SLEAK:
A Side-channel Leakage Evaluator and Analysis Kit

Dan Walters
Andrew Hagen
Eric Kedaigle

WPI Applied Cryptology Seminar
March 2015

This technical data was produced for the U.S. Government under Contract No. FA8702-14-C-0001, and is subject to the Rights in Technical Data-Noncommercial Items Clause (DFARS) 252.227-7013 (NOV 1995)
Agenda

- **Background / Motivation**
  - Need for automated side-channel vulnerability analysis

- **Technical Approach**

- **Proof of concept**
  - Experimental results
  - Countermeasure analysis

- **Conclusion**
Background

- Side-channel attacks (SCA) present a major threat to secure embedded systems
  - Widely demonstrated over the last 15 years
  - Timing, instantaneous or static power consumption, electromagnetic radiation, others yet to be discovered…

- Software countermeasures known in theory, but difficult to implement properly
  - Requires expertise in side-channel and crypto
  - Unexpected compiler transformations
  - Platform-specific leakage sources

- All countermeasures must be empirically verified for effectiveness
SCA Testing Challenges

- **Expensive and slow**
  - Side-channel collection equipment
  - Instrumentation for each target hardware platform
  - Expertise to collect and analyze data

- **Results are often inconsistent** – affected by several parameters:
  - Skill and experience of tester/attacker
  - Choice of side-channel
  - Environmental conditions (e.g. temperature and background noise)
  - Data-collection equipment
  - Analysis techniques (new techniques constantly emerging)

- **Performed near the end of development cycle**

- **Difficult to isolate effects of individual countermeasures**
Our approach

- Use full-system simulator to analyze target binary
  - Inspect intermediate states of simulator during target execution
  - Automate vulnerability detection

- Advantages
  - Analysis is automated
  - Reduces expertise needed for tester
  - Results are reproducible
  - Reduces need for side-channel collection equipment
  - Analysis is performed on actual machine code (unmodified binary)
  - Applicable regardless of countermeasure or algorithm
  - Does not require a complex leakage model
    - Measure of “worst-case” information leakage
  - Source-code not required
Definitions

- **Target binary**
  - The machine-code/executable program under evaluation
  - Directly analyze the *unmodified* binary – not required that the program is branch-free
  - Requires an interface for setting all inputs (and categorization of inputs)

- **Leakage model**
  - Specifies the observable information that leaks via side-channels
  - *Leakage function* takes state of the machine as input and returns a value or set of values that represents observable information

- **Intermediate value**
  - The value (or state) of some system component during the system’s execution of the target binary

- **Trace**
  - Set of intermediate values for a single execution of the target binary
Definitions (cont.)

- **Mutual Information**
  - A measure of the mutual dependence between two random variables
    \[ I(X; Y) = \sum_{y \in Y} \sum_{x \in X} p(x, y) \log \left( \frac{p(x, y)}{p(x)p(y)} \right) \]

- **Perfect Masking**
  - An intermediate value is perfectly masked if it is statistically independent of the key
  - The MI between the intermediate state and the key is zero

- **Leakage Location**
  - An intermediate value that leaks secret information
    - \( \text{MI}(V,S) > 0 \), for some intermediate value \( V \) and secret value \( S \)
SLEAK – Algorithm

- **User inputs:**
  - Target binary
  - Interface to binary
    - Secret
    - Random
    - Public

- **Start by identifying intermediate values that depend on secret bits**
  - Optional optimization

