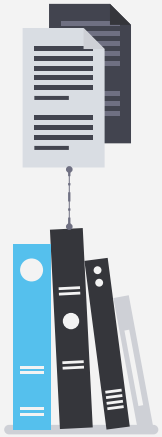
PackGenome: Automatically Generating Robust YARA Rules for Accurate Malware Packer Detection

Li, Shijia, et al. Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 2023.

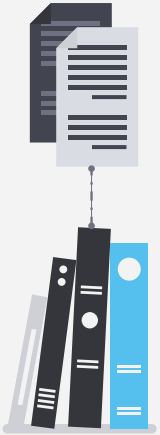
Table of contents



01

Introduction

Problem Background



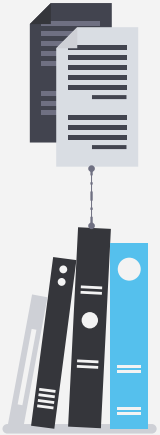
程式中通常會有明顯資訊可以進行分析，而防毒軟體可以使用這些資訊來識別程式。

防毒軟體通常會使用 YARA rule 來偵測惡意軟體，但通常惡意軟體會使用「殼 Pack」來隱藏自己的資訊，不僅造成 YARA rule 偵測的困難，也讓分析人員難以撰寫 YARA rule。

若是能辨識出加殼器就可以更有效率的進行分析，甚至是自動化脫殼。

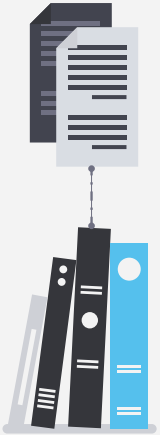
Research Motivation

基於現有檢測方法的侷限性和加殼技術的不斷演變，提出一種自動化、準確且高效的方式來偵測加殼器，減少分析人員的負擔，並提高檢測的有效性和可靠度。
通過自動生成基於加殼器基因的 YARA 規則，解決傳統手動方法的不足，並提供一個能夠快速適應和應對新威脅的檢測工具。



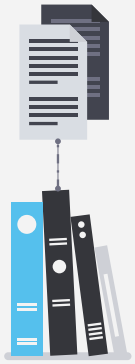
Research Objectives

1. 自動生成 YARA rule，減少人工編寫的工作量。
2. 提高檢測準確度，降低誤報和漏報。
3. 確保執行效率，不會因為大規模樣本造成資源消耗。
4. 提供通用解決方案，能應用在不同加殼技術和平台上。

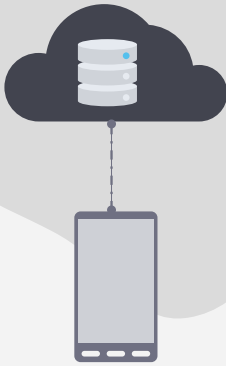


02

Background



Malware Packing Techniques



Packer

目的：壓縮
常見工具：UPX

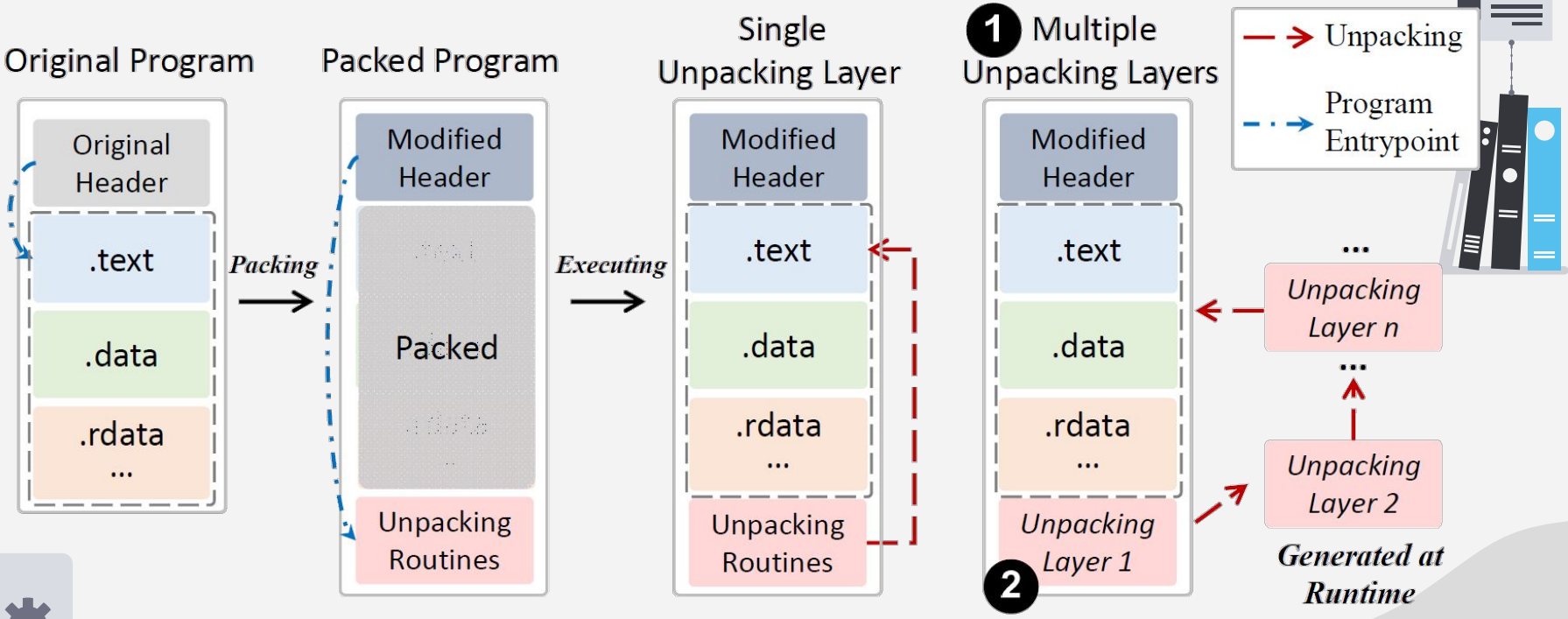
Crypter

目的：加密/混淆
常見工具：ZProtect

Protector

目的：防止破解或修改
常見工具
：VMProtect、
WProtect

Malware Packing Techniques



Existing Detection Methods

靜態分析

- Signature-based
- Entropy-based

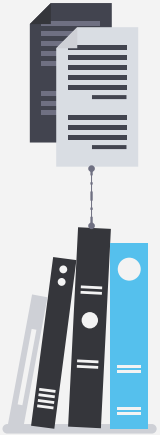
對二進位檔進行分析，辨識 signature
優點：不執行文件
缺點：容易被加殼技術躲避、
signature 維護困難

動態分析

- Behavioral analysis
- Heuristic analysis

執行文件並監控行為
優點：可以觀察到行為特徵
缺點：資源消耗高、可能被反分析
技術躲避

Signature-based Packer Detection



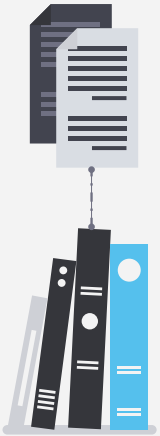
YARA 是 VirusTotal 的開源專案，旨在幫助惡意軟體研究人員識別和分類惡意軟體樣本。根據文字或二進位 pattern 建立惡意軟體的描述（Rule）。

```
rule silent_banker : banker
{
   meta:
      description = "This is just an example"
      threat_level = 3
      in_the_wild = true

   strings:
      $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
      $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
      $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

   condition:
      $a or $b or $c
}
```

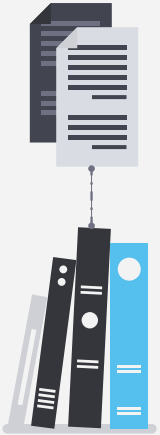
Signature-based Packer Detection



1. text strings（$a）
2. text strings with regular expression
3. hexadecimal strings（$b）
4. hexadecimal string with special constructions（wildcards（?? in $c）,jumps（[4] in $d）,alternatives（57|87） in $d）

```
1 rule UPX {
2 strings:
3   $a = "UPX"
4   $b = {60 E8 00 00 00 00 58 83 E8 3D}
5   $c = {EB ?? ?? ?? ?? ?? 8A 06 46 88 07 47 01 DB 75 07 8B
          1E 83 EE FC 11 DB}
6   $d = {60 E8 [4] 58 83 E8 3D 50 8D B8 [4] (57|87) 8D B0}
```

