Static Analysis of Executables to Detect Malicious Patterns (POPLミーティング :6/8)

米澤研究室
M1 佐藤秀明

本日の概要

- 論文のサーベイ
- Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns, Usenix Security Symposium, August 2003.
  - SAFE(a Static Analyzer For Executables)
    - 難読化されたウイルスを検知するシステム
    - 一般化されたシグネチャによるマッチング

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

Malicious Code

- 悪いことをするコード
- ホストの防御において Malicious code の検知は最重要

Malicious Code の分類

- ウイルス
  - 自分自身のコピーを他のプログラムに attach する
- ワーム
  - ネットワークを介して増殖する
- トロイの木馬
  - システム攻撃やデータ流出を行うコードを内包する
- バックドア
  - システムを外部から制御可能にする
- スパイウェア
  - 役に立つプログラムではあるが、内部のデータを流出させる

本論文の照準

- ウイルスに感染したプログラムの高精度な検出
- 動的解析は使わない
  - 検出にかかるオーバーヘッド

(単純な) シグネチャマッチング

- 要するに grep
  - ex. Chernobyl/CIH
    - E800 0000 005B 8D4B 4251 5050 0F01 4C24 FE5B 83C3 1CFA 8B2B

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 42h]
51              push ecx
50              push eax
50              push eax
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
FA              cli
8B 2B           mov ebp, [ebx]
```

(単純な) シグネチャマッチングの問題

- ウイルスが難読化された場合に対処できない
  - 難読化 : 意味的に等価な変換をプログラムに施すこと
- 一般的な難読化手法に耐え得る枠組みが必要

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

難読化されたウイルス

- Metamorphic virus
  - 複製時にコードの変換を行う(詳細は後述)
    - Dead-code Insertion
    - Code transposition
    - Register reassignment
    - Instruction substitution
- Polymorphic virus
  - 暗号化された命令を実行時に復号化
  - 複製される度に異なる鍵で暗号化
  - 復号ルーチン自体は Metamorphic に難読化

考慮する難読化手法

- Metamorphic virus を取り扱う
  - 一般的に用いられるお手軽な手法
  - 単純なシグネチャマッチングを無効化

Dead-code insertion(1)

- 余分なコードを挿入
- ex1. nop の挿入 … 正規表現で grep できる

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 45h]
90              nop
51              push ecx
50              push eax
50              push eax
90              nop
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
90              nop
FA              cli
8B 2B           mov ebp, [ebx]
```

```
E800     0000     00(90)*
5B(90)*  8D4B     42(90)*
51(90)*  50(90)*  50(90)*
0F01     4C24     FE(90)*
5B(90)*  83C3     1C(90)*
FA(90)*  8B2B
```

Dead-code insertion(2)

- ex2. Nop 以外の挿入 … 不要な命令の除去で対応
  - (*) は簡単
  - (**) は困難

```
call 0h
pop ebx
lea ecx, [ebx+42h]
nop                          (*)
nop                          (*)
push ecx
push eax
inc eax                      (**)
push eax
dec [esp - 0h]               (**)
dec eax                      (**)
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code Transposition

- 命令ブロックをシャッフル

```
        call 0h
        pop ebx
        jmp S2
S3:     push eax
        push eax
        sidt [esp - 02h]
        jmp S4
S5:     cli
        jmp S6
S2:     lea ecx, [ebx+42h ]
        push ecx
        jmp S3
S4:     pop ebx
        add ebx, 1Ch
        jmp S5
S6:     mov ebp, [ebx]
```

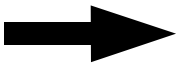
Register Reassignment

- レジスタの名前を交換
  - live なレジスタを dead なレジスタに置き換える

Instruction Substitution

- 等価な別の命令列に置き換える
  - 等価な命令列の辞書をあらかじめ用意

```
call 0h
pop ebx
lea ecx, [ebx+42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```



```
call 0h
pop ebx
lea ecx, [ebx+42h]
sub esp, 03h
sidt [esp - 02h]
add [esp], 1Ch
mov ebx, [esp]
inc esp
cli
mov ebp, [ebx]
```