- **Calculate MI between each intermediate value and each bit of the secret input**
  - Empirical studies show approximation rapidly converges to true value

```plaintext
Input: Target: the executable binary under analysis
Input: Public: the input space for the Public class
Input: Random: the input space for the Random class
Input: SecretBits: the set of input bits that are of type Secret

Output: Mutual information values for each leakage location

1: for s ∈ SecretBits do
2:   L ← IDENTIFYDEPENDENCIES(s)
3:   i ← 0
4:   Let P be a simple random sample of Public
5:   for p ∈ P do
6:     for all r ∈ Random do
7:       trace0_i ← RUNTARGET(p, r, s = 0, L)
8:       trace1_i ← RUNTARGET(p, r, s = 1, L)
9:       i ← i + 1
10:   end for
11: end for
12: for c ∈ L do  -> Calculate MI for each dependency
13:   values0 ← (trace0[c]_0, trace0[c]_1, ..., trace0[c]_{i-1})
14:   values1 ← (trace1[c]_0, trace1[c]_1, ..., trace1[c]_{i-1})
15:   Record CALCMI(0∥1, values0∥values1)
16: end for
17: end for
```
SLEAK - Proof-of-concept implementation

- Utilizes Gem5 simulator
  - Approach is not tied to any particular simulator

- Produces large quantity of information
  - Calculates mutual information for each key-dependent intermediate state

- Visualization tools to analyze results
SLEAK Views – Leakage Timeline

[Diagram showing MI Leakage over register and leakage location indices]
## SLEAK Views – Assembly Highlighting

<table>
<thead>
<tr>
<th>PC</th>
<th>#</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x897c</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>753b strb r3, [r7, #20]</td>
</tr>
<tr>
<td>0x897e</td>
<td>1</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>68bb ldr r3, [r7, #8]</td>
</tr>
<tr>
<td>0x8980</td>
<td>1</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>f103 0301 add.w r3, r3, #1</td>
</tr>
<tr>
<td>0x8984</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>60bb str r3, [r7, #8]</td>
</tr>
<tr>
<td>0x8986</td>
<td>1</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>683b ldr r3, [r7, #0]</td>
</tr>
<tr>
<td>0x8988</td>
<td>1</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>f103 0304 add.w r3, r3, #4</td>
</tr>
<tr>
<td>0x898c</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>681a ldr r2, [r3, #0]</td>
</tr>
<tr>
<td>0x898e</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>68bb ldr r3, [r7, #8]</td>
</tr>
<tr>
<td>0x8990</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>18d3 adds r3, r2, r3</td>
</tr>
<tr>
<td>0x8992</td>
<td>1</td>
<td>1.000000</td>
<td>1.000000</td>
<td>1.000000</td>
<td>781b ldrb r3, [r3, #0]</td>
</tr>
<tr>
<td>0x8994</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>2b00 cmp r3, #0</td>
</tr>
<tr>
<td>0x8996</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>d1dd bne.n 8954</td>
</tr>
<tr>
<td>0x8998</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>693a ldr r2, [r7, #16]</td>
</tr>
<tr>
<td>0x899a</td>
<td>1</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.000000</td>
<td>687b ldr r3, [r7, #4]</td>
</tr>
</tbody>
</table>

Assembly view highlighting leaking instructions
Experimental Results

- Compared results from SLEAK to results based on side-channel measurements on physical hardware

- Hardware platform:
  - BeagleBone Rev A6
    - Common dev board used for hobbyist work and prototyping
  - 720MHz ARM Cortex-A8
    - Core used in many SoCs and consumer devices (e.g. smart phones)
  - Angstrom Linux

- Collection equipment
  - EM Side-channel
  - Near-field magnetic probe
  - 14-bit ADC 400MS/s
Experimental Results – Leakage Patterns

- Tested with AES256 - unprotected implementation
  - Tracked number of register writes performed over a sliding time window
  - Same binary run on HW platform and SLEAK

- EM Radiation from Cortex-A8 strongly related to register writes
  - Strong indication of simulator accuracy for execution of test program
  - Register updates are a significant source of observed EM leakage
Experimental Results – Masking Strength

- Tested with custom test programs
  - Performs s-box table lookup on secret input value
  - Protected with boolean masking countermeasure
  - Multiple versions test program, adjusted distribution of mask values

EM Leakage of S-Box Lookups

Mutual Information (bits)

Time (ns): 150, 200, 250, 300

Graph showing EM leakage with different masking strengths.
Experimental Results – Masking Strength Comparison with SLEAK

Comparison between MI calculated with real EM measurements and MI from SLEAK using a generic leakage model
Experimental Results – Masking Strength Comparison with SLEAK