Signature-based Packer Detection



1. Address-based

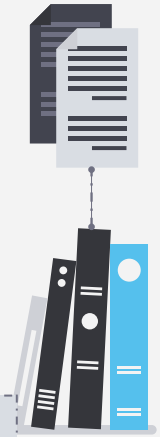
   只在特定位址搜尋，如 $b 和 $d，只在 PE 檔的進入點進行搜尋。
   超過 90% 的加殼器偵測規則只在特定位址搜尋，但可以透過更改進入點的指令來繞過。

2. Full-binary matching

   為了增加穩健性，可以搜尋整個二進位檔，如 $a 和 $c，但偵測器只配對位元組的格式而不是 instruction encoding，因此有問題的整體二進制規則會導致高誤判。

```
7  condition:
8    $a and ($b at pe.entry_point or $d at pe.entry_point)
           and $c
9  }
```

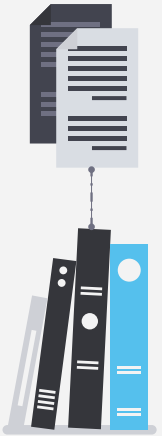
Challenges Of Generating Packer Detection Rules



從公開可用的 YARA 和 DIE 中收集加殼器偵測規則並除重。
Packer 表示支援的加殼器。
Meta 表示從 PE header 資訊和文字字串建立的規則。
SC Bytes 表示使用特殊結構的規則。

Sources	#Packer	Search Scope		Search Content			#Total
		Address-Based	Full-Binary	Meta	Bytes	SC Bytes	
YARA[45-50]	492	9582	667	31	6672	3549	10249
DIE	324	1074	30	201	321	650	1104

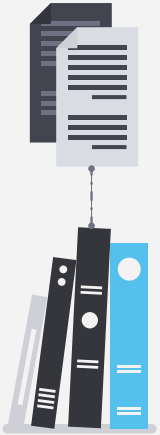
Challenges Of Generating Packer Detection Rules



1. 缺乏生成規則的準則
   分析人員依靠經驗來開發規則，DIE 嚴重依賴程式的 Meta 資訊，這可以靠修改 unique strings 來繞過。
   99.3% 的 YARA rule 只考慮加殼程式進入點的位元組，這些規則可以透過修改進入點指令來繞過。
   為了減少誤判，會增加位元組規則長度，80% 的規則超過 25 個位元組。

2. 加殼器的指令經常與無關的指令配對
   因為 signature-based 偵測在配對位元組時不會考慮到 instruction format，這會導致 mismatch。
   signature-based 偵測可以支援靜態反組譯指令的配對，但會增加開銷。

instruction format



有 zero, one, two, and three-address instructions。
每種類型在程式碼大小、執行時間和彈性方面都有自己的優缺點。

opcode	operand/address of operand	mode

One address instruction

opcode	Destination address	Source address	mode

Two address instruction

Challenges Of Generating Packer Detection Rules

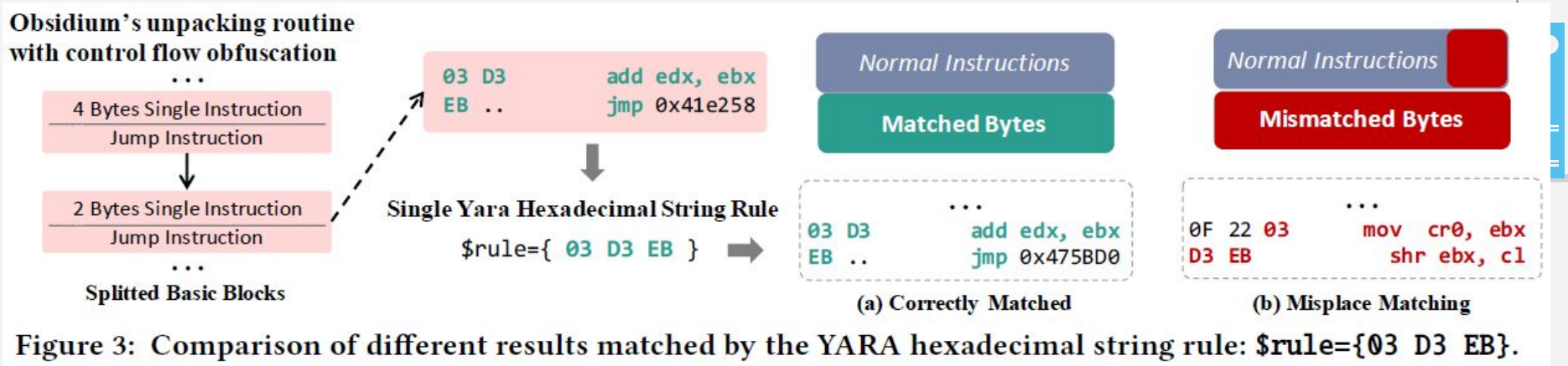
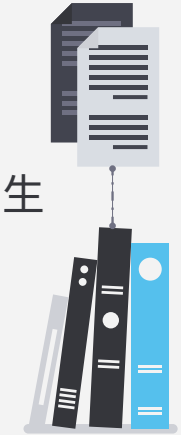


Figure 3: Comparison of different results matched by the YARA hexadecimal string rule: `$rule={03 D3 EB}`.

Challenges Of Generating Packer Detection Rules



緩解這些問題的一個方向是自動生成加殼程式的 signature rule，但現有的自動規則生成器主要關注的是為惡意軟體的 payload 自動生成 signature，而不是為加殼器生成。

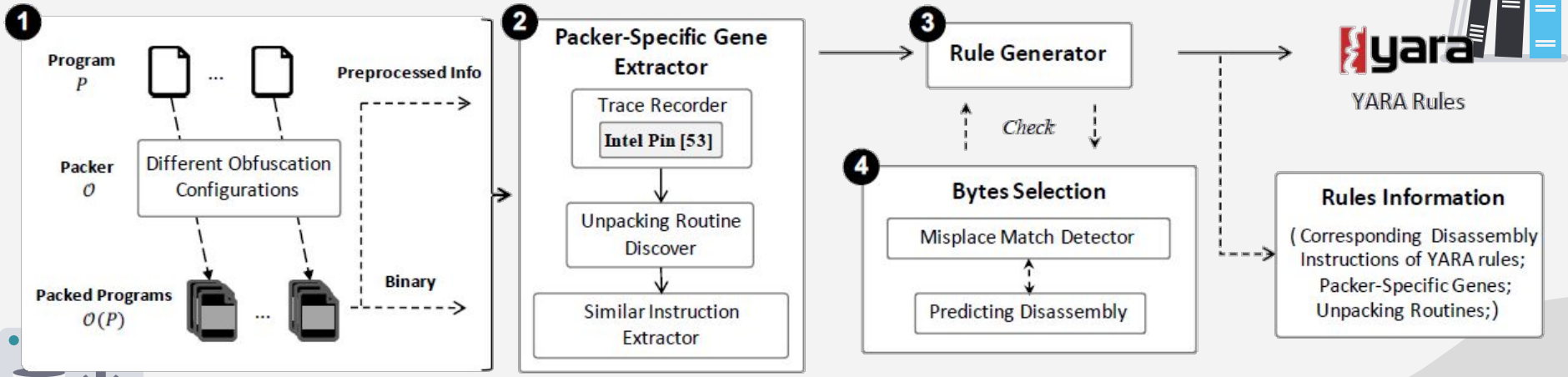
- YaraGenerator
  基於惡意軟體家族之間共用的常見文本特徵（如 string）來生成規則。
- yabin
  使用 function prologues 的固定長度位元組來生成。
- yarGen
  從預先建立的白名單資料庫中過濾出突出文字和十六進位字串來生成規則。
- AutoYara
  結合 heuristic 和 biclustering 演算法，從有限的樣本中找到頻繁出現的 large N-gram 位元組來生成高品質規則。