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

対象とするウイルス(1)

- Chernobyl(CIH)
  - プログラム中の未使用領域を探索
    - 各セクション間(アラインメントの制約による)
  - その各領域に自分自身のコードを分割して挿入
- z0mbie-6.b
  - コードの暗号化
  - 毎回異なるサイズのコードが生成される

対象とするウイルス(2)

- f0sf0r0
  - コードの暗号化
  - エントリポイントやヘッダ情報を改変しない
    - プログラムの最初からではなく、途中でウイルスに制御が移る
    - 検出が面倒
- Hare
  - 復号化ルーチンが毎回変化

予備実験

- 難読化を施したウイルスを商用ウイルススキャナにかけてみた
  - Nop insertion と code transposition を行っただけで全滅
- 我々のシステム (SAFE) は Nop insertion と code transposition への耐性をもつ

商用ソフトの難読化への耐性

		Norton® Antivirus 7.0	McAfee® VirusScan 6.01	Command® Antivirus 4.61.2	SAFE
Chernobyl	original	✓	✓	✓	✓
	obfuscated	✗[1]	✗[1,2]	✗[1,2]	✓
z0mbie-6.b	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓
f0sf0r0	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓
Hare	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓

Obfuscations considered: [1] = nop-insertion (a form of dead-code insertion)
[2] = code transposition

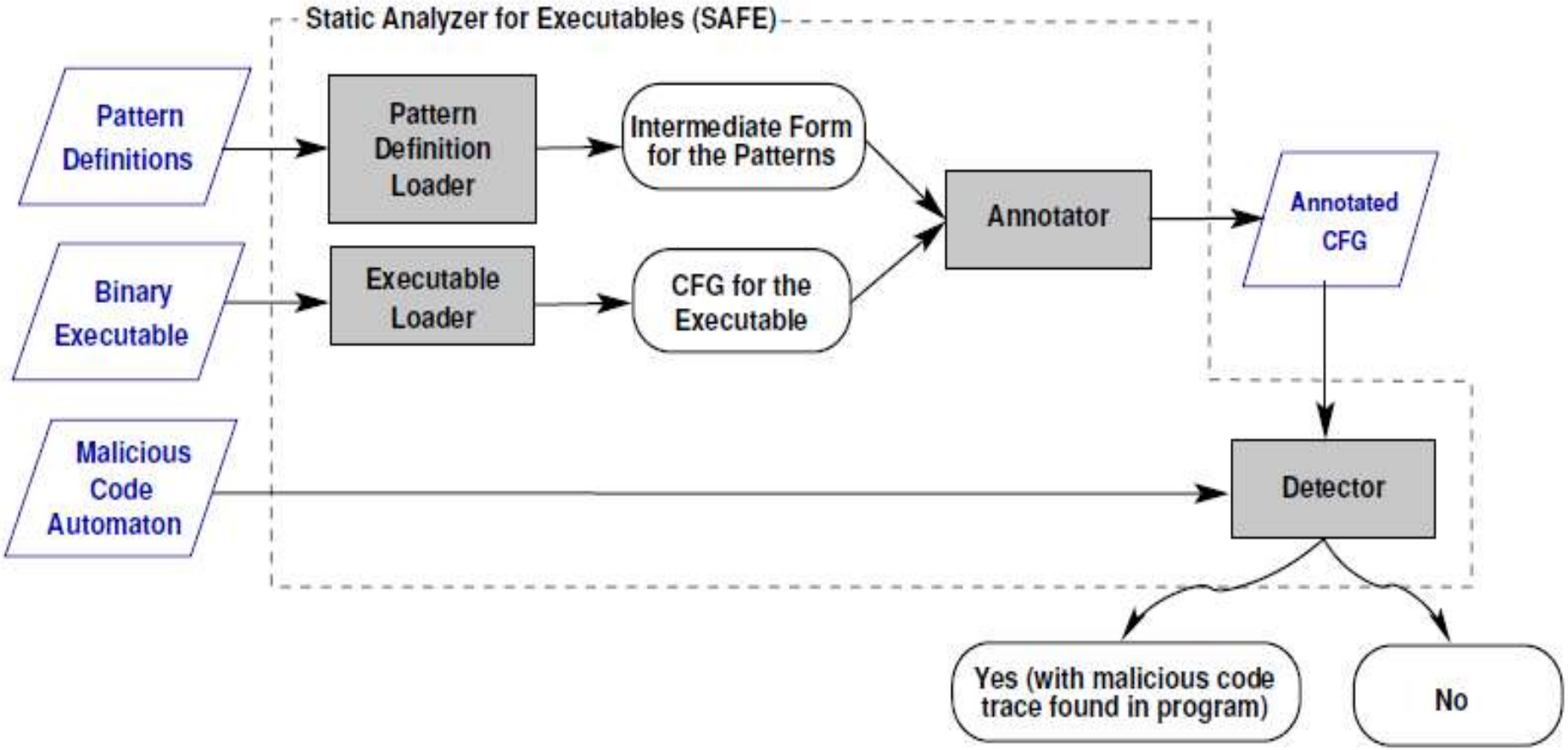
構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

