MI-PB-14

Disasemblování: lineární a rekurzivní průchod. Obfuskační techniky.

Disassembling: proces překladu binárního kódu do kódu v člověkem čitelném assembleru cílového CPU

- Nástroj: **disassembler**
- Výsledek: **disassembly** -- statická analýza kódu
  - hodně řádek kódu
  - postrádá množství informace, která existovala v čase kompilace
  - míchá kód a data

Lineární průchod

Disassembluje lineárně byte po bytu od začátku sekce `.text` až do jejího konce

Rychlý, jednoduchý

Nevýhody:

- Nejednoznačný tok kódu: Instrukce následovaná nějakými byty se může jevit jako jiná instrukce
- Promíchaný kód a data -- data se taky disassemblují

Rekurzivní průchod

Disassembluje kód na základě sledování jeho toku

Začíná vstupním bodem programu, zaznamenává si, jaké instrukce na jakých adresách již navštívil, pamatuje si návratové adresy skoků, rekurzivně analyzuje větve skoku

Pomalejší
Rozlišuje kód od dat (nenavštívené adresy jsou data)

Problém: Nepřímé skoky (skok na adresu z tabulky skoků), vypočítané `goto`, volání z VMT

Rozšířený lineární průchod

Tabulky skoků musí mít v `.text` záznamy v tabulce relokací -- dají se tak rozpoznat od dat.

Rozdělení `.text` na segmenty oddělené tabulkami skoků, disassemblovat jejich kód a ověřovat úspěšnost disassembly každé instrukce (že nepřetekla mezi segmenty)

- Všechny tabulky skoků jsou data, kromě prvních dvou záznamů v každé tabulce
- Pro každou posloupnost neoznačených adres v sekci `.text`:
  - Použij lineární průchod, zastav po dosažení označené adresy
  - Pokud poslední instrukce přetekla do označené adresy, označ ji jako data
  - Prozkoumej další úspěšně disasemblovanou instrukci a v případě potřeby ji označ jako data

Hybridní přístup

Prvotní disassemblování: rozšířený lineární průchod

Ověření každé nalezené funkce: rekurzivní průchod
Pokud instrukce získaná lineárním průchodem nebyla nalezena i rekurzivním, vyvolej chybu. Pokud ve funkci nebyla chyba, je funkce správná

Obfuskace

Činí objekt nesrozumitelným. Změna transformačního cíle tak, aby mu bylo obtížné rozumět, ale aby zůstala zachována jeho funkcionalita.

Použití: zabránění reverzování schémat sériových čísel, ochrana proti kopírování, zabránění analýzy malwaru

Cíl: Zmást disassembler, debugger, decompiler a analytika

Metriky:

- **Potence:** jak moc transformace P na P' zmate lidského čtenáře

$\mathcal{T}_{pot}(P) = \frac{E(P')}{E(P)} - 1$, kde $E(P)$ je složitost P podle nějaké metriky (délka programu, cyklomatická složitost, složitost vnoření, složitost datového toku...).

- **Odolnost:** jak obtížné je odstranit transformaci automatickým deobfuskátorem
  $\mathcal{T}_{res} = \text{Resilience}(\mathcal{T}_{\text{práce deobfuskátoru}}, \mathcal{T}_{\text{práce programátora}})$
  kde $\mathcal{T}_{\text{práce programátora}}$ je množství času potřebné pro vytvoření automatického deobfusátoru, který sníží $\mathcal{T}_{pot}$, a $\mathcal{T}_{\text{práce deobfuskátoru}}$ je doba běhu a velikost prostoru, které auto-deobfuskátor použije.
  $\mathcal{T}_{res} \in \{\text{triviální}, \text{slabá}, \text{silná}, \text{plná}, \text{jednosměrná}\}$

$\mathcal{T}_{pr.\,deob.} \setminus \mathcal{T}_{pr.\,prog.}$	Lokální	Globální	Meziprocedurová	Meziprocesová
Polynomiální	triviální	slabá	silná	plná
Exponenciální	slabá	silná	plná	plná

- **Cena:** Musí být co nejnižší

Obfuskace rozložení:

- Změna formátování kódu, jmen proměnných, funkcí, rozložení tříd a datových struktur, odstranění komentářů
- Nulová cena, jednosměrná odolnost

Transformace řízení:

- Obfuskace toku kódu
- Mnohdy závisí na **neprůhledných predikátech/proměnných:**
  - Hodnota je známá obfuskujícímu, ale ne deobfuskátorovi

```
if( ovar == VAL )
  nikdy nenastane
else
  něco udělej
```

```
if( ovar == VAL )
  něco udělej
else
  nikdy nenastane
```

```
if( ovar == VAL )
  něco udělej
else
  něco udělej
```

  - Lineární průchod nepozná nedosažitelnou část, rekurzivní průchod pozná snadno
- Dělení:
  - **Transformace agregace:** rozdělují výpočty, které patří k sobě
    - Inlining: tělo funkce se rozkopíruje do míst volání
    - Outlining: část kódu vytržena, přeměněna na samostatnou funkci
    - Interleaving: spojení kódu a argumentů dvou nebo více funkcí do jedné funkce s extra argumentem, který rozdohuje o tom, který kód které funkce se vykoná
    - Klonování: Více instancí jedné třídy, které volají funkce nezávislé na konkrétní instanci
    - Transformace smyček:

        - Loop blocking: rozdělení těla smyčky tak, aby se její části vešly do cache
        - Loop unrolling: replikace těla smyčky několikrát po sobě (eliminace skoků)
        - Loop fission: dělí smyčku na několik samostatných s různými těly (ty pak mohou běžet třeba paralelně)
    - **Transformace pořadí:** náhodné pořadí výpočtů
    - **Transformace výpočtu:**
        - Přidání zbytečného/mrtvého kódu (neprůhledné predikáty)
        - Rozšíření podmínek cyklů -- komplikace rozhodnutí o ukončení smyčky

          ```
          for(i=0, j=1, k=0; (k<170) && (j%2==1); i++, k+=17 )
          ```
        - Konverze reducibilního Code Flow Grafu na nereducibilní
            - CFG reducibilní, pokud používá pouze strukturované příkazy, např.

              ```
              for, do-while, while, if-then-else, break, continue
              ```
            - Přidání `goto` -- graf nereducibilní
            - Ošklivý kód vyšší úrovně
        - Odstranění knihovních volání a programovacích idiomů
            - Vlastní altermativy ke knihovním funkcím -- analytik nezná jména, kód je delší
        - Tabulková interpretace
            - Rozdělení programu do několika kusů, ve smyčce pomocí `switch` kusy spouštět
            - Tabulka klidně i více úrovní
            - Běh programu v miniaturním VM
        - Předání redundantních argumentů
        - Paralelizace kódu
            - Vlákna provádí datově nezávislé bloky kódu zároveň
            - Datově závislý kód: Každý kus kódu provádí sekvenčně jiné vlákno, ostatní čekají
            - Vlákna mohou provádět neužitečný kód