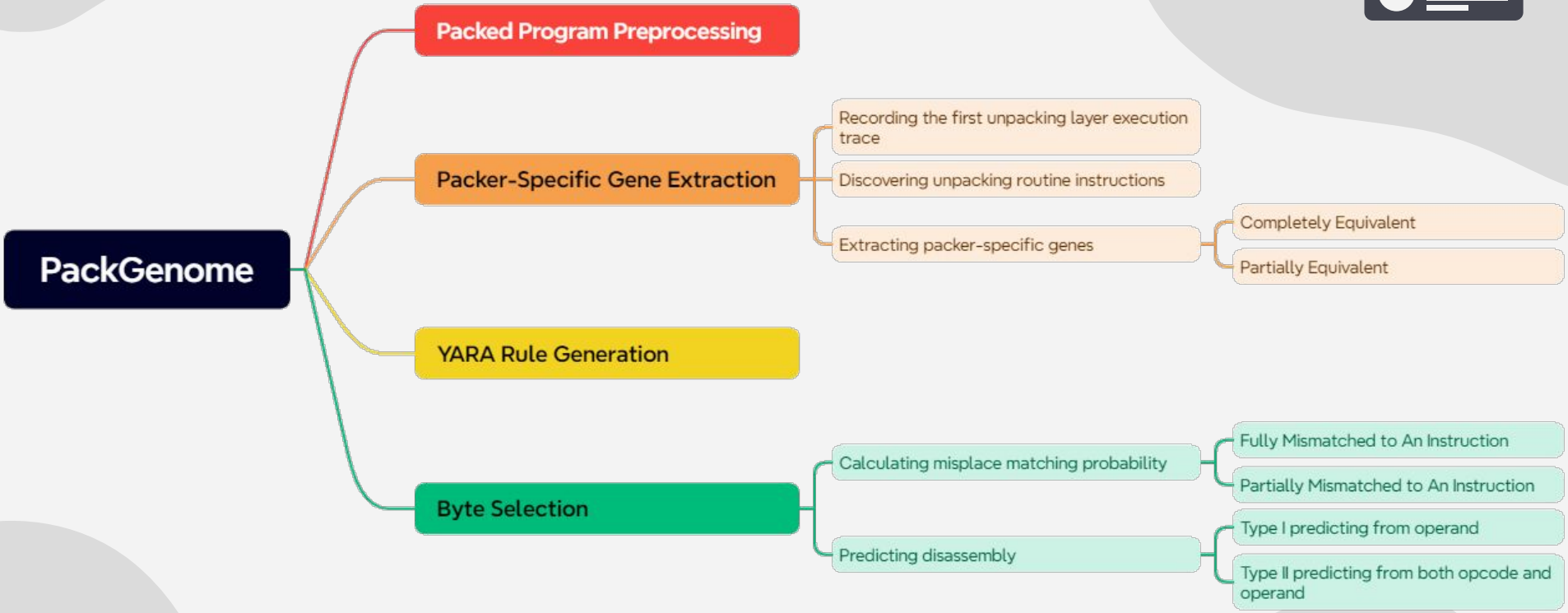
03

Method

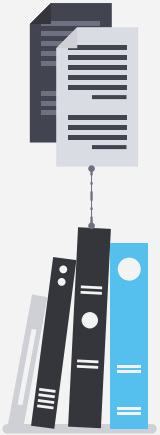
Overview

1. Packed Program Preprocessing
2. Packer-Specific Gene Extraction
3. YARA Rule Generation
4. Byte Selection




PackGenome
Packed Program Preprocessing
Packer-Specific Gene Extraction
Recording the first unpacking layer execution trace
Discovering unpacking routine instructions
Extracting packer-specific genes
Completely Equivalent
Partially Equivalent
YARA Rule Generation
Byte Selection
Calculating misplace matching probability
Fully Mismatched to An Instruction
Partially Mismatched to An Instruction
Predicting disassembly
Type I predicting from operand
Type II predicting from both opcode and operand

Packed Program Preprocessing



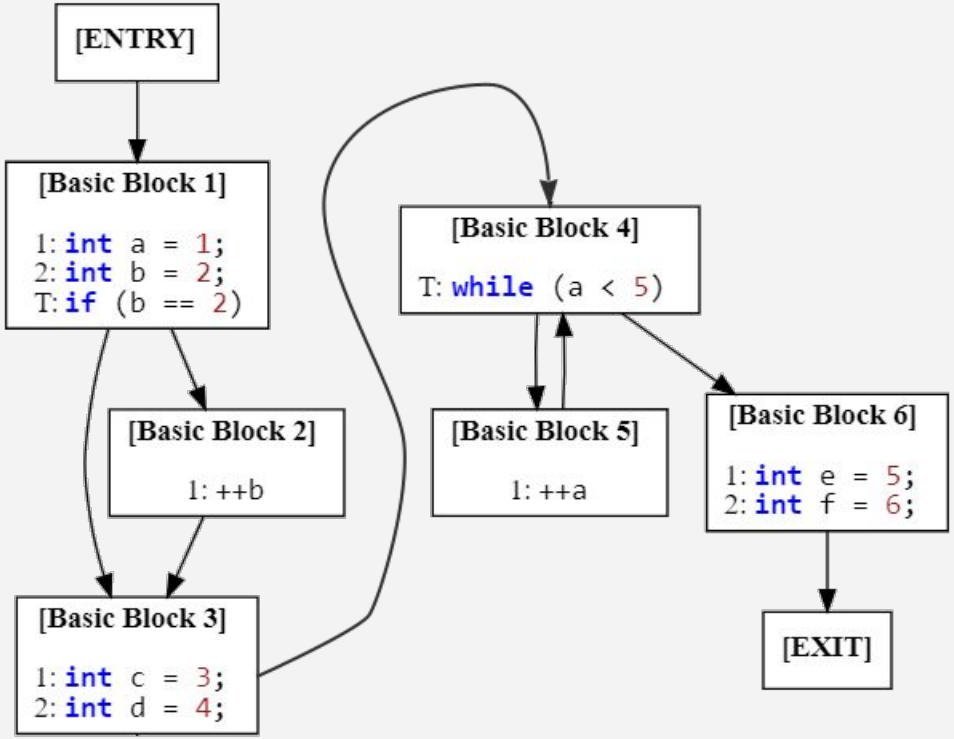
準備加殼程式並從中收集必要資訊，來幫助提取 packer-specific gene。

1. 製作大量的多樣化樣本庫，遍例加殼器的所有配置組合來涵蓋不同的脫殼例程。
2. 收集加殼程式的 section 資訊（如 name 和 address），以幫助在執行時發現靜態可見的脫殼例程指令。
3. 透過收集的資料，監控加殼程式中被寫入並被靜態可見的脫殼例程指令執行的區域，並為這些指令分配標籤。

