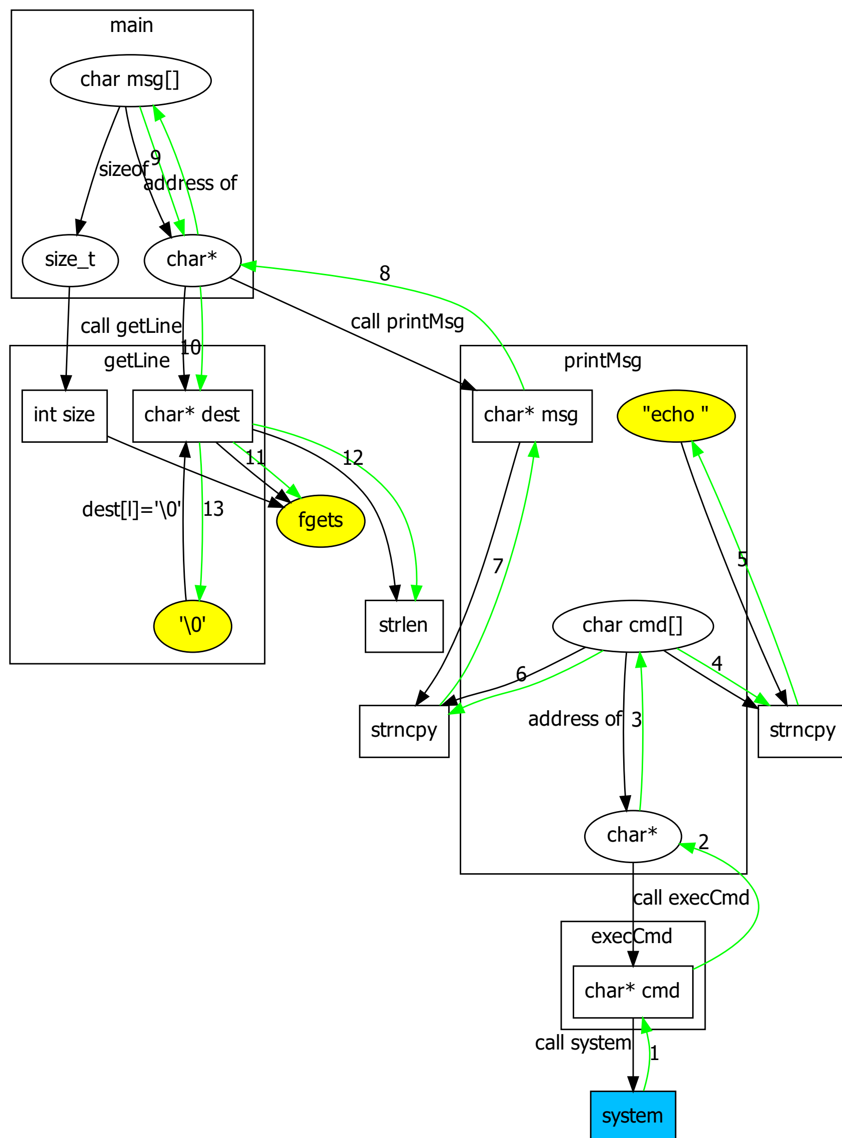
Kód 3.4: Příklad běhu zpětného hledání zdrojů

Algoritmus provede tyto kroky:

1. (fáze 1) zjistí, že argument volání `cmd` je parametr funkce,
2. (fáze 1) nalezne volání aktuální funkce uvnitř funkce `printMsg`,
3. (fáze 1) zjistí, že argument volání je lokální pole `cmd`,
4. (fáze 2) zjistí, že první použití `cmd` je kopírování přes `strncpy`,
5. (fáze 1) zjistí, že zdroj do `strncpy` je konstanta,
6. (fáze 2) zjistí, že druhé použití `cmd` je kopírování přes `strncpy`,
7. (fáze 1) zjistí, že zdroj do `strncpy` je parametr `msg`,
8. (fáze 1) nalezne volání aktuální funkce uvnitř funkce `main`,
9. (fáze 1) zjistí, že argument volání je lokální pole `msg`,
10. (fáze 2) zjistí, že první použití `msg` je první argument volání `getLine`,

11. (fáze 2) zjistí, že první použití `dest` je volání `fgets` – uživatelský vstup,
12. (fáze 2) zjistí, že druhé použití `dest` je volání `strlen` – jen čtení,
13. (fáze 2) zjistí, že třetí použití `dest` je ruční zápis `'\0'` – konstanta,
14. (fáze 2) zjistí, že druhé použití `msg` je volání `printMsg` – žádná akce (odtud jsme přišli).

Vizualizace metody je na obrázku 3.4.



Obrázek 3.4: Příklad běhu zpětného hledání zdrojů

Tato metoda je alternativou k podobné metodě popsané v [21], ale ta prochází datové toky opačným směrem – od zdroje dat k jejich použití. Takový způsob vede k většímu větvení a delšímu prohledávání, protože analyzuje dopady všech uživatelských vstupů, i těch, které se nikdy nedostanou k žádné kritické funkci.

3.2 Detekce injekce příkazu či formátovacího řetězce

Nejjednodušší aplikací výše popsaného algoritmu je detekce injekce příkazů a formátovacího řetězce. Obě zranitelnosti nastávají tehdy, pokud se uživatelský vstup může dostat do konkrétního parametru funkce. Pro jazyk C je to první parametr funkce `system` a první parametr `printf` nebo `scanf` a jejich variant (`vprintf`, `sprintf`, `fprintf`, ...).

Stačí nám tedy definovat seznam takových funkcí a k nim příslušné parametry podle zvoleného jazyka a operačního systému. Potom už jen projdeme celý program a nalezneme všechna volání kterékoliv z nich. Pro každé takové volání spustíme hledání zdrojů, a pokud je některý z nich vstupem od uživatele, zahlásíme upozornění.

3.3 Detekce přetečení bufferu

Přetečení bufferu je o něco složitější. Navíc, touto metodou jsme schopni pokrýt jen některé způsoby, jak k němu může dojít, konkrétně tyto dvě:

1. Uživatelský vstup jde do řetězcové operace, která nehlídá hranice pole.
2. Je použita operace hlídající hranice, ale délka je zadána špatně.

První případ stále spadá do kategorie případů z předchozí podkapitoly. Stačí najít volání problémové funkce a prohledat zdroje dat do příslušného parametru, žádný z nich nesmí pocházet od uživatele.

