ENHANCEMENT OF DECOMPILATION BY USING DYNAMIC CODE ANALYSIS

Jaromír Končický

Master Degree Programme (2), FIT BUT

E-mail: xkonci01@stud.fit.vutbr.cz

Supervised by: Jakub Křoustek

E-mail: ikroustek@fit.vutbr.cz

Abstract: This paper describes dynamic code analysis and its advantages over static code analysis. Dynamic analysis is used to improve the decompilation process and the results of a retargetable decompiler developed within the Lissom project at FIT BUT. Dynamic analysis can provide many kinds of information about the executed program. It is significantly utilized in malware analysis and complex data-type analysis, which are also described in this paper.

Keywords: decompiler, dynamic analysis, malware analysis, data-type analysis, Lissom

1. ÚVOD

Zpětný překlad je proces, jehož cílem je z binárního spustitelného programu vytvořit zdrojový kód tohoto programu ve vyšší formě reprezentace. Jedná se tedy o proces opačný k činnosti překladačů, které ze zdrojového kódu vytváří spustitelné soubory. Ačkoli překladačů dnes existuje celá řada a jejich úroveň je vysoká, v případě zpětných překladačů tomu tak není.

Zpětný překlad je velice obtížný a většinou není možné tímto způsobem získat kód shodný s původním zdrojovým kódem programu. Při překladu do strojového kódu totiž dochází ke ztrátě vysokoúrovňových informací. Procesor ke své činnosti nepotřebuje znát názvy proměnných, funkcí a konstrukce vyšších jazyků. Zpětný překladač musí na základě složitých analýz tyto informace rekonstruovat. Současným cílem je vyvíjet nové techniky, pomocí kterých je možno získat stále dokonalejší a přesnější výsledný kód. Jednou z nich je právě dynamická analýza.

V rámci projektu Lissom, probíhajícím na FIT VUT v Brně, je vyvíjen rekonfigurovatelný zpětný překladač [1]. Předmětem této práce je dynamická analýza, jejímž cílem je zlepšit výsledky tohoto zpětného překladače.

2. DYNAMICKÁ ANALÝZA

Při statické analýze je zkoumán samotný spustitelný soubor, aniž by byl jakkoli spuštěn nebo vykonáván. Strojové instrukce jsou dekódovány do vnitřní reprezentace, nad níž jsou pak prováděny mnohé analýzy s cílem nalezení co největšího množství vysokoúrovňových informací. Statická analýza má však mnohá omezení. Některé informace, potřebné pro zpětný překlad, jsou známy pouze za běhu programu. Jedná se například o hodnoty v registrech či v paměti v určitém bodu vykonávání, cíle nepřímých volání a skoků, hodnoty parametrů volaných funkcí atd.

Podstatou dynamické analýzy je spuštění zkoumaného programu a získávání informací o něm za jeho běhu. Tímto způsobem je navíc možno získat představu o tom, co program vykonává a jaká je jeho funkce, což při statické analýze nemusí být hned zcela zřejmé.

Za běhu programu jsou zaznamenávány a shromažďovány informace různého druhu, které jsou posléze dodány zpětnému překladači. Zde poslouží v případech, kdy statická analýza nedostačuje, a napomohou k lepším a přesnějším výsledkům. Příkladem je detekce začátků funkcí a cílů nepří-

mých volání, kde jsou využity informace o cílových adresách volání. Využity jsou rovněž informace o strukturovaných datových typech z typové analýzy (více popsané v kapitole 4).

Dynamická analýza je implementována pomocí instrumentačního nástroje Pin [2] společnosti Intel.

3. ANALÝZA ŠKODLIVÝCH PROGRAMŮ

Velké množství škodlivého softwaru (angl. malware), jeho rychlé šíření a dopady na informační technologie jsou stále vážným problémem. K jeho detekci a odstraňování jsou vyvíjeny antivirové programy. Pro jejich úspěšný vývoj je nutné nejprve porozumět chování škodlivých programů a jejich vnitřní struktuře. Pro tento účel jsou využívány mj. zpětné překladače. Zpětný překlad a vůbec pochopení činnosti škodlivých programů jsou pomocí statické analýzy mnohdy obtížné nebo i nemožné. Jedním z důvodů je to, že programy jsou často různými způsoby proti analýze chráněny (např. obfuskací a šifrováním). Dynamická analýza v tomto případě může významně pomoci [3].

Důležitým krokem analýzy je zjištění, co škodlivý program v operačním systému provádí a jaké změny v něm způsobuje. Každý program komunikuje se svým okolím výhradně pomocí systémových knihovních funkcí, a to pro práci se soubory, práci se sítí apod. Součástí dynamické analýzy je proto záznam volání všech potenciálně nebezpečných funkcí i jejich parametrů.

```
CreateFileA("C:\Documents and Settings\VM\Insert.vbe", 0xc0000000, 0, 0, 2, 128, 0) = 1872
WriteFile(1872, "
  NameUrlDown = "http://www.aerotecnicastar.it/images/stories/ZIMLOX"
  LocalMachineEnd = ComponentScriptT.ExpandEnvironmentStrings("%userprofile%")+"\drone.tmp"
  Set ObjZeusD = CreateObject("MSXML2.XMLHTTP")
  ObjZeusD.open "GET", NameUrlDown, false
  ObjZeusD.send()
", 2737, 0x12fd6c, 0x0) = true
WinExec("wscript.exe /B "C:\Documents and Settings\VM\Insert.vbe"", 6) = 33
RegOpenKeyW(0x8000002,"Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers", 0x12ed5)
DeleteFileA("C:\Documents and Settings\VM\Insert.vbe") = true
```

Příklad 1: Ukázka záznamu volání knihovních funkcí škodlivým programem.