脫殼例程的明顯特徵是必須將脫殼後的原始內容放回原始的虛擬位址上，如執行時脫殼指令必須放在＂.text＂ section 的虛擬位址。

basic block

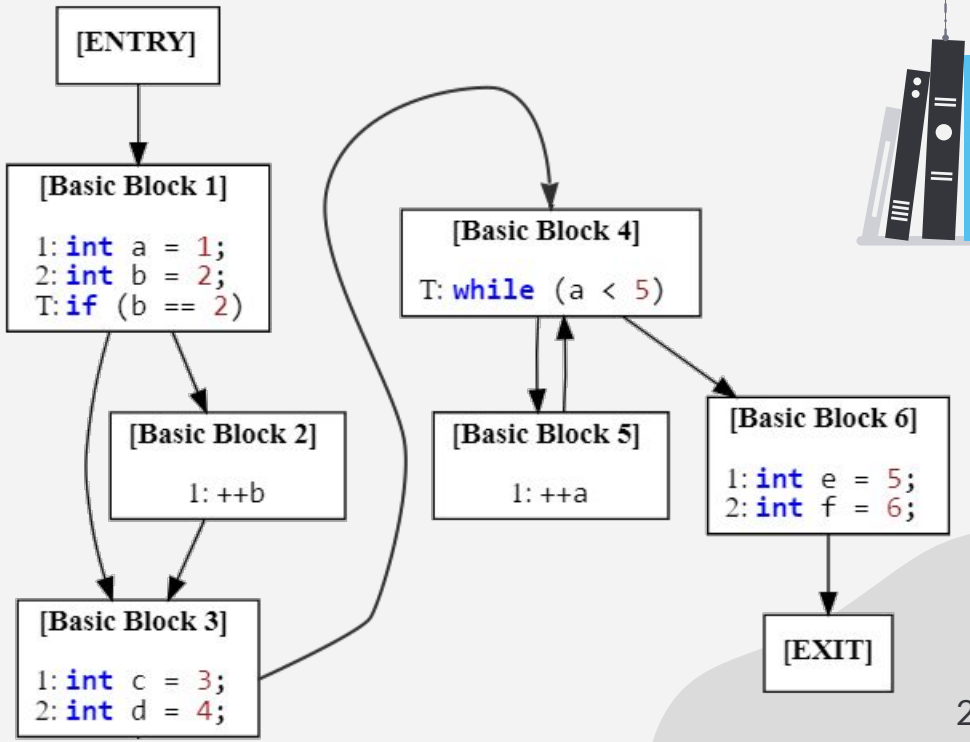
一段不包含分支和跳轉的連續指令序列，只有一個入口和出口



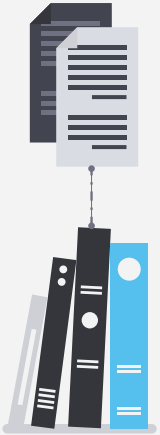
control flow graph

可達性分析：確定哪些 basic block 是可到的
路徑分析：列舉所有可能路徑
循環分析：識別循環結構和巢狀關係

由 basic block 組成的有向圖



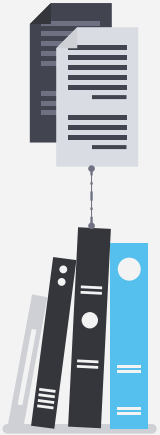
control flow analysis



用來了解和表示程式的控制流，即程式在執行過程中可能的執行路徑，主要是生成和分析 CFG，幫助理解程式的執行邏輯、潛在錯誤和最佳化。

惡意軟體中透過分析 control flow 可以識別惡意行為和隱藏程式。

Packer-specific Gene Extraction

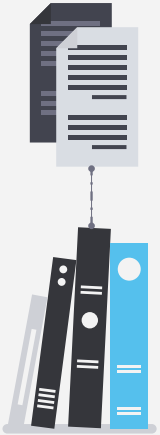


從加殼程式中提取獨特的指令序列（Packer-specific Gene），用於識別加殼技術。

記錄第一層脫殼的 executed trace 並給指令分配標籤，根據 control flow 資訊和執行次數在 basic block 之間傳播標籤，最後從脫殼例程中重複使用的相似指令中提取出 packer-specific genes。

1. Recording the first unpacking layer execution trace
2. Discovering unpacking routine instructions
3. Extracting packer-specific genes

Recording The First Unpacking Layer Execution Trace



捕捉加殼程式在第一層脫殼時的行為和特徵。

使用 PIN 來記錄靜態可見的脫殼例程指令在執行時的資訊（記憶體位址、指令長度、basic block 的執行次數、basic block 的指令、標籤），根據指令執行時的行為分配標籤。

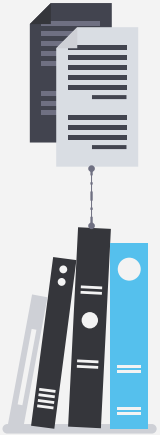
若指令 I 將脫殼後的指令 I' 寫到原始程式並且 I' 在程式執行時被執行，則將 I 分配標籤。

為了應對 anti-instrucmentation，將 PIN 和 ARANCINO 整合。

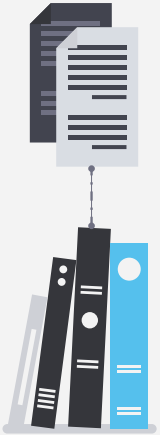
written then executed

1. 寫入：將程式碼寫入記憶體
2. 執行：執行寫入的程式

惡意軟體將加殼部分在運行時進行脫殼並執行。



Discovering Unpacking Routine Instructions

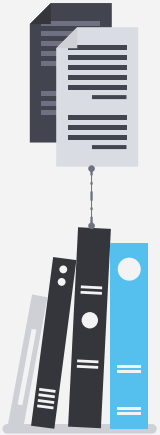


識別脫殼行為。

PIN 只分配標籤給直接寫入到監控地址的指令，如 written then executed 區域。忽略脫殼例程指令中的其他部分，如只解碼資料的部分。

為了找到完整的脫殼例程指令，通過 control flow analysis 將指令的標籤傳播到相關的 basic block B。標籤只在執行次數相近的 B 之間傳播，以避免傳播到進入點指令，進入點指令容易被加殼器修改。

Discovering Unpacking Routine Instructions



用 $REN(B_i)$ 找到標籤多的 B_i

$$REN(B_i) = \frac{N_{B_i}}{\sum_{j=1}^{m} N_{B_i}}$$

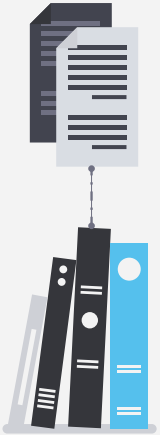
N_B basic block 的執行次數

$REN(B_i)$ basic block B_i 的相關執行次數

m basic block B 的總數

B_i 單個 trace 中記錄的集合$\{B_1, ..., B_m\}$的一個

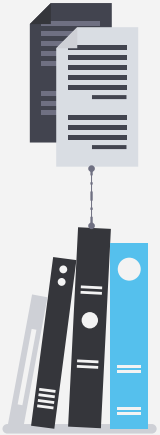
Discovering Unpacking Routine Instructions



將動態分析中記錄的指令位元組與靜態分析提取的指令位元組進行比較，如果不同則為自修改指令，將其從 B 中去除。

剩下的靜態可見的脫殼例程指令是作為 packer-specific genes 的候選項。

Extracting Packer-specific Genes

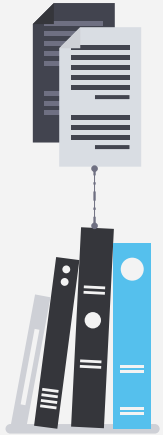


從重複使用的脫殼例程中找到相似指令。

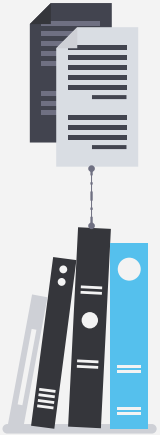
計算多個相同殼的加殼程式的 basic block $\mathcal{B}$ 的相似度，然後使用 $\mathcal{B}$ 的相似度來選擇 packer-specific genes，並為規則生成器準備在語法上相似的指令的相似性資訊（不同位元組的偏移量）。

Extracting Packer-specific Genes

$$P_a \quad \{ß_{a1}, \ldots, ß_{an}\}$$
$$P_b \quad \{ß_{b1}, \ldots, ß_{bn}\}$$



Extracting Packer-specific Genes



為了發現重複使用的 $\mathcal{B}$，使用以下兩個步驟比較不同 $\mathcal{B}$ 的相似性。

1. Bytes

   直接比較 $\mathcal{B}_{ai}$ 和 $\mathcal{B}_{bj}$ 的位元組。

   若位元組相同，跳過 Slice 比較，
   否則在 Slice level 中比較。

2. Slice

program slice

Static Slicing

不考慮程式執行時的輸入，通過分析程式中的 control flow 和 data flow 來識別與特定變數或計算相關的所有指令。

Dynamic Slicing

依賴程式的執行時輸入，關注特定輸入下程式的執行路徑，提取與輸入相關的計算片段。

Forward Slicing

往後追蹤，找到受它影響的程式碼。

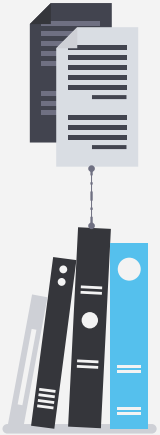
Backward Slicing

往前追蹤，找到影響它的程式碼。

Slice

為了對抗混淆，比較從 $\mathcal{B}$ 提取出的 slice。

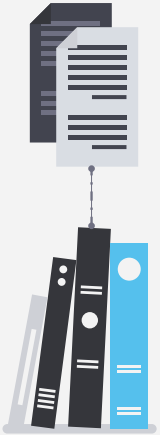
先通過從 $\mathcal{B}$ 的輸出開始的 backward slicing 來將 $\mathcal{B}$ 分解成 slice S。
然後計算 slice 之間的統計相似性，並將 slice 的相似度提升到 $\mathcal{B}$ 的相似性。



operand（運算元）

是指令中用於指定操作對象或操作參數的部分。
根據指令的具體類型和操作需求，可以是 Immediate Operand、Register Operand、Memory Operand。

