基於OLLVM之常數分離混淆

Based on OLLVM Constant Value Split Obfuscation

黃建豪 學生　吳育松 指導教授
國立陽明交通大學
vincent.huang1116@gmail.com

摘　要

在程式碼的混淆技術中，Obfuscator LLVM(以下簡稱 OLLVM)為最常被使用的開源混淆軟體。本文主要參考 OLLVM 論文 [6]以及原始碼 [5]，並進行常數層級混淆能力增強。選擇常數層級做增強，主要原因為，OLLVM 本身在指令層級與基本塊層級已經有相當高的完善度，並 OLLVM 的論文後段明確提出常數層級的混淆是 OLLVM 所不足的，也是 OLLVM 論文中希望發展的方向。

關鍵詞: OLLVM、混淆、常數

1. 前言

在 OLLVM 中，有三個混淆功能，分別為指令代換、控制流扁平化與虛假控制流，指令代換屬於指令層級的混淆，而控制流扁平化與虛假控制流屬於基本塊層級的混淆，本次實驗中將加入常數層級的混淆，觀察是否能提升混淆程度。

2. 背景知識簡述 [6][5]

2.1 OLLVM

OLLVM 是基於 LLVM 所設計出的混淆程式，其功能有指令代換、控制流扁平化與虛假控制流，可分別使用與一起使用，以下將簡單介紹各個混淆功能。

2.2 指令代換

功能是將指令以等價且較複雜的指令取代，以增加程式的複雜程度。

Operator	Equivalent Instruction Sequence
a = b + c	a = b - (-c) a = -(-b+ (-c)) a = b + r; a += c; a -= r a = b - r; a += c; a += r
a = b - c	a = b + (-c) a = b + r; a -= c; a -= r a = b - r; a -= c; a += r
a = b & c	a = (b^ !c) & b
a = b | c	a = (b&c) | (b^ c)
a = b ^ c	a = (!b&c) | (b&!c)

圖 1 指令代換

2.3 控制流扁平化

功能為將原本垂直的 if else 控制流轉為由 switch 分發的控制流，並時常搭配基本塊分離去做使用。

扁平化的控制流由一個分發器與數個可分發的基本塊節點(以下簡稱節點)所組成。分發器會將控制流轉向特定變數中所存的節點位置，而在節點的末端，將更改該變數中所存的節點位置，如此告訴分發器下一個節點位置。

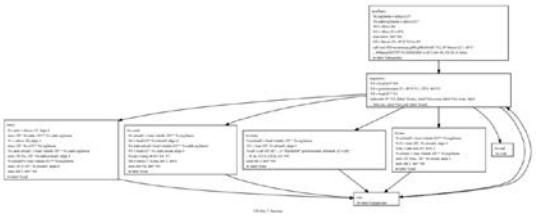

圖 2 控制流扁平化

2.4 虛假控制流

首先解釋不透明謂詞，不透明謂詞是一個表示式，其結果對程式開發者來說是已知的，但對於編譯器或靜態分析器而言是未知的，必須運行程式到該表示式才能確定結果。

例如: y > 10 || x * (x + 1) % 2 == 0，該式為恆真式，但基於某些原因，對於編譯器或靜態分析器而言是未知的。

虛假控制流即是使用了不透明詞謂作為 if 判斷式的依據，創造無法到達的基本塊，進而得到混淆的效果，如圖 4。

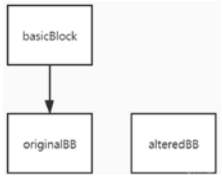

圖 3 加入虛假控制流前[3]

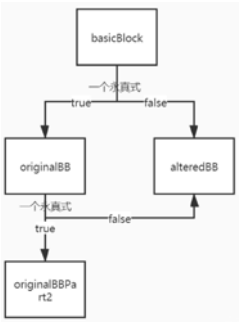

圖 4 加入虛假控制流後[3]

2.5 LLVM [1][2]

LLVM 是一個編譯器基礎設施，提供一系列的編譯器模組與工具鏈，特別之處為其將編譯器拆解為前端、IR 層與後端，本實驗與 OLLVM 即利用 IR 層的 pass 達成混淆效果。

3. 問題點

在控制流與指令混淆中，常數扮演著重要的腳色，用於控制流扁平化的下一節點位置，以及虛假控制流中的不透明謂詞，除此之外，在某些程式中，有些常數扮演著獲得重要資料的關鍵腳色(如圖 6)，若能將常數項加以混淆，有機會大幅增加程式的複雜度。