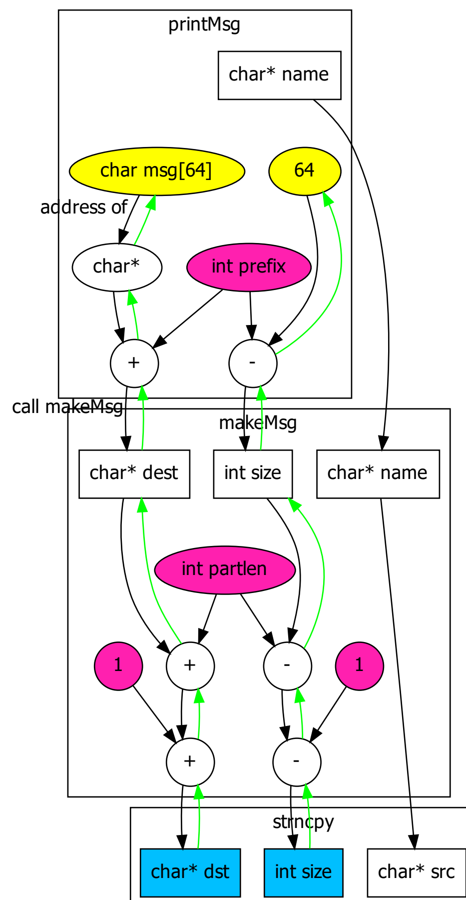
Druhý případ je komplikovanější. Můžeme použít první fázi algoritmu na nalezení cílového pole, do kterého se má zapisovat, a ověřit, zda délka zadaná programátorem souhlasí se skutečnou délkou pole. V příkladě 3.4 ale můžeme vidět situaci, pro kterou to nebude fungovat. Konkrétně řádku:

```
strncpy( cmd + 5, msg, sizeof(cmd) - 5 );
```

Do cílového pole se nezapisuje od začátku, ale od určitého indexu dál. Ukazatel na začátek je tím pádem posunutý a příslušný argument délky je o stejnou hodnotu zmenšený. Po cestě při zpětném hledání tedy musíme tyto posuny zaznamenávat a následně porovnat. Tuto myšlenku znázorňuje kód 3.5 a obrázek 3.3 (fialově jsou zvýrazněny posuny).

```
#define SIZE 64
void printMsg( char* name )
{
    char msg[SIZE];
    int prefix = sprintf( msg, "[%d] ", counter++ );
    makeMsg( msg + prefix, SIZE - prefix, name );
    printf( "%s\n", msg );
}
void makeMsg( char* name, char* dest, int size )
{
    char* part = "hello";
    int partlen = strlen(part);
    strcpy( dest, "hello" );
    dest[partlen] = ' ';
    strncpy( dest+partlen+1, name, size-partlen-1 );
}
```

Kód 3.5: Příklad kódu s posunutým ukazatelem



Obrázek 3.5: Příklad kontroly správné velikosti pole

Správně napsaný kód tedy musí splňovat 2 podmínky.

1. Počáteční hodnota délky pole musí souhlasit se skutečnou délkou deklarovaného pole.
2. Každá hodnota, která se přičte k adrese pole, se musí odečíst od délky.

Pokud některý sčítanec na jedné straně chybí, nebo jsou počáteční hodnoty různé, zahlásíme varování. Pokud zjistíme, že původní velikost pole a pole samotné pochází z úplně jiného místa, zahlásíme varování také. Sice se nemusí nutně jednat o zranitelnost, ale je to programovací praktika, která snadno v budoucnu na zranitelnost vést může.

Kapitola 4

Omezení falešných poplachů

Falešný poplach, v anglické literatuře označovaný jako „false positive“ nebo jen zkratkou FP, je případ, kdy nějaký detektor oznámí problém (v tomto případě zranitelnost), ale ve skutečnosti se o problém nejedná. Obecně nastává, když je detektor příliš citlivý a aktivuje ho i malý náznak problému.

Statická analýza je obecně náchylná k častým falešným poplachům. Při detekci zranitelností nastávají zejména tehdy, když programátor nežádoucí případ vyloučí způsobem, který hledač zranitelností nedokáže rozpoznat. Takových způsobů je celá řada a jejich řešení je většinou dost netriviální. Na některé z nich se teď podíváme.

4.1 Pořadí provádění

Nejjednodušším falešných poplachem je špatné pořadí instrukcí. Společný buffer sice může být použit jak pro načítání vstupu, tak pro systémový příkaz, ale načtení proběhne až po provedení příkazu. Sice na první pohled netypický případ, ale může se stát, pokud se někdo snaží uspořit paměť ve vestavném zařízení nebo není příliš zkušený v programování.

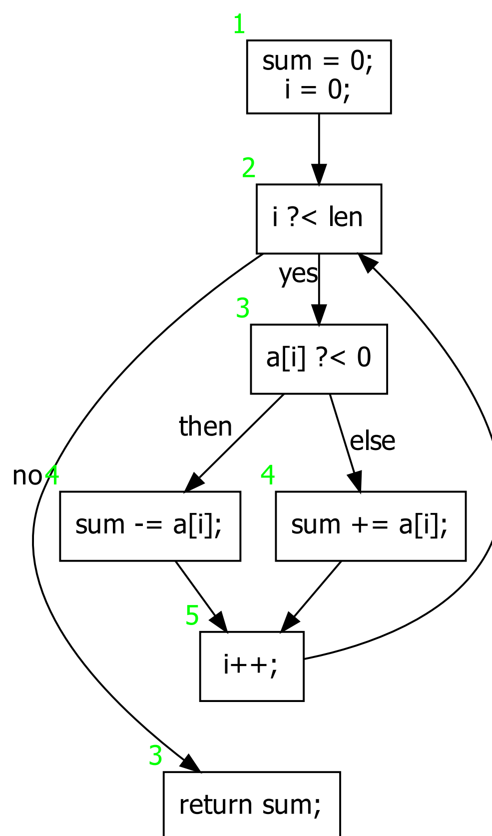
Pořadí je potřeba zkontrolovat v každém místě, kde se kopírují data do pole před jeho použitím. Při hledání zdrojů je tedy potřeba zaznamenávat taková místa a pořadí ověřit pro každé z nich. V příkladě 3.4 jsou taková místa dvě, první je kopírování do `cmd` pomocí `strncpy` s následným voláním `execCmd`, druhé je načítání vstupu do `msg` pomocí `getLine` s následným výstupem přes `printMsg`.