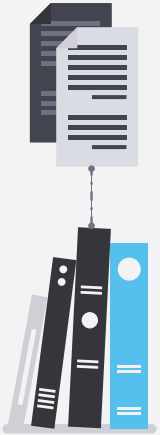
MOV EAX, 5 EAX,5 都是 Operand。



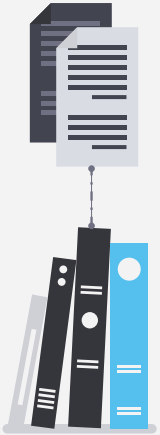
output operand

一條指令中，作為輸出結果的 operand。這些 operand 通常是指令執行後結果被存放的目標位置，例如暫存器或記憶體位址。

ADD EAX, EBX　EAX 是 output operand。



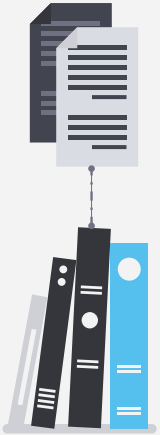
opcode



Operation code 是指令中用於識別操作的部分，表示特定的操作或指令類型，為二進位數字或位元組。通常位於指令開頭，不同的 opcode 對應不同指令。

" **add eax,0x41** " opcode 0x81

mnemonic（助記符）



組合語言中用來表示指令操作的簡短、易記的字母或字母組合。
每個 mnemonic 對應於一個或多個 opcode，便於人類理解和編寫低階語言。

opcode 和 mnemonic 為多對多關係。

" **add eax,0x41** " 和 " **or eax,0x41** " 共享相同 opcode 0x81

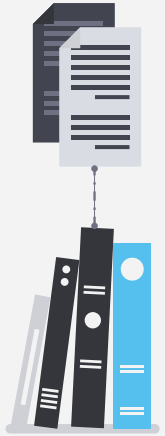
Slice

計算 S_a S_b 的相似度。

$$SimSlice\left(S_a, S_b\right) = \sum_{k=1,l=1 I_k \in S_a, I_l \in S_b}^{n} SimIns\left(I_k, I_l\right) / n$$

S_a S_b 有相同的 output operands
n S_a S_b 的最大指令數

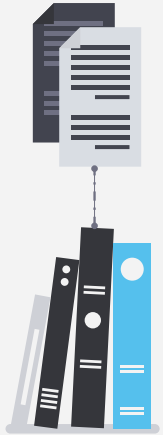
$SimIns(I_k, I_l)$被用於比較指令的相似性，當I_k I_l 的 instruction format（mnemonic 和 operand type 如 REG）相同時為 1，否則為 0。



example

$$S_a = \{I_1, I_2\} \quad S_b = \{I_3, I_4\}$$

$$SimSlice\,(S_a, S_b) =$$
$$\{SimIns\,(I_1, I_3) + SimIns\,(I_1, I_4) +$$
$$SimIns\,(I_2, I_3) + SimIns\,(I_2, I_4)\}/4$$

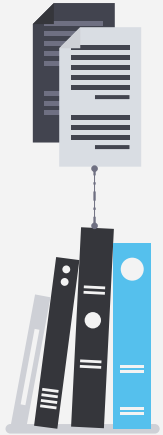


Slice

計算 $\mathcal{B}_a$ $\mathcal{B}_b$ 的相似度。

$$SimBS\left(ß_a, ß_b\right) = \sum_{k=1,l=1,S_i \in ß_a, S_j \in ß_b}^{n} SimSlice\left(S_i, S_j\right)/n$$

n $\mathcal{B}_a$ $\mathcal{B}_b$ 的最大 slice 數
如果 $\mathcal{B}_a$ $\mathcal{B}_b$ 的每個 slice group 在語法上相似，那 $\mathcal{B}_a$ $\mathcal{B}_b$ 在 slice level 上為高度相似。



Extracting Packer-specific Genes

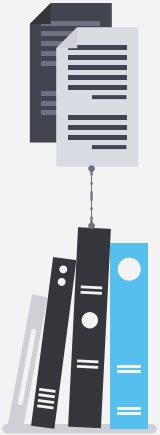
Completely Equivalent

若 $\mathcal{B}_{ai}$ $\mathcal{B}_{bj}$ 的位元組相同，則視為 完全相等。
如 UPX 在每個加殼程式中重複使用相同的脫殼例程指令，這些完全相等的位元組可以直接用於生成偵測加殼器的 YARA rule。

Partially Equivalent

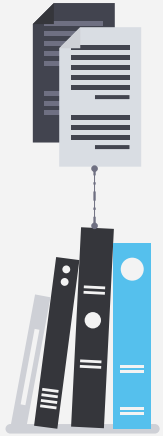
當加殼器採用混淆來保護脫殼例程指令時，可能會發現 $\mathcal{B}_{ai}$ 只有部分相等於 $\mathcal{B}_{bj}$，這表示他們具有相同 slice 但不同位元組。

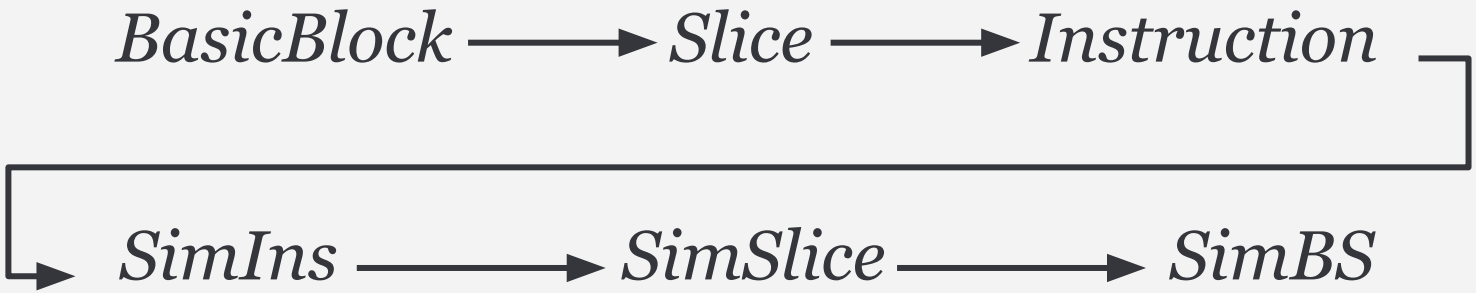
Partially Equivalent



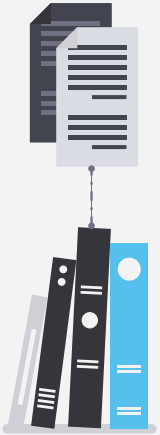
如從 Enigma 加殼的程式中提取出的兩個 slice " mov ecx,0x579;dec ecx; " 和 " mov ecx,0x586;dex ecx; "，由於 mov 指令的不同 operand value，因此具有不同的位元組。

透過 $SimBS(B_a, B_b)$ 計算，具有較高相似性的 B 更適合作為 packer -specific gene 的候選項。




BasicBlock
Slice
Instruction
SimIns
SimSlice
SimBS

YARA Rule Generation

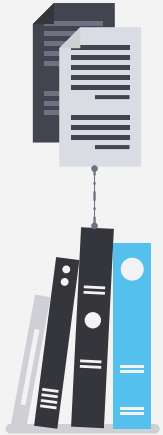


基於相似性資訊從每個 pakcer-specific genes 的 basic block 中生成十六進位字串規則（hexadecimal string rule,HSR）。

若 packer-specific gene 的位元組完全相同，直接將這些位元組轉換成 HSR。
否則從部分相同的位元組中找出相異部分，用詳細的特殊構造來取代並建立 HSR。

HSR 最小長度為 2。

mismatch

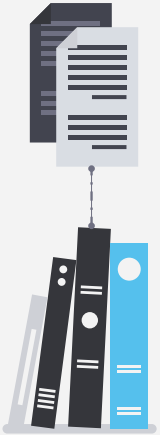


嘗試在特定位置配對特定位元組序列或指令 pattern 時，預期的內容與實際的內容不一致。

預期的位元組序列：8B 03 83 C0
MOV EAX, [EBX] -> 8B 03
ADD EAX, 1 -> 83 C0 01

實際位元組序列：
MOV EAX, [ECX] -> 8B 03
ADD EAX, 2 -> 83 C0 02

misplace match

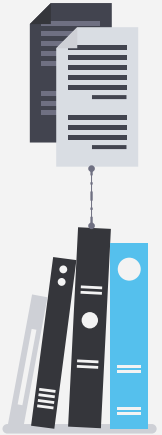


配對過程中，識別出預期的內容，但配對的位置與預期不一致。
通常是由於程式混淆、花指令等技術導致。

預期的位元組序列：03 D3 EB
ADD EDX, EBX -> 03 D3
JMP 0x475BD0 -> EB ..

