演習 3 (6/30)

31010 佐藤秀明

テーマの選択について

- 型システム、または Watermarking
  - 双方の資料にざっと目を通してみた。で、結局…
- Watermarking に決定
  - 型システムについてはまたの機会に (すみません)

概要

- Path-based watermarking
  - プログラムの分岐構造の中に透かしを埋め込む
    - Java バイトコード
    - ネイティブコード
- 今後の予定

Path-based watermarking

- dynamic(cf. static)
  - ある特定の入力を元に実際にプログラムを実行することで透かしが得られる
- blind(cf. informed)
  - 対象プログラムから透かしを感知するために必要なのはその対象プログラムのみ
- fingerprinting(cf. watermark)
  - プログラムの各コピーごとに違う値の透かしを埋め込むことが可能

Java バイトコード (1)

- 概要

  1)埋め込みたい透かし情報 W(n ビットのビット列) を k 個のビット列 $w_1, w_2, ..., w_k$ に分解する

  2)ある特定の入力 I に対してプログラムを実行したときの、各 breakpoint におけるローカル変数の値を記録する

  3) 1) で作成した $w_1, w_2, ..., w_k$ に対応するような分岐の列を、プログラムのあちこちに分散させて埋め込む

  4) 3) を I について実行すると、その分岐構造は $w_1, w_2, ..., w_k$ を含むようなビット列になっている

Java バイトコード (2)

- 透かし情報の分解
  - 中国の剰余定理

    m_1, m_2を互いに素、a_1, a_2を整数する。このとき,

    $$x \equiv a_1 \left(mod\ m_1 \right)$$
    $$x \equiv a_2 \left(mod\ m_2 \right)$$

    を満たす整数 xが $m_1 m_2$を法にしてただひとつ存在する。
  - 互いに素な正整数 $p_1, p_2, \ldots, p_k$ を用いて W を複数個の $w_1, w_2, \ldots, w_k$ に分解する
  - $w_1, w_2, \ldots, w_k$ のうち数個を読み取ることができれば、W を復元することが可能

Java バイトコード (3)

- 透かし情報の分解の例

$$\left(p_1=2, p_2=3, p_3=5 \text{のとき}\right)$$

$$W=17 \Leftrightarrow \begin{cases} W=5 \bmod p_1 p_2 \\ W=7 \bmod p_1 p_3 \\ W=2 \bmod p_2 p_3 \end{cases} \Leftrightarrow \begin{cases} 5=5=w_1 \\ p_1 p_2+7=13=w_2 \\ p_1 p_2+p_1 p_3+2=18=w_3 \end{cases}$$

- $w_1, w_2, \ldots, w_k$ から W を復元するときは、間違った w_k が含まれている可能性を考慮する必要がある
  - 同時には成り立ち得ない合同式の対を両方同時には採用しないようにする

Java バイトコード (4)

- コード生成
  - ある条件分岐命令 i について、その直後に実行される命令 j が、最初に i を実行したときの直後に実行された命令 j' と同一であれば、その i はビット 0 に対応するものとみなす。そうでなければビット 1 とみなす。
    - 分岐構造を基にしてビット列を構成
  - 常に false になるが、静的解析だけでは false であることが分からないような条件 P^F を用意する
    - 例えば、x(x-1) != 0 mod 2 など

Java バイトコード (5)

- コード生成の例
  - ループによる表現

```
int bits = 0xa;
int count = 5;
int j = 0;
for(int i = 0; i<counter; i++, bits = bits >> 1)
  if((bits & 1) == 1) j++;
if(P^F) (生きている変数) += j;
```

    - これは”0101”を表す

Java バイトコード (6)

- コード生成の例 (続き)
  - Condition Code による表現

```
/* a, b, c, d は生きているローカル変数。これらの値は
解析により既知 */
int tmp = 0;
/* この時点で c==d, a==b */
/* この時点では、c == d はビット 1、a==b はビット 0 に
対応させておく */
if(c == d) tmp++;
if(a == b) tmp++;
/* この時点で c!=d, a==b */
if(c == d) tmp++;
if(a == b) tmp++;
if(P^F) (生きている変数) += tmp;
```

    - これは ”1010” を表す

ネイティブコード (1)

- Java バイトコードと異なり、リターンアドレスに対する演算が可能
- 無条件分岐の命令を、分岐を処理するための関数 (branch function) を call する命令によって表現する

$$a_i : \mathrm{jmp}\ b_i$$
$$\downarrow$$
$$a_i : \mathrm{call}\ f$$

- f は a_i から b_i を求められるような map を保持しており、f の call から抜けた後は b_i のアドレスに PC が移る

ネイティブコード (2)

- map の作り方

  1)ハッシュ関数 $h : \{a_1, a_2, ..., a_n\} \rightarrow \{1, 2, ..., n\}$ を用意する

  2)メモリに表 T をとり、$T[h(a_i)] = a_i \text{ xor } b_i$ となるように各値を store する

- Call 命令の埋め込み方
  - $a_{i+1} = b_i$ とし、ビット列を連続的に表現する
  - $W_i = 1$ なら $a_i < a_{i+1}$ となるように、$W_i = 0$ なら $a_i > a_{i+1}$ となるように、call 命令を a_i に埋めてゆく

攻撃の種類

- nop 挿入
- 分岐の真偽を入れ替える
- Watermarking を再びかけることでもともとの watermark を隠蔽する
- Branch function をバイパスする
- Branch funciton への到達方法に細工を施す

実験結果 (論文より)

- Java バイトコード
  - 何度も実行されるコードの割合が多いと実行時間が長くなる
  - W をできるだけたくさんの w_i に分割したほうが攻撃に対して強い
  - ブランチの増加による slowdown とのトレードオフ
- ネイティブコード
  - Watermarking を施してもサイズはほとんど増加しない
  - Watermarking により実行の速くなったプログラムもあった
  - Branch function の存在を悟られにくいような工夫が必要

今後の予定

- 他の watermarking 手法について調べる
  - 例えばレジスタ干渉グラフなど
- 余裕があれば開発 framework をいじってみる

References

- C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn and M. Stepp, “Dynamic Path-Based Software Watermarking”
  - http://www.cs.arizona.edu/~linnc/research/