Jakmile máme dvojici instrukcí způsobující zápis do pole a jeho použití, např. `call getLine` a `call printMsg`, jejich vzájemné pořadí určíme seřazením basic bloků a porovnáním čísel bloků, ve kterých se instrukce nacházejí. Seřazení se provede průchodem do hloubky ze vstupního bloku funkce a přiřazením aktuální hloubky každému navštívenému uzlu.

Demonstrujeme na následujícím příkladu:

```
int sum_abs( int* a, int len )
{
    int sum = 0;
    for (int i = 0; i < len; i++) {
        if (a[i] < 0) {
            sum -= a[i];
        } else {
            sum += a[i];
        }
    }
    return sum;
}
```

Basic bloky tohoto kódu a jejich pořadí bude vypadat takto:



Obrázek 4.1: Basic bloky funkce sum_abs

Jak je vidět, některé bloky mají stejné číslo, protože může následovat kterýkoliv z nich podle výsledku podmínky. Problém statického vyhodnocení podmínek je vědní oblast sama o sobě, částečně se ho dotkneme v následující podkapitole, ale není v ambicích této práce takový problém vyřešit. Proto bloky v sousedních větvích podmíněného příkazu budeme považovat za rovnocenné a pokud se načtení vstupu a jeho použití odehrají v takovýchto rovnocenných blocích, budeme tento případ ignorovat.

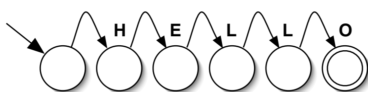
Ještě může nastat případ, že obě instrukce leží ve stejném bloku. Potom místo sestavování pořadí bloků jednoduše určíme, která z instrukcí je v bloku dřív.

4.2 Sanitace vstupních dat

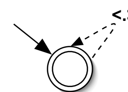
Typickým způsobem, jak zbavit uživatelský vstup škodlivého vlivu, je modifikace vstupu a neutralizace speciálních znaků, buď odstraněním nebo tzv. vyescapováním.

Můžeme zkoušet hledat neutralizující funkce typické pro daný jazyk, jako je například `mysqli_escape_string`, standardizovaná v PHP. Problém je, že nízkoúrovňové jazyky většinou podobné funkce nemají a programátor si je musí napsat sám. Problém se tedy převádí na problém detekce vlastnoručně napsaných neutralizujících funkcí.

Zajímavý nápad dostali autoři nástroje Saner [33]. Pro každou cestu ze zdroje dat do jejich použití se sestrojí konečný automat popisující výsledný řetězec po provedení všech řetězcových operací. Textová konstanta je modelována jednoduchým lineárním automatem (obrázek 4.2) a vstup je modelován jako posloupnost libovolných znaků (obrázek 4.3). Obrázky jsou převzaty z [33].



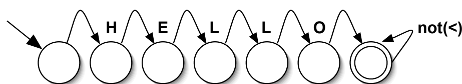
Obrázek 4.2: Automat pro konstantu



Obrázek 4.3: Automat pro vstup

Každá řetězcová operace se potom reprezentuje odpovídající operací nad automaty. Třeba spojení řetězců znamená spojit automaty a vymazání znaků se provede vymazáním stavu nebo odebráním prvku z množiny znaků u hrany přechodu. Například následující modifikace řetězce by se automatem reprezentovala jako automat na obrázku 4.4 (převzato z [33]).

```
input = readInput();
sanitized = str_replace( '<', '', input );
message = "Hello " + sanitized;
```



Obrázek 4.4: Automat po nahrazení znaků a spojení

Vzhledem v náročnosti implementace popsané metody náš nástroj podporuje pouze detekci předem specifikovaných neutralizačních funkcí. Je na analytikovi používajícím nástroj, aby identifikoval funkce neutralizující škodlivý vstup a ověřil jejich správnost. Program potom vezme seznam těchto funkcí a pokud nalezne volání některé z nich na cestě od vstupu dat k jejich použití, data jsou dále označena jako bezpečná a dosažení kritické funkce nevyvolá varování.

4.3 Podmínky vylučující provedení

Doposud popsané metody fungují skvěle na lineárním kódu – kódu, který se nikde nevětví a všechny příkazy po cestě se provedou vždy. Ale častým způsobem ošetření škodlivého vstupu je nějaká kontrola jeho obsahu a přerušení akce s chybou, pokud není vstup korektní, například:

```
input = readInput();
if (!isValid( input ))
    error( "invalid input" );
execCmd( input );
```

Pro detekci validace by šel použít podobný postup jako u detekce escapování. Museli bychom pomocí automatů modelovat všechny kontroly řetězce mezi načtením dat a jejich použitím, a sestrojit takový automat, který přijímá stejný jazyk jako je ten, který projde všemi podmínkami až do místa použití.

Programátor ale může části kódu obsahující načítání a použití obalit dalšími podmínkami, které se vstupem ani nesouvisí, třeba:

```
char buffer[64];
if (param1 == 1) {
    readInput( buffer );
} else {
    strcpy( buffer, "constant cmd" );
}
if (param2 == 2) {
    execCmd( buffer );
} else {
    print( buffer );
}
```

Takový případ bychom mohli opět řešit zpětným prohledáváním zdrojů dat a následným vyhodnocováním splnitelnosti podmínek. Tím ale dostáváme

problém typu „slepice nebo vejce“, protože nalezené vstupní hodnoty mohou být opět něčím podmíněny.

Jediné systematické řešení je symbolické provádění – spustit program od určitého bodu a po celou dobu si uchovávat kontext, který obsahuje omezující podmínky pro hodnoty všech proměnných. V momentě, kdy objevíme ono místo uživatelského vstupu a místo použití dat v kritické funkci, vyhodnotíme splnitelnost konjunkce podmínek pro obě místa. Pokud jsou podmínky nesplnitelné a nebo jsou v konfliktu s tím co je potřeba k úspěšné exploitaci, pak program není zranitelný.

Tato metoda není v práci implementována, protože by vyžadovala implementaci kompletního symbolického virtuálního stroje LLVM, včetně emulace systémových volání a řešičku SAT problému pro vyhodnocení splnitelnosti.