Comparison between MI calculated with real EM measurements and MI from SLEAK using a hamming-weight leakage model
Experimental Results – Key Extraction

Maximum correlation for each guess of an AES key-byte plotted against the number of traces. The correct guess is black and incorrect guesses in gray.
Real-world Countermeasure Analysis

- Analyzed two different AES implementations with countermeasures
  - DPA Contest v4.1 (www.dpacontest.org)
    - Uses Rotating S-box Masking (RSM) - a modern and sophisticated masking countermeasure
    - (Ported for our HW platform)
  - Custom boolean-masked version of AES
    - Intended to be completely secure against register-based leakage

- SLEAK identified unexpected vulnerabilities in both implementations
Low-Entropy Masking Schemes

- **Rotating S-box Masking** is a “low-entropy masking scheme”
  - Number of discrete mask values is less than number of possible intermediate values that it protects
    - RSM uses 16 mask values (4 bits of entropy), while it is used to protect 8-bit intermediate values
  - Trade-off between security and performance
    - Reduced entropy allows for more efficient implementations

- **SLEAK detected several leakage locations due to low-entropy**
  - Not surprising
  - Easy to ignore locations with an “acceptable” level of leakage
Register updates are commonly a major source of side-channel leakage
- Identified register-update vulnerability in RSM implementation

Cannot be detected from static-analysis of source-code or IR
- Register allocation occurs in compiler backend

Assembly Listing:
```
c3ca:  68fb  ldr   r3, [r7, #12]
c3cc:  781b  ldrb  r3, [r3, #0]  \(\text{New value of r2 (mask index)}\)
c3ce:  461a  mov   r2, r3
```
```
c416:  18d3  adds  r3, r2, r3
cc18:  781a  ldrb  r2, [r3, #0]  \(\text{Old value of r2 (state byte)}\)
c41a:  f107  add.w  ip, r7, #64  ; 0x40
```

C context:
```
// subBytes
for(=0; i<16; ++i){
    idx = (((i0 + dummy_idx[i]) % 16)*256);
    tmp[dummy_idx[i]] = *(aes_sbox0+(idx + (state->s[ dummy_idx[i]])));
}
```
**Mask Cancellation and Intrinsic Functions**

- **memcpy call is a single intrinsic function at IR level**
  - Later compiler stage “expands” to several assembly instructions
  - Performance optimization - loads multiple bytes into single register before storing to memory

- **Cannot be detected from static-analysis of source-code or IR**
  - Lowering of intrinsic functions occurs in compiler backend

- **Implementation of intrinsic functions can be significantly different across different target platforms**

```c
call void @llvm.memcpy.p0i8.p0i8.i32( i8* %2, i8* %3, i32 16, i32 1, i1 false)
```

---

Partial Assembly Listing for memcpy
(leaking instructions highlighted in gray)

```
bl138: ldrb r0, [r4, #9]
bl13c: ldrb r2, [r1, #8]!
bl140: orr r0, r2, r0, lsl #8
bl144: ldrb r2, [r1, #2]
bl148: ldrb r1, [r1, #3]
bl14c: orr r1, r2, r1, lsl #8
bl150: orr r0, r0, r1, lsl #16
bl154: str r0, [sp, #12]
```
Conclusion

- Propose the use of full-system simulators to analyze software for side-channel vulnerabilities
  - Demonstrated technique for automated analysis

- SLEAK can help reduce the cost of evaluating the security of software implementations
  - Detects complex vulnerabilities
    - Platform-specific leakage
    - Vulnerabilities introduced from compiler effects
  - Can be used directly by cryptographic software developer
    - Does not require physical hardware or side-channel traces
    - Does not require expertise in side-channel analysis
    - Testing can be performed early in the development cycle (when it is easier to fix)
  - Approximating mutual information is a practical approach

- Identified real-world cases where high-level code and IR is secure, but vulnerabilities are introduced during compiler’s final stages
Questions

Dan Walters (dwalters@mitre.org)