SAFE の構成

- Pattern-definition loader
  - プログラム中に現れる典型的なパターンを用意
- Executable loader
  - 検査するプログラムの CFG を作成
- Annotator
  - パターンを CFG に unify し、CFG に印をつける
- Malicious code automaton(MCA)
  - 一般化された Malicious code
- Detector
  - Annotated CFG と MCA を照合して結果を出す

SAFE の構成 (図)


Static Analyzer for Executables (SAFE)
Pattern Definitions
Pattern Definition Loader
Intermediate Form for the Patterns
Annotator
Annotated CFG
Binary Executable
Executable Loader
CFG for the Executable
Malicious Code Automaton
Detector
Yes (with malicious code trace found in program)
No

Predicates(1)

- 種々の静的解析を行うためのライブラリ

$Dominators(B)$	the set of basic blocks that dominate the basic block B
$PostDominators(B)$	the set of basic blocks that are dominated by the basic block B
$Pred(B)$	the set of basic blocks that immediately precede B
$Succ(B)$	the set of basic blocks that immediately follow B
$First(B)$	the first instruction of the basic block B
$Last(B)$	the last instruction of the basic block B
$Previous(I)$	$\begin{cases} \bigcup_{B' \in Pred(B_I)} Last(B') & \text{if } I = First(B_I) \\ I' & \text{if } B_I = \langle \ldots, I', I, \ldots \rangle \end{cases}$
$Next(I)$	$\begin{cases} \bigcup_{B' \in Succ(B_I)} First(B') & \text{if } I = Last(B_I) \\ I' & \text{if } B_I = \langle \ldots, I, I', \ldots \rangle \end{cases}$
$Kills(p, a)$	*true* if the instruction at program point p kills variable a
$Uses(p, a)$	*true* if the instruction at program point p uses variable a
$Alias(p, x, y)$	*true* if variable x is an alias for y at program point p
$LiveRangeStart(p, a)$	the set of program points that start the a's live range that includes p
$LiveRangeEnd(p, a)$	the set of program points that end the a's live range that includes p
$Delta(p, m, n)$	the difference between integer variables m and n at program point p
$Delta(m, p_1, p_2)$	the change in m's value between program points p_1 and p_2
$PointsTo(p, x, a)$	*true* if variable x points to location of a at program point p

Predicates(2)

- 値の変更に無関係な命令を Delta() で検知
  - Polyhedral analysis[Cousot et al., 1978] を利用

型システムの導入 (1)