Virtuální stroj pro symbolickou exekuci LLVM již existuje – projekt KLEE [35]. Bylo zvažováno jeho napojení do tohoto nástroje a kombinace obou přístupů. Jeho současná architektura ale bohužel propojení neumožňuje. KLEE je navržen jako konzolový nástroj spíše než knihovna, a jeho účelem je generovat vstupy procházející všechny větve analyzovaného programu. Pro jeho využití bychom potřebovali umět spustit analýzu z libovolného místa v programu a dodefinovat podmínky, které v určitých místech musí platit. To ale v současné podobě nelze.

Kapitola 5

Implementace

Pro ověření konceptu byla metoda zpětného hledání původu dat implementována v jazyce C++ nad meziformou LLVM s použitím LLVM frameworku ve verzi 3.8. V této kapitole probereme některé konkrétní vlastnosti LLVM, které jsou užitečné pro analýzu datových toků, a také některé implementační problémy, které bylo potřeba vyřešit.

5.1 Využití LLVM API pro hledání původu dat

Jak bylo popsáno v podkapitole o meziformách kompilátorů, LLVM interně uchovává mezikód jako objektový AST. Každá instrukce má odkazy na svoji funkci, na svůj basic blok a také na operandy.

5.1.1 Fáze 1

První fázi hledání zdrojů (hledání bufferu, na který ukazuje ukazatel) realizujeme rekurzivním průchodem řetězce závislostí. Uvnitř funkce se v datových závislostech pohybujeme nahoru přechodem na operand instrukce metodou `Instruction::getOperand(int index)`, což můžeme ukázat na následujícím příkladu.

```
%command = alloca [128 x i8], align 16
...
%4 = getelementptr [128 x i8], [128 x i8]* %command, i32 0, i32 0
%5 = getelementptr i8, i8* %4, i64 5
...
%7 = call i8* @strncpy(i8* %5, i8* %6, i64 128)
```

Pokud chceme například dotrasovat cíl kopírování dat pomocí `strncpy`, začneme s hodnotou `%5` (první argument volání funkce), která je typu `GetElementPtrInst`. Prvním použitím `getOperand(0)` se dostaneme k `%4`, což je další `GetElementPtrInst`, a druhým použitím `getOperand(0)` dojdeme na `%command` – deklarace lokálního pole v podobě instrukce `AllocaInst`.

Pokud dojdeme k parametru, je potřeba nalézt všechna volání aktuální funkce. LLVM objekty dokáží kromě předků najít i svoje potomky. Metodou `Function::getUsers()` dostaneme seznam instrukcí, které funkci nějakým způsobem používají. Proiterujeme přes seznam a vybereme ty instrukce, které jsou typu `CallInst`. Z instrukce volání získáme ty argumenty, které přísluší parametru, ke kterému jsme došli uvnitř předchozí funkce, a pokračujeme opět posunem přes operandy.

Bohužel v neoptimalizovaném kódu jsou všechny lokální proměnné a parametry ze zdrojového jazyka modelovány paměťovým místem alokovaným instrukcí `AllocaInst` a všechna jejich použití obnáší instrukce load a store, místo toho, aby byly modelovány podle pravidel SSA. Je to nutné kvůli debuggerům, aby mohly číst obsahy konkrétních proměnných. Tím ztrácíme některé výhody SSA formy a musíme si poradit i s těmito proměnnými. Když tedy například při hledání bufferu dojdeme k lokální proměnné typu ukazatel na buffer, musíme najít nejbližší instrukci zápisu a potom pokračovat s jejím operandem.

5.1.2 Fáze 2

V druhé fázi hledáme všechna místa zápisu do nalezeného bufferu. K tomu opět využijeme seznam vrácený z `Instruction::getUsers()` a metodu dále rekurzivně voláme na každého potomka stromu datových závislostí. Pokud narazíme na zápis do lokální proměnné typu ukazatel na pole, musíme najít všechna její čtení. Pokud dojdeme k volání funkce, zjistíme cíl volání metodou `CallInst::getCalledFunction()` a pokračujeme uvnitř funkce příslušným parametrem. Fázi stručně vyjadřuje následující příklad.

```
define i32 @main(i32 %argc, i8** %argv) #0 {
  ...
  %input = alloca [128 x i8], align 16
  ...
  %5 = getelementptr [128 x i8], [128 x i8]* %input, i32 0, i32 0
  %6 = call i32 @getInput(i8* %5, i32 128)
  ...
}
define i32 @getInput(i8* %dest, i32 %maxSize) #0 {
  ...
  %2 = alloca i8*, align 8
  ...
  store i8* %dest, i8** %2, align 8
  ...
  %12 = load i8*, i8** %2, align 8
  ...
  %15 = getelementptr i8, i8* %12, i64 %14
  ...
  %19 = call i32 @getLine(i8* %15, i32 %18)
  ...
}
```

Začínáme s lokálním polem `%input`. Prvním voláním `getUsers()` dostaneme `%5`, druhým `%6`. Poté se přesuneme do funkce `getInput` k parametru `%dest`. Hodnota `%dest` se zapisuje do paměťové proměnné `%2` – přejdeme k ní. Z proměnné je čteno do hodnoty `%12` – přejdeme k ní. Z `%12` se vyrobí `%15` a ta putuje do prvního parametru `getLine`. Dále bychom pokračovali vnitřkem funkce `getLine`, dokud bychom nenašli volání některé funkce uživatelského vstupu, nebo funkce kopírování.

5.1.3 Visitor pattern

Abychom omezili větvení a opakování velkých switch konstruktů pro mnoho typů instrukcí, používáme tzv. visitor pattern [36]. LLVM framework sice obsahuje nadtřídu `InstVisitor`, která obsahuje switch a visit metody pro všechny typy instrukcí (podtřídy `Instruction`), ale my potřebujeme umět reagovat na všechny typy hodnot (podtřídy `Value`). Naše implementace tedy definuje vlastní nadtřídu `ValueVisitor`, která dokáže řešit i hodnoty, které nejsou instrukce, např. `Argument`, `Constant` nebo `GlobalVariable`. Každý dílčí vyhledávací algoritmus pak dědí od třídy `ValueVisitor` a předefinovává visit metody pro jednotlivé instrukce, které sám potřebuje řešit.

5.2 Pole na zásobníku, na haldě a globální

Přestože původní záměr byl detekovat hlavně přetečení zásobníku, protože má nejhorší dopady, přetečení pole alokované dynamicky či pole globálního se implementačně moc neliší. Zásobníkové pole najdeme v LLVM kódu jako instrukce `AllocaInst`, dynamické pole jako volání `CallInst` s názvem funkce `malloc` či `calloc`, a globální pole jako `GlobalVariable` u které `->isConstant()` vrací true. Velikost pole je u zásobníkového a globálního dána atributem instrukce a u dynamického je to argument volání.