4. 常數分離混淆

4.1 原理

效果為，若二元運算式中有常數運算元，會將該常數做分離，通常會接著進行指令代換。

此舉能確保在靜態分析時，常數值不會被直接的觀察出，再依序搭配基本塊分離、控制流扁平化，如此將取得該常數值的線索打散在各個基本塊之中。

表 1 中，r、R1、R2 與 const 為常數，a、b 與 temp 為變數，此外，{r+const}是一個常數，該值已經被計算出，不是二元運算式，目的為方便了解該值為 r+const。

重要附註，在本實驗中，常數分離中皆使用 100 作為 r 值，以方便觀察常數分離的情形，實際使用將會是亂數值。

表 1 常數分離

混淆前	加入常數分離	再加入指令代換
a = b + const	a = b + {r+const} a -= r	a = b + R1; a += {r+const} a -= R1 a = a + R2; a -= r a -= R2
a = b - const	a = b – {r+const} a -= (-r)	A = b + R1; a -= {r+const} a -= R1 a = a + R2; a -= (-r) a -= R2
a = b ^ const	a = b ^ {r^const} a ^= r	a = (!b&{r^const}) | (b&!{r^const}) a = (!a&r) | (a&!r)

4.2 實作方式

環境使用 ubuntu18.04 搭配第 9 版的 OLLVM 與第 9 版的 LLVM，並將常數分離功能以 shared object file 的方式掛載使用。

圖 5 為部分功能虛擬碼，僅列出當常數在第二個運算元且利用加法做常數分離時之虛擬碼。

步驟為，利用 LLVM 的模組，遍歷基本塊中的所有指令，接著判斷該指令是否含有常數以及其他特定條件，若符合，首先，創造一個新的指令在原指令之前，而該指令的運算元為原指令的第一個運算元以及原指令之常數加上亂數 r，接著，修改原指令的第一運算元為新指令的結果，並修改原指令的第二運算元為亂數 r 的負值 。

```
FOR Basic_Block IN Function
    FOR Instruction IN Basic_Block
        I = Instruction;
        IF (I.isBinaryOperation() AND\
                I.getOperand(1).isConstant() AND\
                I.getOprator() == I.Add){
            // initial: a = b + const

            r = rand();
            r_const = I.getOperand(1) + r;
            newValue = createBinaryInstruction(I.Add,\
                I.operand(0), r_const);
            // to here: newValue = b + r_const
            //          a = b + const

            I.setOperand(0, newValue);
            I.setOperand(1, -r);
            // final: newValue = b + r_const
            //        a = newValue + (-r)
        }
```

圖 5 常數分離部分功能虛擬碼

6. 效果測試

6.1 測試方式

使用圖 6 的例子做分析，並使用 ghidra 軟體作為反組譯的工具，觀察常數的混淆狀況。

圖 6 中，若第一個輸入加 10 等於第二個輸入即可得到 flag。混淆該程式有兩個目標，目標一，反組譯時無法輕易的發現常數 10，目標二，反組譯時無法輕易的發現第 9 行不透明謂詞的存在。

```
#include<stdio.h>

int main(int argc, char** argv){
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        int c;
        c = a + 10;

        if (a > 0|| c*(c-1) % 2 == 0){
                if (c == b)
                        printf("real flag");
                else
                        printf("fake flag");
        }
        else{
                printf("impossible to here");
        }
}
```

圖 6 簡易 CTF 例子

6.2 測試結果

在反組譯未做混淆之目的檔(圖 7)中，於反組譯後可於第 14 行發現獲得 flag 的方式，以及於第 10 行發現不透明謂詞的存在。

```
undefined4 main(undefined8 param_1,long param_2)

{
  int iVar1;
  int iVar2;

  iVar1 = atoi(*(char **)(param_2 + 8));
  iVar2 = atoi(*(char **)(param_2 + 0x10));
  if ((iVar1 < 1) && (((iVar1 + 10) * (iVar1 + 9)) % 2 != 0)) {
    printf("impossible to here");
  }
  else {
    if (iVar1 + 10 == iVar2) {
      printf("real flag");
    }
    else {
      printf("fake flag");
    }
  }
  return 0;
}
```

圖 7 反組譯未做混淆之目的檔

在反組譯僅做指令代換之目的檔(圖 8)中，雖然代碼看起來與圖 7 相同，不過這是由於 ghidra 本身在反組譯分析時的指令優化所致，不過也顯示混淆的不足，可於圖 9 以及圖 10 中看出在組合語言的部分的確有指令代換，但還是可看出常數 10。

```
undefined4 main(undefined8 param_1,long param_2)

{
  int iVar1;
  int iVar2;

  iVar1 = atoi(*(char **)(param_2 + 8));
  iVar2 = atoi(*(char **)(param_2 + 0x10));
  if ((iVar1 < 1) && (((iVar1 + 10) * (iVar1 + 9)) % 2 != 0)) {
    printf("impossible to here");
  }
  else {
    if (iVar1 + 10 == iVar2) {
      printf("real flag");
    }
    else {
      printf("fake flag");
    }
  }
  return 0;
}
```