實際位元組序列：
MOV CR0, EBX -> 0F 22 03
SHR EBX, C1 -> D3 EB

Byte Selection

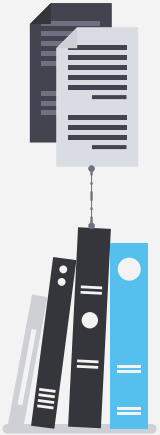


根據計算 YARA rule 的 misplace match 機率來選擇 YARA rule。

計算 YARA rule 中的每個 HSR 的 misplace match 機率 P_{HSR}，使用 predicting disassembly 將 mismatch 機率轉換成可能 mismatch 的指令發生的機率。
然後將 P_{HSR} 相乘，計算 YARA rule 的 mismatch 機率 $\mathscr{P}_{rule}$，並過濾出 mismatch 機率高的規則。

1. Calculating misplace matching probability
2. Predicting disassembly

Byte Selection



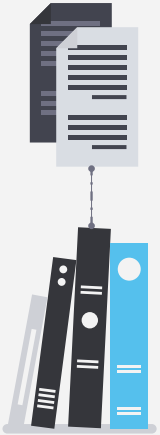
Packer_v1 的 misplace match 機率是 $\rho_{Packer_v1} = P_a * P_b * P_c$。
Packer_v2 的 misplace match 機率較高。

```
rule Packer_v1 {
strings:
  $a = {a4 eb} //P_a = 0.7
  $b = {21 41 3c e8 74} //P_b = 0.5
  $c = {8b 96 8c 00 00 00 8b c8 c1 e9 10 33 db 8a 1c 11
        8b d3 eb} //P_c = 0
condition:
  $a and $b and $c
}
rule Packer_v2 {
strings:
  $a = {a4 eb} //P_a = 0.7
  $b = {21 41 3c e8 74} //P_b = 0.5
condition:
  $a and $b
}
```


Byte Selection
Calculating misplace matching probability
Fully Mismatched to An Instruction
Partially Mismatched to An Instruction
Predicting disassembly
Type I predicting from operand
Type II predicting from both opcode and operand

Calculating misplace matching probability

評估某些指令或位元組序列出現在非預期位置的機率。

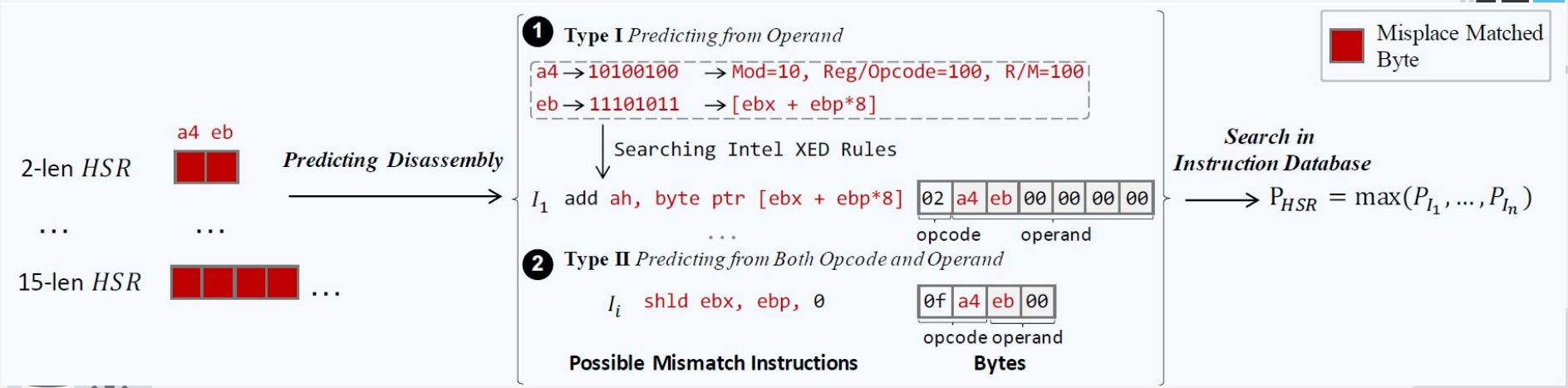


根據 HSR 的 mismatch 類型，可以用兩種方式計算 HSR 的 misplace match 機率 P_{HSR}：

1. Fully Mismatched to An Instruction
2. Partially Mismatched to An Instruction

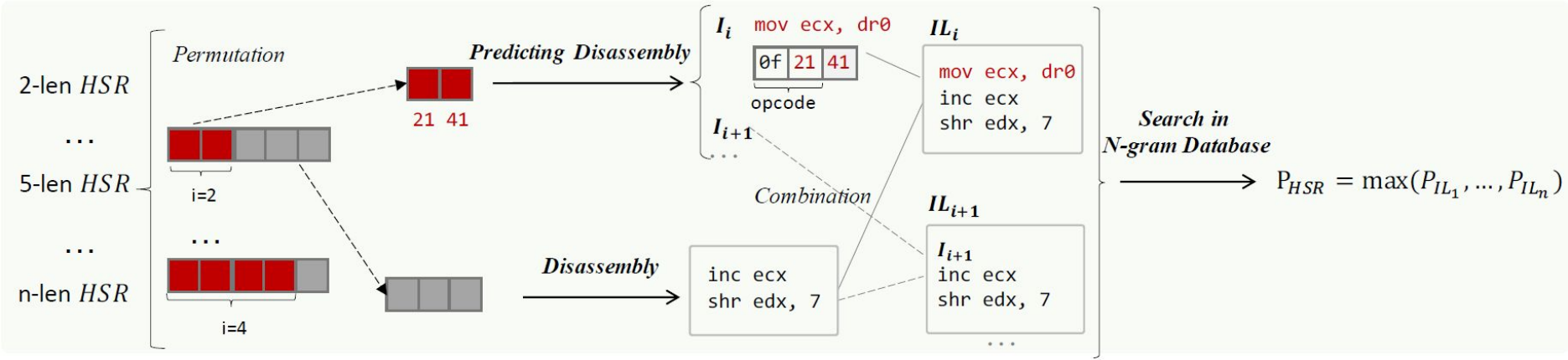
Fully Mismatched to An Instruction

P_{HSR} mismatch 的指令 I 發生在現實的機率
I_i 可能 mismatch 的指令
P_{li} 發生機率

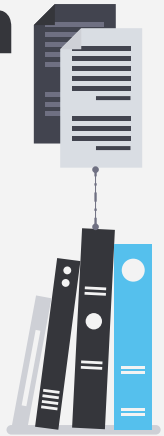


Partially Mismatched to An Instruction

P_{HSR} 現實程式中會出現的所有可能 mismatch 的指令序列 IL 的出現機率
IL_i 可能 mismatch 的指令
P_{lLi} 發生機率



Partially Mismatched to An Instruction

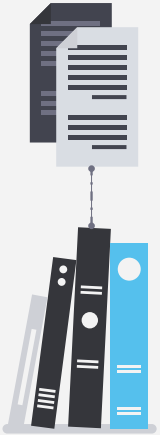


不同於其他 N-gram based 技術只提取指令的 opcode，使用四個元件來表示一條指令。

1. prefix
   可選部分，用於指定指令的額外屬性或條件
2. opcode
3. mnemonic
4. format of operands

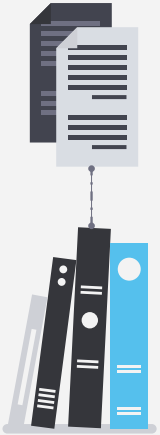
" **add eax,0x41** " 和 " **or eax,0x41** " 共享相同 opcode 0x81

examples



```
1  rule Packer_v1 {
2  strings:
3    $a = {a4 eb} //P_a = 0.7
4    $b = {21 41 3c e8 74} //P_b = 0.5
5    $c = {8b 96 8c 00 00 00 8b c8 c1 e9 10 33 db 8a 1c 11
           8b d3 eb} //P_c = 0
6  condition:
7    $a and $b and $c
8  }
9  rule Packer_v2 {
10 strings:
11   $a = {a4 eb} //P_a = 0.7
12   $b = {21 41 3c e8 74} //P_b = 0.5
13 condition:
14   $a and $b
15 }
```

Predicting disassembly

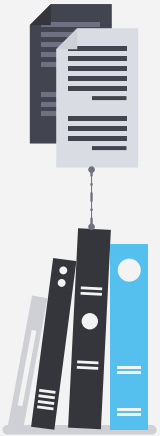


有效找到每個可能被 HSR fully misplace match 的指令。

暴力遍歷每個可能被 HSR mismatch 的位元組組合：256^{15} 種位元組組合

x86/x64 指令的最大長度為 15，且每個位元組的值為 0～255。
被最短 HSR fully misplace match 的指令需要遍歷 256^{15-2}（$2*10^{31}$）個位元組組合。

Predicting disassembly



輸入 HSR，搜尋能容納 mismatch HSR 全部位元組的 Intel XED 規則。
為了找到符合條件的 XED 規則，根據 XED 規則的編碼文法將 HSR 轉換成可搜尋格式。

a4 → 10100100 → Mod=10, Reg/Opcode=100, R/M=100

eb → 11101011 → [ebx + ebp*8]

Searching Intel XED Rules

I_1 add ah, byte ptr [ebx + ebp*8]

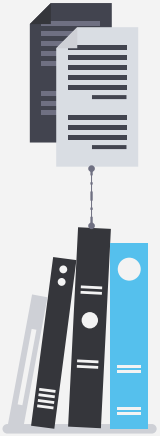
02	a4	eb	00	00	00	00

opcode operand

...