Jako bonus tedy nástroj dokáže detekovat i potenciální přetečení na haldě či v prostoru globálních proměnných.

5.3 Znovupoužití výsledků

Během prohledávání se často stává, že algoritmus znovu navštěvuje funkce, které již prohledal dříve, ale z jiného počátečního místa. Pro omezení těchto zbytečných výpočtů bylo implementováno ukládání mezivýsledků formou prohledávacího kontextu, který uchovává některé informace o již navštívených funkcích. Když algoritmus projde funkci a nenajde žádný vstup, označí ji jako „read only“, a při příští návštěvě ji přeskočí.

5.4 Intermodulární vyhledávání

LLVM mezikód vzniká během kompilace konkrétní kompilační jednotky zdrojového jazyka (v C soubory s příponou .c, v C++ .cpp, ...), ne až při linkování. To znamená, že souborů s LLVM mezikódem je tolik, kolik je kompilačních jednotek. V praxi ale málokterý program obsahuje pouze jeden zdrojový soubor a zranitelnosti pak často vznikají na hranicích několika modulů.

Pro zvýšení praktické použitelnosti bylo přidáno rekurzivní vyhledávání definic funkcí v sousedních modulech. V LLVM, funkce, které jsou v modulu volány ale nejsou zde definovány, mají přítomnou deklaraci bez těla funkce. Tyto deklarace program na začátku projde, a pokusí se je vyhledat v ostatních LLVM modulech ve stejné složce. Pokud je definice v některém modulu nalezena, tento modul je sloučen s aktuálním a hledání definic deklarovaných funkcí rekurzivně pokračuje pro tento modul.

5.5 Konfigurační soubor

Při implementaci a testování bylo primárně cíleno na jazyk C, a podle toho byly konstruovány seznamy kritických funkcí a funkcí provádějících operace nad řetězci. Pro větší univerzálnost se ale tyto funkce dají specifikovat v externím konfiguračním souboru ve formátu JSON. Zároveň může programátor zadat názvy vlastnoručně napsaných funkcí pro sanitaci vstupu a algoritmus je pak bude brát jako místo ukončení škodlivého vlivu vstupu.

Bonusovým efektem tohoto přístupu je, že stejnou metodou nyní lze detekovat i zranitelnost injekce SQL příkazu, protože její povaha je stejná jako injekce systémového příkazu. Přestože práce s databázemi nebývá u nízkoúrovňových jazyků obvyklá, knihovny pro klienty SQL databází existují a na jejich použití lze občas narazit. Jednoduchým přidáním příslušných funkcí dané knihovny do konfiguračního souboru dokážeme detekovat i chyby v takovýchto programech.

Kapitola 6

Testování a výsledky

Nástroj byl otestován na 2 sadách programů specializovaných pro testování statických analyzátorů – *Juliet Test Suite* a *C Test Suite for Source Code Analyzer v2*. První z nich sice obsahuje velké množství vzorků lišících se ve všech možných aspektech, ale má pouze zranitelné programy a nezabývá se metodami ošetření vstupů, takže nelze otestovat množství falešných poplachů. Druhá se skládá jak ze zranitelných programů, tak z jejich opravených variant, ale bohužel jich je málo.

6.1 Juliet Test Suite

Sada *Juliet Test Suite* obsahuje vzorky kategorizované podle několika kritérií. Prvním kritériem je typ zranitelnosti podle kategorizace CWE. Z nich byly použity pouze ty, na které náš nástroj cílí – CWE-78 (injekce OS příkazu), CWE-121 (přetečení bufferu na zásobníku), CWE-134 (neověřený formátovací řetězec). Druhým kritériem je způsob, jakým se uživatelský vstup dostává do programu. Jedna skupina vzorků používá čtení z konzole, druhá ze souboru, třetí ze socketu atd. Další podobné kritérium je způsob použití načtených dat. Například u injekce příkazu některé vzorky používají `system`, další `popen`, jiné `execl` s prvním argumentem `"/bin/sh"` atd. Poslední kritérium je způsob propagace dat, v některých vzorcích je použití ve stejné funkci jako načtení, v jiných data prochází několika funkcemi a v dalších dokonce napříč několika kompilačními jednotkami. Vzorků je celkem 64099, ale to je dáno tím, že sada se snaží o úplné kombinatorické pokrytí všech možností u všech kategorií. Pro naše účely nemá cenu testovat všechny, ale pouze ty určitých vybraných kategorií.

Pro otestování injekce příkazu stačí vyzkoušet všechny způsoby toku dat. Nemá cenu zkoušet varianty lišící se pouze funkcemi použití (alternativami na `system`), protože pro ně náš nástroj poběží úplně stejně, pouze si vybere jiný název funkce z konfiguračního souboru. Stejně tak pro načítání dat (funkce `fgets` se zamění za `fread` nebo `recv`). Postup testování formátovacího řetězce

je úplně stejný, protože povaha zranitelnosti je stejná a na oba případy se používá stejný algoritmus. Otestovat přetečení bufferu je o něco složitější, protože způsobů, jak k němu může dojít je mnoho. Vzorky byly tedy vybrány tak, aby pokryly všechny způsoby aspoň jedním případem, a konkrétním způsobům propagace dat již nebyla věnována taková pozornost. Výsledky všech třech typů zranitelností jsou shrnuty tabulkou 6.1.

typ zranitelnosti	počet otestovaných	počet detekovaných	úspěšnost
CWE-78	38	31	82 %
CWE-134	38	32	84 %
CWE-121	46	14	30 %

Tabulka 6.1: Výsledky pro sadu Juliet Test Suite

Injekce příkazu a formátovacího řetězce, na které nástroj primárně cílí, jsou detekovány celkem spolehlivě, a to včetně případů, kdy data procházejí napříč několika moduly. Nedetekované případy byly většinou způsobeny větší úrovní indirekce, jako pole ve struktuře, pole v poli či pole v unionu, a také použitím ukazatelů na funkce, se kterými si nástroj momentálně neví rady. Výsledky pro přetečení bufferu nejsou přesvědčivé, protože způsobů, jak může dojít k přetečení je nepřeberné množství a náš nástroj z povahy dokáže rozpoznat jenom některé z nich.