- アセンブリに型をつける
- subtype に関して束を形成

τ	::	ground	*Ground types*
	|	τ [n]	*Pointer to the base of an array of type* τ *and of size* n
	|	τ (n]	*Pointer into the middle of an array of type* τ *and of size* n
	|	τ `ptr`	*Pointer to* τ
	|	$\mathrm{s}\{\mu_1, \ldots, \mu_k\}$	*Structure (product of types of* μ_i*)*
	|	$\mathrm{u}\{\mu_1, \ldots, \mu_k\}$	*Union*
	|	$\tau_1 \times \cdots \times \tau_k \rightarrow \tau$	*Function*
	|	$\top$ (n)	*Top type of* n *bits*
	|	$\bot$ (n)	*Bottom type of* n *bits (type "any" of* n *bits)*
μ	::	(l, τ, i)	*Member labeled* l *of type* τ *at offset* i
ground	::	`int` $(g{:}s{:}v)$ | `uint` $(g{:}s{:}v)$ | ...	

型システムの導入 (2)

- Ground type は triple(g:s:v) をもつ
  - g: 無視される上位ビット長
  - s: サインビット長
  - v: 値を表す下位ビット長

IA-32 Datatype	*Type Expression*
`byte unsigned int`	uint (0:0:8)
`doubleword signed int`	int (0:1:31)
`double precision float`	float (0:1:63)
`near pointer`	⊥ (32)
`far pointer (logical address)`	uint (0:0:16) × uint (0:0:32) → ⊥ (48)
`eax, ebx, ecx, edx`	⊥ (32)
`esi, edi, ebp, esp`	⊥ (32)
`eip`	int (0:1:31)
`cs, ds, ss, es, fs, gs`	⊥ (16)

型システムの導入 (3)

- 型付けの例

Code	*Type*
`call 0h`	
`pop ebx`	ebx : ⊥(32)
`lea ecx, [ebx + 42h]`	ecx : ⊥(32),
	ebx : ptr ⊥(32)
`push ecx`	ecx : ⊥(32)
`push eax`	eax : ⊥(32)
`push eax`	eax : ⊥(32)
`sidt [esp - 02h]`	
`pop ebx`	eax : ⊥(32)
`add ebx, 1Ch`	ebx : int(0:1:31)
`cli`	
`mov ebp, [ebx]`	ebp : ⊥(32),
	ebx : ptr ⊥(32)

Abstraction Pattern(1)

- プログラムがとる典型的なパターン
- Γ(FV)=(V,O,C) の構成をもつ
  - Γ: パターンの名前
  - FV: パターン中に現れる自由変数
  - V: パターン中に現れる変数
  - O: パターンを形成する命令列
  - C: 条件を補足する boolean expressions
- レジスタは変数に置き換えられている
  - Register reassignment に対応するための多相化

Abstraction Pattern(2)

- Abstraction Pattern の例

$$
\begin{aligned}
&\Gamma(\, X : int(0:1:31)\,) = \\
&\qquad (\quad \{\, X : int(0:1:31)\,\}, \\
&\qquad\quad \langle\ p_1 :\ \text{“}pop\ X\text{”}, \\
&\qquad\qquad\ p_2 :\ \text{“}add\ X, \texttt{03AFh}\text{”}\ \rangle, \\
&\qquad\quad p_1 \in LiveRangeStart(p_2, X)\ \ )
\end{aligned}
$$

Executable Loader

- プログラムを読み込み、CFG を構成
  - 逆アセンブルに IDA Pro を利用
  - プログラム解析に CodeSurfer を利用

Annotator

- CFG の各ノードと各 Abstraction Pattern を unify

$$Annotation(n) = \{\ [\Gamma, \mathcal{B}]\ : \Gamma \in \{\Gamma_1, \ldots, \Gamma_m\} \wedge \mathcal{B} = Unify(S(n), \Gamma)\ \}$$

  - n: CFG 中のノード
  - Γ: パターン
  - B: unify による変数束縛の結果
  - {Γ1, Γ2, ..., Γm}: パターンの集合
  - S(n): ノード n に至る命令列
- Chernobyl のコード片に対する難読化と Annotation の例は Figure 8 に