V příkladu 1 je uvedena posloupnost volání vybraných knihovních funkcí škodlivým programem, z níž je patrná jeho činnost. Program nejprve vytvoří soubor `Insert.vbe` a do něj zapíše tělo skriptu, který následně spustí. Činností skriptu je stažení souboru z internetu a další akce s ním (v příkladu neuvedeny). Program poté změní hodnotu v registru systému Windows a skript smaže.

4. ANALÝZA STRUKTUROVANÝCH DATOVÝCH TYPŮ

Strukturované datové typy (pole a struktury) jsou nedílnou součástí složitějších programů. Znalost podoby a významu vnitřních datových struktur je mnohdy klíčová pro analýzu a pochopení funkce programu. Ve zpětném překladači projektu Lissom je implementována statická typová analýza, rovněž jsou využity typové informace z dynamické analýzy (se všemi výhodami s ní spojenými).

Relativně nejsnazší je typová analýza dynamicky alokované paměti. Programy dynamicky alokují jednotlivé bloky paměti, přičemž každý takový blok obsahuje oddělenou datovou strukturu. Předmětem analýzy je v tomto případě rozpoznání vnitřní struktury každého bloku paměti.

Během vykonávání programu jsou zaznamenány veškeré přístupy (čtení a zápisy) do každého bloku paměti. Ke každému přístupu je uložena informace o instrukci, která do paměti přistoupila, adrese přístupu (počtu bytů od začátku bloku) a další informace, jako velikost a typ dat. Jsou tak sestaveny tzv. „vzory přístupů“, na jejichž základě se pomocí speciálního algoritmu odvodí struktura tohoto bloku. Tato metoda byla inspirována [4].

V příkladu 2 jsou uvedeny vzory přístupů (pouze zápisy) do paměťového bloku obsahující strukturu `struct s`. Na jejich základě je zpětně odvozena podoba této struktury. Výstupem typové analýzy je pak samotná definice složitých typů a jejich mapování na instrukce přistupující do nich.

```
write_int   inst_addr: 0x4013b3 data_size: 4 offsets:  0
write_int   inst_addr: 0x4013c0 data_size: 1 offsets:  4
write_float inst_addr: 0x4013d4 data_size: 4 offsets:  8
write_int   inst_addr: 0x4013f6 data_size: 4 offsets: 12 16 20 24
```

```
struct s {
    int b;
    char c;
    float f;
    int e[4]
}
```

Příklad 2: Jednoduché vzory přístupů a struktura daného typu.

V příkladech 4 a 5 jsou uvedeny experimentální výsledky zpětného překladu kódu z příkladu 3 bez typové analýzy a s použitím dynamické typové analýzy. Její přínos je zřejmý.

```
int main(void) {
   struct s *x = malloc(sizeof(struct s));
   x->b = rand();
   x->c = (char) rand();
   x->f = rand();
   for (int i=0; i<4; i++)
      x->e[i] = rand();
   return 0;
}
```

Příklad 3: Původní zdrojový kód.

```
int main(int argc, char **argv) {
    int32_t * mem = malloc(28);
    *mem = rand();
    *(int8_t *)(mem + 4) = (int8_t)rand();
    *(float32_t *)(mem + 8) = (float32_t)rand();
    int32_t banana = mem + 12;
    for (uint32_t i = 0; i < 4; i++)
        *(int32_t *)(banana + 4 * i) = rand();
    return 0;
}
```

Příklad 4: Výsledný kód bez typové analýzy**.**

```
struct struct_0 {
    int32_t e0;
    int8_t e1;
    float32_t e2;
    float32_t e3[4];
};

int main(int argc, char **argv) {
    struct struct_0 * mem =
        (struct struct_0 *)malloc(28);
    mem->e0 = rand();
    mem->e1 = rand();
    mem->e2 = rand();
    for (uint32_t i = 0; i < 4; i++) {
        mem->e3[i] = rand();
    }
    return 0;
}
```

Příklad 5: Výsledný kód s dynamickou typovou analýzou.

5. ZÁVĚR

V tomto článku byla představena dynamická analýza a její význam z hlediska zpětného překladu. Zmíněno bylo především využití jejích výsledků ve zpětném překladači a její výhody oproti statické analýze. Zdůrazněno bylo využití dynamické analýzy v oblasti analýzy škodlivého softwaru, což je jeden z hlavních důvodů vývoje zpětných překladačů. Důležitou součástí dynamické analýzy je analýza strukturovaných datových typů, jež zde byla podrobněji popsána.

PODĚKOVÁNÍ

Tento příspěvek vznikl za podpory grantu FIT-S-14-2299 - Výzkum pokročilých metod ICT a jejich aplikace.

REFERENCE

[1] Ďurfina, L., Křoustek, J., Zemek, P.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. In: IJSIA, Vol. 5, No. 4, p. 91-106, 2011

[2] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Programming Language Design and Implementation, Chicago, IL, 2005.

[3] Bayer, U., Moser, A., Kruegel, C., Kirda, E.: Dynamic analysis of malicious code. In Journal in Computer Virology 2, 66 – 67, 2006.

[4] Troshinaa, E. N., Chernov, A. V.: Using Information Obtained in the Course of Program