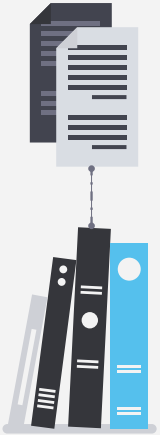
Predicting disassembly

1. Type I predicting from operand
2. Type II predicting from both opcode and operand



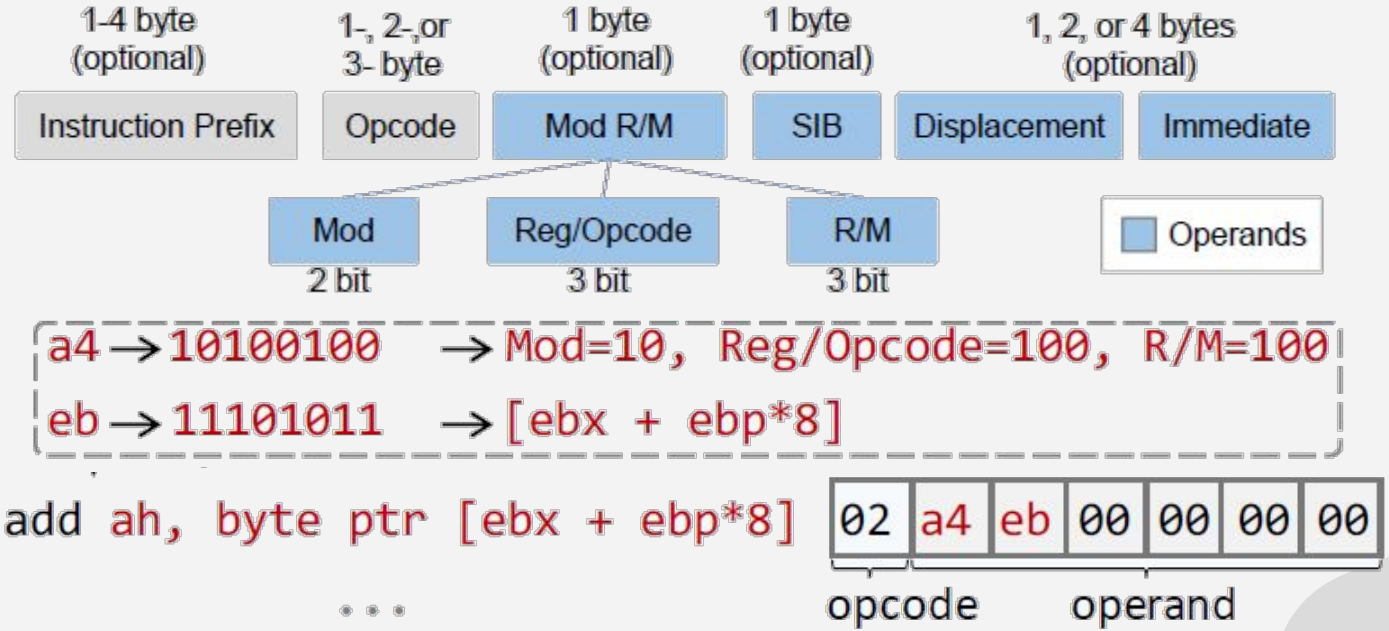
Predicting from opcode（and prefix）

由於 HSR 無法滿足任何 opcode 和 prefix 的組合，因此此類型不需要考慮。
只需處理後面兩個類型。



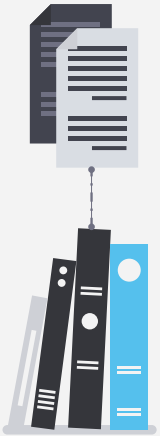
Type I Predicting from operand

將 HSR 轉換成 XED 規則的 operand 編碼格式，並收集與轉換成的 HSR 有相同 operand 編碼格式的 XED 規則。

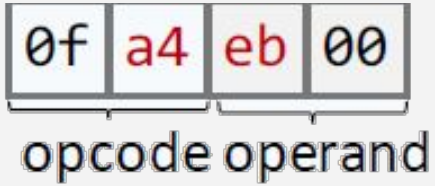


Type II Predicting from both opcode and operand

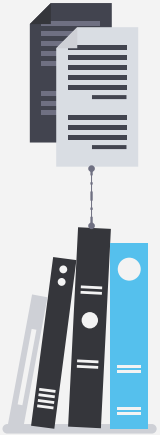
從 opcode 和 prefix 的組合中搜尋 HSR 的前幾個位元組，並收集符合條件的 XED 規則。
然後將 HSR 的剩餘位元組視為 Type I，並從先前收集的 XED 規則中搜尋符合條件的規則。



```
shld ebx, ebp, 0
```



Algorithm



Input：HSR
Result：INL（List of misplace matched instruction）

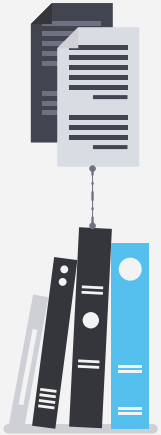
B2Opcode(HSR)：將 HSR 轉換成與 XED 規則的 opcode 相同的位元組格式
B2Operand(HSR)：根據 XED 規則的語法將 HSR 轉換為 operand
InsGen(rule)：從 XED rules 中生成指令
OPC[opcode]：滿足 opcode 的 XED 規則
OPE[operand]：滿足 operand 的 XED 規則
INL={}

Algorithm

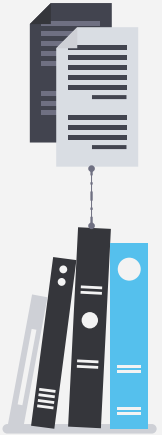
```
if the length of HSR > 15:
        return false

//Type I
if B2Operand(HSR) ≠ {}:
        for code in B2Operand(HSR):
                INL += InsGen(OPE[code])

//Type II
for i in range(len(HSR)):
        sopcode = B2Opcode(HSR[:i])
        if sopcode ≠ {}:
                soperand =  B2Operand(HSR[i:])
                if soperand ≠ {}:
                        INL += InsGen(OPC[sopcode],OPE[soperand])
```



Predicting disassembly



收集到符合條件的 XED 規則後，predicting disassembly 從收集的規則中合成出可能 mismatch 的指令。

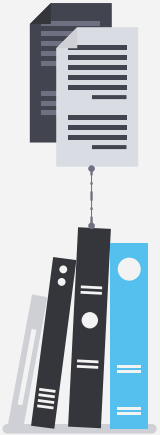
HSR " {a4 eb} "
Type I 9055 rules
Type II 1 rules

計算每條指令的發生機率，最大的發生機率 $P_{HSR} = 0.7$ ，這表示這條 HSR 容易 mismatch，建立 YARA rule 時，應該將其與 misplace match 機率低的 HSR 組合。

04

Evaluation

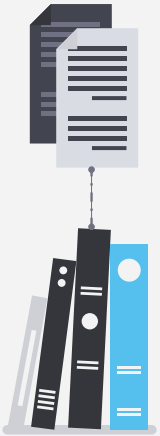
Evalution



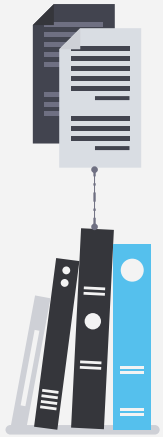
1. PackGenome 是否能有效的生成不同類型的加殼器的偵測規則？
2. 與人工編寫的規則和其他自動化工具相比，PackGenome 的規則準確率和效率如何？
3. PackGenome 生成的規則在偵測加殼樣本的可擴展性如何？
4. PackGenome 生成的規則在偵測現實程式時的性能如何？