圖 8 反組譯僅做指令代換之目的檔

```
0040055f 83 ea 0a        SUB    EDX,0xa
00400562 29 d0           SUB    EAX,EDX
00400564 89 45 e4        MOV    dword ptr [RBP + local_24],EAX
```

圖 9 部分組合語言，可看出常數 10 (0xa)

```
00400594 8b 45 e4        MOV    EAX,dword ptr [RBP + local_24]
00400597 3b 45 e8        CMP    EAX,dword ptr [RBP + local_20]
```

圖 10 部分組合語言

在反組譯常數分離且指令代換後目的檔(圖 11)中，可發現先經過常數分離後，再進行指令代換可增加常數層級與指令層級的複雜度，除此之外，成功的讓常數 10 無法被觀察到，也讓第 12 行的不透明謂詞無法輕易的被觀察出。

```
undefined4 main(undefined8 param_1,long param_2)

{
  int iVar1;
  int iVar2;
  int iVar3;

  iVar2 = atoi(*(char **)(param_2 + 8));
  iVar3 = atoi(*(char **)(param_2 + 0x10));
  iVar1 = -(-(iVar2 + 0x6e) + 100);
  if ((iVar2 < 1) && (-(100 - (iVar1 * (100 - (-(iVar2 + 0x6e) + -1)) + 0xca)) % -100 != 0)) {
    printf("impossible to here");
  }
  else {
    if (iVar1 == iVar3) {
      printf("real flag");
    }
    else {
      printf("fake flag");
    }
  }
  return 0;
}
```

圖 11 反組譯常數分離且指令代換後目的檔

在圖 12 中演示了在依序經過常數分離、指定代換以及控制流扁平化之目的檔中，分析控制流的部分過程，以及觀察常數與指令之混淆情形。

方式為，首先，在基本塊 block1 中搜尋到"real flag"字串，接著，在該基本塊的上方得知該基本塊在扁平化中的跳轉節點值為 0x7c150acd，且節點值存在 local_48 中，進而查找到基本塊 block2 有對 local_48 附值 0x7c150acd 的動作，故 block2 為 block1 的上一個基本塊節點，依此方式反覆追蹤，於追蹤至 block4 時知道，若要得到 flag 必須使 local_44 == local_40，且於 block5 中可發現因常數分離與指令代換，被打散於各個基本塊中的不透明謂詞部分指令。因繼續追蹤下去會過於冗長，故觀察到此。

```
// block1
if (local_48 == 0x7c150acd) {
    printf("real flag");
    local_48 = 0x795eaf77;
}

// block2
if (local_48 == 0x5607ec54) {
    local_48 = 0x73bbb7c3;
    if ((local_d & 1) != 0) {
        local_48 = 0x7c150acd;
    }
}

// block3
if (local_48 == 0x36f310cf) {
    local_d = local_18 == local_14;
    local_48 = 0x5607ec54;
}

// block4
if (local_48 == 0x231a6de6) {
    local_18 = local_44;
    local_14 = local_40;
    local_48 = 0x36f310cf;

// block5
if (local_48 == 0x5235a934) {
    local_48 = 0x413f22ab;
    if (local_1c % -100 == 0) {
        local_48 = 0x231a6de6;
}
```

圖 12 反組譯依序經過常數分離、指定代換以及控制流扁平化之目的檔(部分基本塊)

7. 結論

在 OLLVM 上增加常數分離，可保護重要常數無法於反組譯時輕易地觀察出，且可讓不透明謂詞更加的隱蔽。此外，常數分離與 OLLVM 既有的功能搭配，可以讓重要指令的資訊與重要常數的部分碎片，散落於不同基本塊中，增加觀察程式邏輯的難度。

OLLVM 繼承了 LLVM 易擴充的優點，未來的擴充方向可能為字串或者是在常數附值時直接進行混淆(本文僅對二元運算式中常數做混淆)。

OLLVM 與本文皆在 IR 層做混淆的動作，後端形成目的檔階段的混淆也是比較不足的，可能方向為利用後端發展暫存器層級與組合語言層級的混淆。

參考文獻

[1] https://en.wikipedia.org/wiki/LLVM, 2021/06

[2] https://llvm.org/, 2021/06

[3] https://toutiao.io/posts/euakfy0/preview, 圖片, 2021/06

[4] OLLVM 筆記控制流源碼學習, 2021/06

[5] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin https://github.com/obfuscator-llvm/obfuscator, 2015/10

[6] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin "Obfuscator-LLVM -- Software Protection for the Masses", 2015/05