A general model for hiding program control flow

Jan Cappaert
K.U.Leuven/ESAT/COSIC

Overview

- Introduction
- Related research
- CFG flattening
  - Complexity
  - Security
  - Model and applications
- Conclusions

Introduction

- Programs (source, binary, …) usually contain 'explicit' control flow information
  - ➔ Subject to static (and dynamic) analysis
    - Local/global
    - Forward/backward
- Goal: "hide control flow"
  - Software / hardware ?
  - Stand-alone / "client-server" ?

Related research

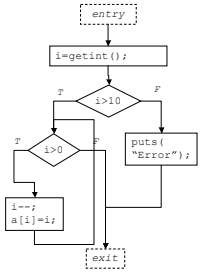
- 1996, Aucsmith. "Tamper Resistant Software: an Implementation."
  - A software-only TRS system
    - Multi-file (mutual checking)
    - Against static/dynamic analysis and tampering
      - Hash functions, encryption, signatures, …
- 2000, Wang *et al.* Software Tamper Resistance: Obstructing Static Analysis of Programs
  - Static CFG "degeneration"
    - 1) `if`-then-`goto` 2) Indirect control transfers
    - Data flow complexity through aliasing or index computation

Related research

- 2005, Ge et al. "Control Flow Based Obfuscation."
  - CF info protected by Aucsmith TRS module
  - Software only
- 2007, László and Kiss. "Obfuscating C++ Programs via Control Flow Flattening."
  - Contains performance impact + "complexity"
  - Hard: `try-catch` translation (error handling)

CFG: an example

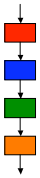
```
i=getint();
if (i>10)
  while (i>0) {
      i--;
      a[i]=i;
  }
else
  puts("Error");
```



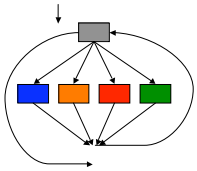
CFG flattening

- Control flow graph flattening transforms all control flow graphs (CFGs) to a similar form
  - Less explicit control flow information
  - Static control flow analysis is less trivial, requires data flow analysis
    - Data flow analysis is provably hard under certain conditions

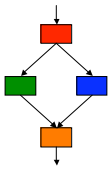
Examples



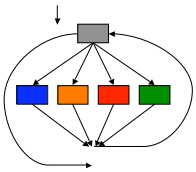
becomes



Examples



becomes



In C

```
A; B; C; D;
```

```
swVar = 0;
while(swVar!=4)
switch(swVar) {
  case '0':
    A; swVar=1;
    break;
  case '1':
    B; swVar=2;
    break;
  case '2': …
}
```


Analysis

- Forward/backward analysis
  - Based on local constants
  - Local ~ trivial
- Solution?
  - Relative updates: `swVar+=cst;`
    - Solve local equations
    - Slightly harder ~ easy

Branching [László and Kiss]

```
…
A;
if (cond) B;
else C;
…
```

```
switch(swVar) {
  …
  case '66':
    A;
    if (cond) swVar=67;
    else swVar=68;
    break;
  case '67':
    B; swVar=…
    break;
  …
```

Branching revisited

```
…
case '66':
  A;
  if (cond) swVar=67;
  else swVar=68;
  break;
…
}
```

```
switch(swVar) {
  …
  case '66':
    A;
    swVar =
     swVar +2 -1*(cond)
    break;
  …
}
```


Analysis

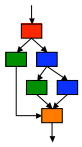
- Forward / backward analysis
  - `(cond)` → {false,true} = {0,1}
  - Local equations have 2 solutions
  - If no convergence → 2^n paths … in theory

Complexity

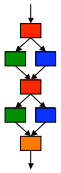
- McCabe's cyclomatic number v(G)
  - CFG G = <V,E>
  - v(G) = E – V + 2P with P connected components
  - v(G) = E – V + 1 in a cyclomatic graph
- CFG flattening ~ v(G) x 3 [László and Kiss]
  - Meaning ?
    - What is v(G) !?
    - Upper bound ?

Complexity

- McCabe's v(G)
  - *"Number of decision points"* + 1
  - *"Number of independent paths"*
- Used for SW testing
  - path coverage

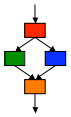


v(G) = 3

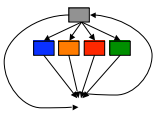


v(G) = 3

Complexity



v(G) = 2



v(G) = 4

- Switch, n + 1 nodes
  - n >> 1
  - v(G)=2n - n + 1
- v(G)
  - CFG structure
  - Number of nodes
    - Node splitting
    - Dummy nodes
    - …

On the importance of …

- … the *offset function* '+'
  - $label_{i+1} = label_i + cst + (cond) * offset$
- Example:
  - `swVar = swVar + 1 + (cond) * 7`
  - `(cond)` ∈ {0,1}; `swVar` ∈ {1, 2, 3, 10}
  - Any guesses?