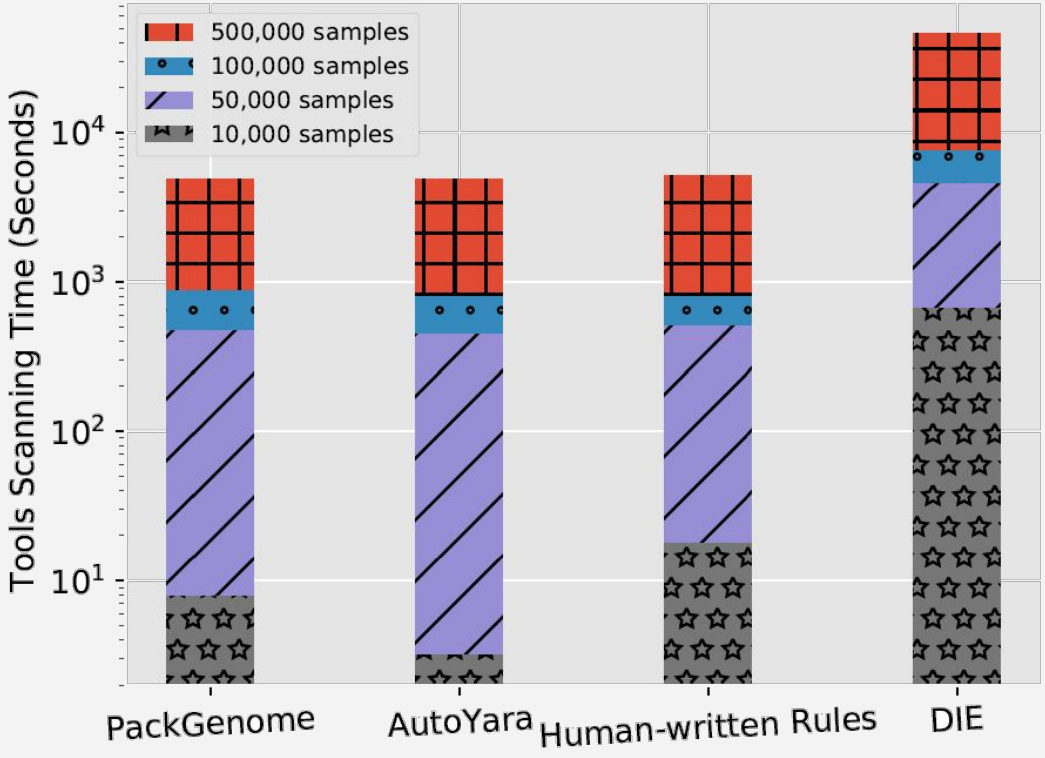
PackGenome是否能有效的生成不同類型的加殼器的偵測規則？

提取 packer-specific genes 並為 20 個現成加殼器生成 70 條規則，Byte selection 可以幫助 PackGenome 生成較低 misplace match 的規則。

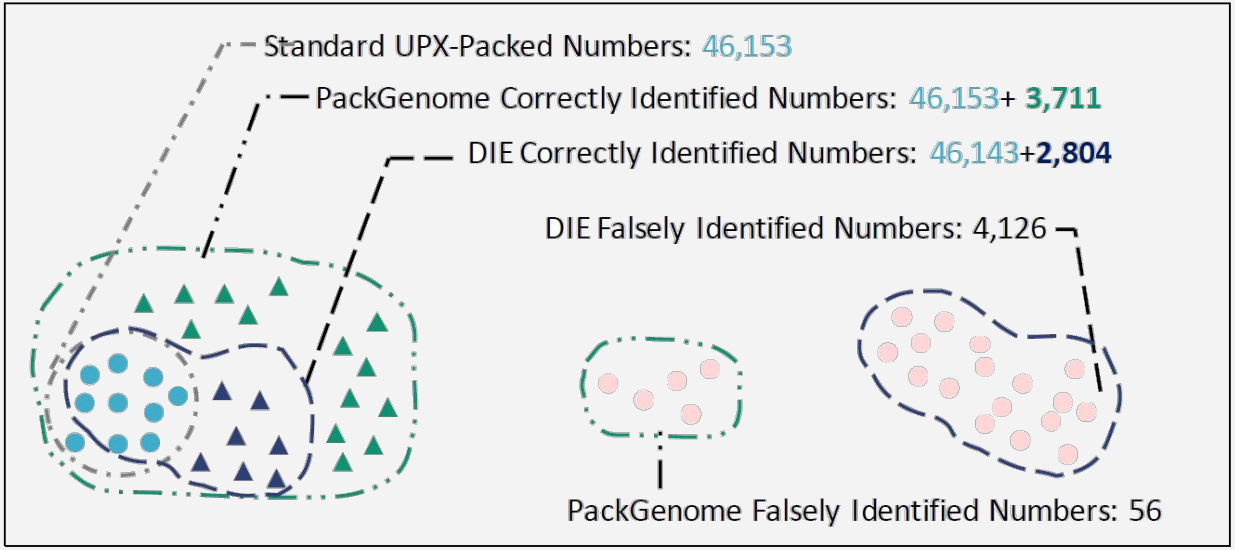


與人工編寫的規則和其他自動化工具相比，PackGenome的規則準確率和效率如 何？



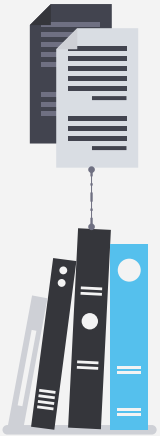


與人工編寫的規則和其他自動化工具相比，PackGenome的規則準確率和效率如 何？



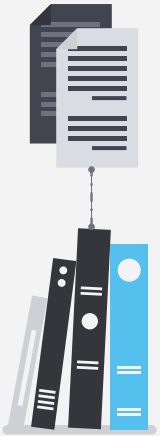
PackGenome生成的規則在偵測加殼樣本的可擴展性如何？

生成的規則適用於不同版本的加殼器，根據 packer specific gene 建立的規則可以直接偵測重複使用相同脫殼例程的客製化加殼器。



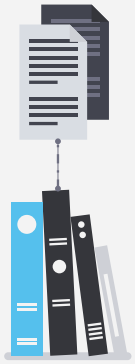
PackGenome生成的規則在偵測現實程式時的性能如何？

可以偵測到客制化加殼器並具有低誤判率。

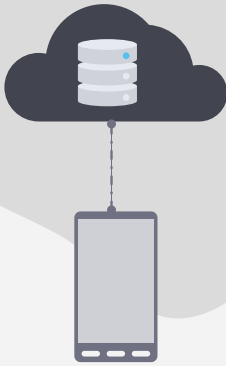


05

Discussion



Discussion



Missing brand-new packers

若能取得新的加殼器，還是能生成規則。或是收集相同加殼器的程式來生成。

Unavoidable byte mismatch

透過 byte selection 來降低 mismatch 機率，在 byte mismatch 和性能之間取得平衡。

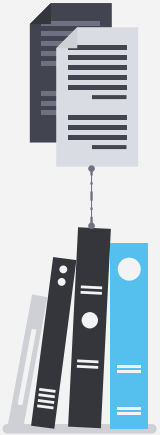
Heavyweight obfuscation

Signature-based 的限制是天生無法處理重度混淆，PackGenome 使用特殊結構來克服輕度混淆。

06

Conclusion

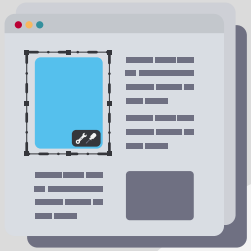
Conclusion



流通的加殼惡意軟體太多，分析人員依靠 signature-based 的偵測來確定加殼器，以便進行脫殼。
但現有的加殼器 signature 生成嚴重依賴分析人員的經驗，這使得編寫和維護規則繁瑣且容易出錯。

PackGenome 是用於加殼器偵測的自動 YARA rule 生成框架，從被相同加殼器的程式重複使用的脫殼例程收集加殼器偵測規則。並提出第一個系統性評估位元組規則 mismatch 機率的模型。

實驗表明，PackGenome 在零漏判、低誤判和微小的掃描開銷方面優於現有的人工規則和工具。



Thanks!

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**