Malicious Code Automaton

- ウイルスを一般化したもの
  - アルファベット (= パターン) を変数で多相化

Formally, a *malicious code automaton* (or *MCA*) $\mathcal{A}$ is a 6-tuple $(V, \Sigma, S, \delta, S_0, F)$, where

- $V = \{v_1 : \tau_1, \ldots, v_k : \tau_k\}$ is a *set of typed variables*,
- $\Sigma = \{\Gamma_1, \ldots, \Gamma_n\}$ is a *finite alphabet* of patterns parametrized by variables from V, for $1 \leq i \leq n$, $P_i = (V_i, O_i, C_i)$ where $V_i \subseteq V$,
- S is a finite set of *states*,
- $\delta : S \times \Sigma \rightarrow 2^S$ is a *transition function*,
- $S_0 \subseteq S$ is a non-empty set of *initial states*,
- $F \subseteq S$ is a non-empty set of *final states*.

- Chernobyl のコード片を表す MCA は Figure 11 に

Detector(1)

- Annotated CFG とCMA がともに受理できる言語を計算
  - 言語 = パターン列の集合
  - Annotated CFG は全ノードを終状態とみる
- アルゴリズムは Figure 12 に

Detector(2)

- 両方の状態遷移を並行になぞる
  - 遷移とともに変数束縛の情報も伝搬
  - 伝搬された束縛情報とAnnotationの束縛情報が矛盾しないときに限り、遷移を許す
    - compatible() - 同じ変数に異なる値が束縛されていないか
- 到達可能な状態の情報を反復的に更新
  - CMAの終状態に到達可能なら陽性

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

評価 (1)

- 各ウイルスにつき 10 種類のバリエーションを作成
  - 難読化のパラメータを変更
- IDA Pro と Code Surfer の実行時間は全体の実行時間に含めず

評価 (2)

- False negative
  - 検出漏れはなかった

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
Chernobyl	1.444 s	(0.497 s)	0.535 s	(0.043 s)
z0mbie-6.b	4.600 s	(2.059 s)	1.149 s	(0.041 s)
f0sf0r0	4.900 s	(2.844 s)	0.923 s	(0.192 s)
Hare	9.142 s	(1.551 s)	1.604 s	(0.104 s)

評価 (3)

- False positive
  - 別のウイルスを Chernobyl として誤検出することはなかった

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
z0mbie-6.b	3.400 s	(1.428 s)	1.400 s	(0.420 s)
f0sf0r0	4.900 s	(1.136 s)	0.840 s	(0.082 s)
Hare	1.000 s	(0.000 s)	0.220 s	(0.019 s)

評価 (4)

- False positive(続き)
  - 無害なプログラムを Chernobyl として誤検出することはなかった
  - 一部非常に時間がかかった

	Executable size	.text size	Procedure count	Annotator		Detector	
				avg.	(std. dev.)	avg.	(std. dev.)
tiffdither.exe	9,216 B	6,656 B	29	6.333 s	(0.471 s)	1.030 s	(0.043 s)
winmine.exe	96,528 B	12,120 B	85	15.667 s	(1.700 s)	2.283 s	(0.131 s)
spyxx.exe	499,768 B	307,200 B	1,765	193.667 s	(11.557 s)	30.917 s	(6.625 s)
QuickTimePlayer.exe	1,043,968 B	499,712 B	4,767	799.333 s	(5.437 s)	160.580 s	(4.455 s)

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

関連研究

- Typed Assembly Language [Morrisett et al., 1998]
  - アセンブリコードに対する型システム
- Specification-based monitoring [Giffin et al., 2002][Wagner et al., 2001]
  - プログラムの挙動をオートマトンで規定

構成

- 動機づけ
- ウイルスとその難読化
- 予備実験
- SAFEの構成
- 評価
- 関連研究
- まとめ

まとめ

- 商用ウイルススキャナは難読化に弱い
- ウイルスパターンを静的に解析するシステムの提案と実装
- 一般的な難読化手法を克服できることを実証