6.2 C Test Suite for Source Code Analyzer v2

Tato sada je rozdělena na 2 části, vzorky obsahující zranitelnost a vzorky, které danou zranitelnost opravují. Máme tedy stejné množství pozitivních a negativních případů a můžeme otestovat počet falešných poplachů. Vzorky jsou nahodilou směsí nejčastějších typů zranitelností. Bohužel příkladů je dohromady jen 102 a těch, které jsou požadovaného typu, je z nich jenom 25 (každý ve zranitelné a opravené variantě). Bylo tedy otestováno 25 zranitelných vzorků s požadavkem, že nástroj zranitelnost zahlásí a 25 čistých vzorků s požadavkem, že nástroj nezahlásí nic. Kromě toho se ale ještě několikrát stalo, že nástroj zahlásil zranitelnost v jiné nesouvisející části kódu, kde ve skutečnosti žádná nebyla. Takové případy byly započítány jako kdyby to byly dodatečné čisté vzorky, u kterých nástroj hlásil positivnost.

Protože jde v zásadě o klasifikační problém, výsledky lze vyjádřit maticí záměn (anglicky „confusion matrix"). Matice 6.2 zobrazuje počty jednotlivých klasifikací a matice 6.3 přepočet na procenta.

Vynechané zranitelnosti nastaly opět kvůli poli ve struktuře, ale také kvůli vlastním funkcím zapisujícím ručně přes indexy. Falešné poplachy byly způsobeny převážně ruční kontrolou obsahu bufferu s podmíněným ukončením a také složitějšími nekonstantními výrazy v místě argumentu velikosti pole.

	detekováno	nedetekováno
zranitelný	20	5
čistý	12	20

Tabulka 6.2: Matice záměn pro sadu C Test Suite v2

	detekováno	nedetekováno
zranitelný	80 %	20 %
čistý	38 %	62 %

Tabulka 6.3: Matice záměn pro sadu C Test Suite v2 - v procentech

6.3 Zhodnocení

Kromě implementačně orientovaných problémů, jako neschopnost pracovat s polem ve struktuře, potvrdilo testování 2 hlavní slabiny této metody, a to že metoda si neuvědomuje pořadí provádění operací a podmínky, za kterých se operace provedou. Pokud se například příkaz spouští z globálního pole, je téměř nemožné bez symbolické exekuce určit, z jakého místa v programu bylo do pole naposledy zapsáno. Pokud je nebezpečná operace ohraničena podmínkou, která omezuje její zneužití, je opět čistou statickou analýzou nemožné určit, kdy bude splněna. V obou případech tak dochází k falešným poplachům.

Na druhou stranu, s několika vylepšeními bude schopna odhalit téměř všechna potenciálně problémová místa, na která se potom mohou zaměřit podrobnější a časově náročnější analýzy, nebo programátoři při ruční kontrole kódu.

Kapitola 7

Možnosti budoucího rozšíření

Uvedeme několik možností, které by významně zvýšily použitelnost nástroje.

7.1 Analýza spustitelných souborů pomocí dekompilace do LLVM

Největší přínos by ale nástroj získal, pokud by pomocí něj bylo možné analyzovat spustitelné soubory. K tomu by ale bylo potřeba transformovat strojový kód do meziformy LLVM. Problém dekompilace je velmi složitý a uspokojivě ho neřeší ani velmi drahé komerční nástroje. Nicméně forma LLVM má ke strojovému kódu daleko blíže než například jazyk C a tudíž při překladu odpadá řada problémů.

Pro účely naší analýzy datových toků musí dekompilátor do LLVM splňovat tyto podmínky:

- funguje (nepadá, vyprodukuje aspoň nějaký výsledek) pro soubory zkompilované běžnými kompilátory – GCC, MSVC, clang
- rozlišuje mezi celým číslem a ukazatelem
- rozpozná lokální pole ve funkci
- rozpozná signatury funkcí (počet parametrů a jejich typy)

Open-source projektů, které se pokoušejí o dekompilaci do LLVM, existuje několik. Tyto projekty byly otestovány na výše uvedené podmínky a výsledky testů shrnuje tabulka 7.1. „Ne“ ve sloupci funguje znamená, že nástroj vyhodí chybu o nerozpoznané instrukci, spadne/zacyklí se uprostřed analýzy nebo selže komponenta třetí strany, kterou používá. Pomlčky ve všech 4 posledních sloupcích znamenají, že repositář se na aktuálních systémech vůbec nezkompiluje.

nástroj	formáty	architektury	funguje	ukazatel	pole	signatury
RetDec	PE, ELF	x86, ARM, MIPS, Sparc	ano	ne	ne	částečně
Dagger	ELF	x86	někdy	ne	ne	ne
mctoll	ELF	x64, ARM	ne	–	–	–
bin2llvm		x86, x64, ARM	–	–	–	–
rev.ng	ELF	x64, ARM, MIPS	ne	–	–	–
McSema	PE, ELF	x86, x64, ARMv8	ne	–	–	–
Fracture	ELF, Mach-O	x86, ARM, PowerPC	ne	–	–	–

Tabulka 7.1: Porovnání existujících dekompilátorů do LLVM

Na stránkách projektu McSema [38] lze také najít srovnání těchto nástrojů, ale z trochu jiného pohledu a jinými požadavky.

Bohužel v době provádění těchto testů (říjen 2018) ani jeden z dekompilátorů nebyl dostatečně vyvinutý na to, aby bylo možné náš algoritmus aplikovat na spustitelné soubory.