The offset function '+'

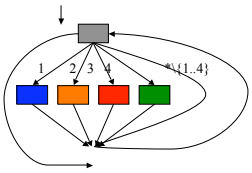
- Other candidates
  - Modular addition
  - Bitwise operations
    - XOR: $x \oplus cst = y$ is *bijective*
      - Can flip any bit
      - Attack Hamming distances
    - OR or AND: "destroy" bits, set or clear
      - x & $cst = y$ is *not bijective*
      - Good: non-deterministic backward analysis
      - Bad: restricted set for forward analysis
  - Others

On the importance of …

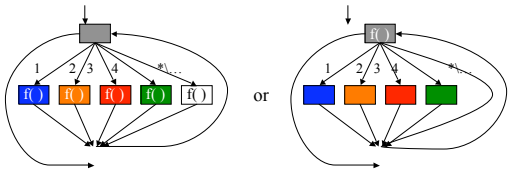
- … the iteration function *f*() …
  - $label_{i+1} = label_i + cst + (cond) * offset$
  - $label_{i+1} = f(label_i) + (cond) * offset$
- … and the "fall through path" of the switch

The switch's "fall-through path"

- Similar to `if-then` fall-through path
- Example:
  - `if` (*cond*) `then A;`
  - In C:
    - Cond → 0 ~ false
    - Cond → *\0 ~ true



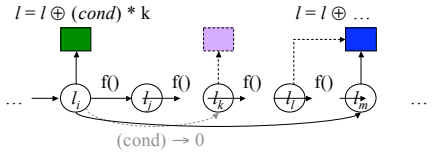
The iteration function



$label_{i+1} = f(label_i) + (cond)*offset$ or $label_{i+1} = f[label_i + (cond)*offset]$

$label_{i+1} = f[label_i + (cond)*offset]$

Iteration function + fall through



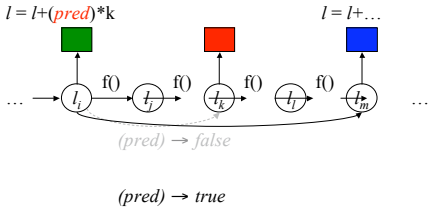
(cond) → 1

Attacks

- Forward/backward analysis
  - Solving local equations (if possible)
  - Iterating solutions
    - How far?
    - Forward? OK. Backward?

Opaque predicates

Static != dynamic



Further extensions

- Iteration function *f*()
- More than 2 targets

$$l_{x+1} = f\left(l_x \bigoplus_{i=0}^{m}(cond_i \cdot k_i) \bigoplus_{j=0}^{n}(pred_j \cdot k_j)\right)$$

- No Boolean mappings

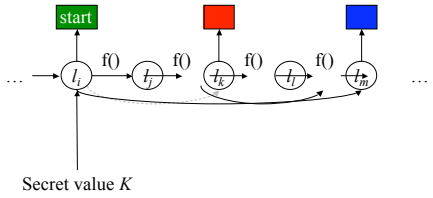
  $(expr_k) \rightarrow \{x, y, z, \ldots\}$

The iteration function

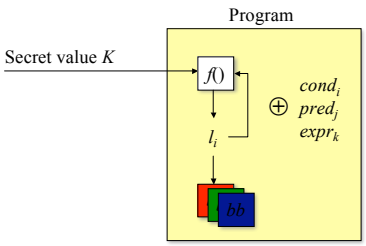
- Properties
  - Light and fast
  - Good diffusion properties
- Candidates
  - Hash (one-way)
  - Cipher (key dependent, SK or PK)
  - PRNG, LFSR, T-functions, …

Security

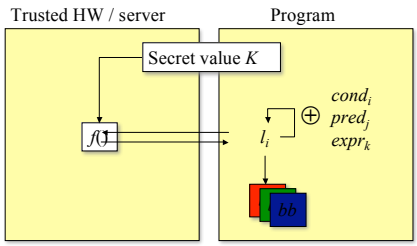
- Goal: $O(2^n)$ ~ brute-force guess of n-bit $l_{i \leftarrow 0}$



Models and applications



Models and applications



Models and applications

- Assumptions
  - If nothing can be learned from CFG without *K,*
  - if *basic blocks* small enough,
  - then a program is secure against static analysis
- Not
  - Dynamic
  - Tampering
- Applications
  - e-Voting, … ?

Conclusions

- CFG flattening
  - Structured form of obfuscation
  - Provable
- Model CFG flattening
  - Pluggable *f*(), '+', …
  - Applications: RE-TRUST model, …
- Links to crypto, info security, …