7.2 Abstrakce nad operacemi datových toků

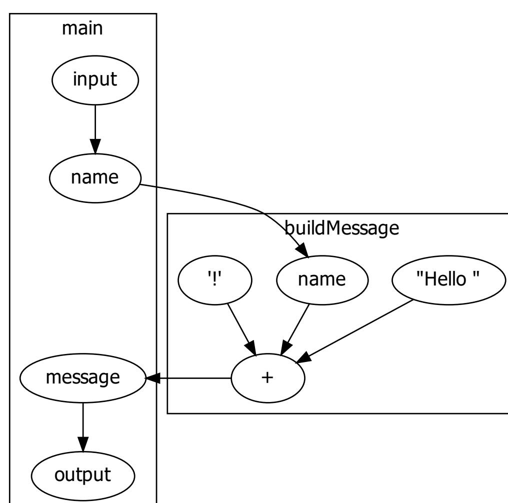
Současná implementace prochází datové toky ad-hoc přímo uvnitř vyhledávacího algoritmu, a to detekcí volání funkcí zadaných v konfiguračním souboru, jako `strcpy`, `strcat`, atd. Univerzálnější by bylo navrhnout nějakou abstraktní reprezentaci datových toků, nad kterou by probíhalo samotné hledání zdrojů dat, a tato abstraktní reprezentace by se konstruovala z LLVM kódu podle zvoleného jazyka (`strcat` pro C, operátor + pro C++, ...). Hlavní část nástroje by pak byla více nezávislá na zdrojovém jazyce a pouze část konstrukce abstraktní formy by musela řešit jazykově specifické konstrukty. Například jak kód 7.1 tak 7.2 by se převedly do podoby na obrázku 7.1

```
void buildMessage( char* name, char* dest )
{
    strcpy( dest, "Hello " );
    strcat( dest, name );
    int len = strlen( dest );
    dest[ len ] = '!';
    dest[ len+1 ] = '\0';
}
void main()
{
    char name[32], message[64];
    fgets( name, sizeof(name), stdin );
    buildMessage();
    fputs( message, stdout );
}
```

Kód 7.1: Kód C s datovými závislostmi

```
string buildMessage( string name )
{
    return "Hello " + name + '!';
}
void main()
{
    string name, message;
    cin >> name;
    message = buildMessage( name );
    cout << message;
}
```

Kód 7.2: Kód C++ s datovými závislostmi



Obrázek 7.1: Společný abstraktní datový tok

Závěr

V této práci byla prezentována metoda zpětného hledání původu dat v LLVM mezikódu a její aplikace na detekci zranitelností typu injekce příkazu, injekce formátovacího řetězce a některé případy přetečení bufferu. Pro ověření konceptu byla metoda implementována s pomocí LLVM frameworku do konzolového nástroje. Testy tohoto nástroje na sadě zranitelných a čistých programů potvrdily počáteční předpověď, a to že metoda může dosáhnout vysoké senzitivity (poměr správně nalezených zranitelností ku všem zranitelným případům), avšak za cenu velkého počtu falešných poplachů. Použitá samostatně, bez kombinace s jinými přístupy, tedy nemá příliš velkou praktickou použitelnost. Avšak může sloužit jako předběžná analýza poukazující na potenciálně problémová místa, na která se potom může zaměřit jiná sofistikovanější a časově náročnější analýza, jako například symbolické provádění. Vytvořený nástroj tedy není vhodný pro nasazení do produkčních build systémů, ale může sloužit jako vodítko při kontrole kvality kódu a dávat rady, na které části programu se zaměřit.

Literatura

[1] MCCANDLESS, David. *Knowledge is beautiful.* New York, NY: Harper Design, an imprint of HarperCollinsPublishers, 2014. ISBN 978-0062188229.

[2] SPAFFORD, Eugene. *The Internet Worm Program: An Analysis* [online]. West Lafayette, 1988 [cit. 2019-05-08]. Dostupné z: https://spaf.cerias.purdue.edu/tech-reps/823.pdf. Technical Report. Purdue University.

[3] *CWE - Common Weakness Enumeration* [online]. Bedford: The MITRE Corporation, c2006-2019 [cit. 2019-05-08]. Dostupné z: https://cwe.mitre.org

[4] The MITRE Corporation [online]. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. 2011. [cit. 2019-05-08]. Dostupné z: https://cwe.mitre.org/top25/archive/2011/2011_cwe_sans_top25.pdf

[5] OWASP Foundation [online]. *OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks*. 2017. [cit. 2019-05-08]. Dostupné z: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[6] Sourcefire, Inc [online]. *25 Years of Vulnerabilities: 1988-2012*. 2013. [cit. 2019-05-08] Dostupné z: https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf

[7] *CVE - Common Vulnerabilities and Exposures* [online]. Bedford: The MITRE Corporation, c1999–2019 [cit. 2019-05-08]. Dostupné z: https://cve.mitre.org

[8] *CVE Details* [online]. San Francisco: Özkan, [cit. 2019-04-09]. Dostupné z: https://www.cvedetails.com

[9] Security Vulnerabilities Published In 2018(Overflow). *CVE Details* [online]. San Francisco: Özkan, 2016, 2018 [cit. 2019-04-09]. Dostupné z: `https://www.cvedetails.com/vulnerability-list/year-2018/opov-1/overflow.html`

[10] ZAHRADNICKÝ, Tomáš. *Bezpečný kód: Přetečení bufferu* [online]. 2015. [cit. 2019-05-08]. Dostupné z: `https://moodle.fit.cvut.cz/pluginfile.php/86748/mod_folder/content/0/bek03cz-Buffer_overflow.pdf`

[11] KUZIN, Mikhail, Yaroslav SHMELEV a Vladimir KUSKOV. New trends in the world of IoT threats. *Securelist* [online]. Moscow: AO Kaspersky Lab, 2018, 18.9.2018 [cit. 2019-04-09]. Dostupné z: `https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/`

[12] ASERT Team. Fast & Furious IoT Botnets: Regifting Exploits. *Netscout* [online]. Westford: NETSCOUT Systems, 2019, 12.12.2018 [cit. 2019-04-09]. Dostupné z: `https://securelist.com/new-trends-in-the-world-of-iot-threats/87991/`

[13] Team Teso. *Exploiting Format String Vulnerabilities* [online]. Stanford, 2001 [cit. 2019-04-09]. Dostupné z: `https://cs155.stanford.edu/papers/formatstring-1.2.pdf`. Technical Report. Stanford University.

[14] ERICKSON, Jon. *Hacking: the art of exploitation*. San Francisco: No Starch Press, c2003. ISBN isbn-1-59327-007-0.

[15] M. Sutton, A. Greene, and P. Amini,Fuzzing: Brute Force Vulnera-bility Discovery. Addison-Wesley Professional, 2007.

[16] SOTIROV, Alexander Ivanov. *AUTOMATIC VULNERABILITY DETECTION USING STATIC SOURCE CODE ANALYSIS*. Tuscaloosa, 2005. Master thesis. University of Alabama.

[17] XING, Zhang, Zhang BIN, Feng CHAO, Zhang QUAN, T. GONG, T. YANG a J. XU. Staticly Detect Stack Overflow Vulnerabilities with Taint Analysis. *ITM Web of Conferences* [online]. 2016, 7 [cit. 2019-04-09]. DOI: 10.1051/itmconf/20160703003. ISSN 2271-2097. Dostupné z: `http://www.itm-conferences.org/10.1051/itmconf/20160703003`

[18] MIHAILA, Bogdan. Adaptable Static Analysis of Executables for proving the Absence of Vulnerabilities. München, 2015. Dissertation. Technischen Universität München. Vedoucí práce Prof. Tobias Nipkow, Ph.D.

[19] Saluki Finding Taint-style Vulnerabilities with Static Property Checking GOTOVCHITS, Ivan, Rijnard van TONDER, David BRUMLEY. Saluki: Finding Taint-style Vulnerabilities with Static Property Checking.

Network and Distributed Systems Security (NDSS) Symposium. 2018, 2 [cit. 2019-04-09]. ISBN 1-1891562-49-5.

[20] J. J., Kronjee. Discovering vulnerabilities using data-flow analysis and machine learning. Valkenburgerweg, 2018. Master thesis. Open University. Vedoucí práce Dr. A. J. Hommersom.

[21] PATTERSON, Jason R. C. Accurate static branch prediction by value range propagation. *ACM SIGPLAN Notices* [online]. 1995, 30(6), 67-78 [cit. 2019-04-09]. DOI: 10.1145/223428.207117. ISSN 03621340. Dostupné z: `http://portal.acm.org/citation.cfm?doid=223428.207117`

[22] KING, James C. Symbolic execution and program testing. *Communications of the ACM* [online]. 19(7), 385-394 [cit. 2019-04-09]. DOI: 10.1145/360248.360252. ISSN 00010782. Dostupné z: `http://portal.acm.org/citation.cfm?doid=360248.360252`

[23] Zardus. A Dozen Years of Shellphish: From DEFCON to the Cyber Grand Challenge. *Hitcon* [online]. 2015 [cit. 2019-04-09]. Dostupné z: `https://hitcon.org/2015/CMT/download/day2-g-r0.pdf`

[24] IDA F.L.I.R.T. Technology: In-Depth. *Hex Rays* [online]. Belgium: Hex-Rays, 2019, 9.4.2019 [cit. 2019-04-09]. Dostupné z: `https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml`

[25] ROSEN, B. K., M. N. WEGMAN a F. K. ZADECK. Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88* [online]. New York, New York, USA: ACM Press, 1988, 1988, s. 12-27 [cit. 2019-04-09]. DOI: 10.1145/73560.73562. ISBN 0897912527. Dostupné z: `http://portal.acm.org/citation.cfm?doid=73560.73562`

[26] DENKER, Marcus. Static Single Assignment Form. *Marcus Denker* [online]. Bern: University of Berne, 2008 [cit. 2019-04-09]. Dostupné z: `https://marcusdenker.de/talks/08CC/08IntroSSA.pdf`

[27] DENKER, Marcus. Optimizations. *Marcus Denker* [online]. Bern: University of Berne, 2008 [cit. 2019-04-09]. Dostupné z: `https://marcusdenker.de/talks/08CC/08IntroSSA.pdf`

[28] GCC Developer Community. GNU Compiler Collection (GCC) Internals. *GCC, the GNU Compiler Collection* [online]. Free Software Foundation, c1988-2019 [cit. 2019-04-09]. Dostupné z: `https://gcc.gnu.org/onlinedocs/gccint/`

[29] MERRILL, Jason. GENERIC and GIMPLE: A new tree representation for entire functions. *In Proc. GCC Developers Summit.* 2003, str. 171–180.

[30] LATTNER, C. a V. ADVE. LLVM: A compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004* [online]. IEEE, 2004, str. 75–86 [cit. 2019-04-09]. DOI: 10.1109/CGO.2004.1281665. ISBN 0-7695-2102-9. Dostupné z: `http://ieeexplore.ieee.org/document/1281665/`

[31] LLVM Language Reference Manual. The LLVM Compiler Infrastructure Project [online]. LLVM Foundation [cit. 2019-04-09]. Dostupné z: `https://releases.llvm.org/3.8.1/docs/LangRef.html`

[32] LLVM Programmer's Manual. The LLVM Compiler Infrastructure Project [online]. LLVM Foundation [cit. 2019-04-09]. Dostupné z: `https://releases.llvm.org/3.8.1/docs/ProgrammersManual.html`

[33] BALZAROTTI, Davide, Marco COVA, Vika FELMETSGER, Nenad JOVANOVIC, Engin KIRDA, Christopher KRUEGEL a Giovanni VIGNA. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In: *2008 IEEE Symposium on Security and Privacy (sp 2008)* [online]. IEEE, 2008, 2008, s. 387-401 [cit. 2019-04-09]. DOI: 10.1109/SP.2008.22. ISBN 978-0-7695-3168-7. ISSN 1081-6011. Dostupné z: `http://ieeexplore.ieee.org/document/4531166/`

[34] Working Draft, Standard for Programming Language C++. Open Standards [online]. 27.11.2017 [cit. 2019-04-09]. Dostupné z: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf`

[35] CADAR, Cristian, Daniel DUNBAR a Dawson R. ENGLER. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *8th USENIX Symposium on Operating Systems Design and Implementation · OSDI 2008*. San Diego, 2008, 8, str. 209–224.

[36] MARTIN, Robert C. Visitor. University of Turku [online]. Turun Yliopisto: University of Turku, 2002 [cit. 2019-04-09]. Dostupné z: `http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/Visitor.pdf`

[37] Software Assurance Reference Dataset [online]. Gaithersburg: National Institute of Standards and Technology, 2016 [cit. 2019-04-09].

[38] McSema. GitHub [online]. GitHub, c2019 [cit. 2019-04-09]. Dostupné z: `https://github.com/trailofbits/mcsema`

Příloha **A**

Seznam použitých zkratek

ASLR Address space layout randomization

AST Abstract syntax tree

CFG Control Flow Graph

CVE Common Vulnerabilities and Exposures

CWE Common Weakness Enumeration

SSA Single Static Assignment

Příloha **B**

Obsah přiloženého CD

- `readme.txt` .... tento popis obsahu CD
- `Text` .... textová část práce
  - `obrazky` .... obrázky použité v práci a jejich zdrojové kódy
  - `DP_Brož_Jan_2019.tex` .... zdrojová forma textu ve formátu LaTeX
  - `DP_Brož_Jan_2019.pdf` .... text práce ve formátu PDF
- `VulnFinder` .... implementační část práce
  - `src` .... soubory se zdrojovým kódem
  - `nbproject` .... soubory projektu NetBeans