One Truth Prevails: A Deep-learning Based Single-Trace Power Analysis on RSA–CRT with Windowed Exponentiation

Kotaro Saito, Akira Ito, Rei Ueno, and Naofumi Homma
Tohoku University
Side-channel attack (SCA) on RSA

- SCA on modular exponentiation to estimate secret exponent
  - Traditional attacks distinguish squaring and multiplication to estimate exponent
  - Many studies have been devoted to how to accurately estimate exponent

Secret key leakage

Power/EM trace from RSA module

Partial key exposure attack

- Secret keys estimated by SCA is not always correct/complete
- Estimate full RSA–CRT secret key from partial/noisy leakage
Deep-learning based SCA (DL-SCA)

- Strongest profiled SCA which requires detailed assumption about leakage (compared to, for example, template attack)

- DL is very strong tool for SCAs, but researchers should still consider “what-to-learn” for key recovery
  - For symmetric cipher, it would be well established
  - But for public key decryption/signing, it varies depending on PKE
This work

- We present new deep-learning based single-trace power/EM analysis on state-of-the-art RSA–CRT implementations
  - New attack methodology for windowed exponentiation with dummy load
  - Leverage DL technique to estimate window values accurately
  - New partial key exposure attack algorithm designed for our situation

- Proposed attack achieves full-key recovery of 1,024-bit and 2,048 RSA–CRT implementations
  - Experimentally demonstrated on GMP implementation
    - Major multiprecision arithmetic library, used in cryptographic libraries
      - OpenSSL has option to adopt it in back-end
      - Can be used on embedded microcontroller
  - Applicable to (stand-alone) OpenSSL, Botan, and ligcrypt
RSA cryptosystem

Plaintext: \( m \), Ciphertext: \( c \),
Public key: \((e, N)\), Secret key: \((p, q, d)\),
\[ N = pq, \quad ed = 1 \mod N \]

- Encryption:
  \[ c = m^e \mod N \]
- Decryption:
  \[ m = c^d \mod N \]

Nice math!

But how to implement it efficiently and securely?
Open-source RSA implementations

- **Chinese remainder theorem (CRT)** is used in decryption/signing

\[ m_p = c^{d_p} \mod p, \quad m_q = c^{d_q} \mod q, \quad m = p^{-1}(m_q - m_p) \mod q \]

**Secret key:** \((p, q, d_p, d_q, p^{-1})\),
\[ N = pq, \quad ed = 1 \mod N, \quad d_p = d \mod p, \quad d_q = d \mod q \]

- Yields 2–4 times faster computation

- Exponentiation algorithm mainly determines the performance
  - Open-source software (OSS) usually employ windowed exponentiation

<table>
<thead>
<tr>
<th>Exponentiation algorithm</th>
<th>Relation to S–M seq.</th>
<th>Execution time</th>
<th>Examples of OSS adoption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left-to-right binary</td>
<td>Exponent-dependent, and bijective to exponent.</td>
<td>Non-constant, slow</td>
<td>None</td>
</tr>
<tr>
<td>Square–multiply always</td>
<td>Exponent-independent</td>
<td>Constant, slow</td>
<td>None</td>
</tr>
<tr>
<td>Montgomery ladder</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed window</td>
<td>Exponent-independent</td>
<td>Constant, fast</td>
<td>GMP, OpenSSL, WolfCrypt, etc.</td>
</tr>
<tr>
<td>Sliding window</td>
<td>Exponent-dependent, but not bijective to exponent</td>
<td>Non-constant, fast</td>
<td>libgcrypt, Gnu TLS, Bouncy Castle, etc.</td>
</tr>
</tbody>
</table>

Secret key: \((p, q, d_p, d_q, p^{-1})\),
\[ N = pq, \quad ed = 1 \mod N, \quad d_p = d \mod p, \quad d_q = d \mod q \]
Fixed window exponentiation $m = c^d \mod N$

- Fastest constant-time exponentiation (let $w$ be window size)
  - Precomputation: Calculate $c^i$ for $i = 0$ to $2^w - 1$ and make table where $\text{table}[i] = c^i$
  - Main loop: Perform squaring $w$ times and then multiplication with $\text{table}[i]$
    - $i$ is temporal window value

<table>
<thead>
<tr>
<th>Temporal window value</th>
<th>1101</th>
<th>1110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square-Multiply sequence</td>
<td>SSSSM</td>
<td>SSSSM</td>
<td>SSSSM</td>
</tr>
</tbody>
</table>

Example of $d = (110111100111)_2$ and $w = 4$

- $m \leftarrow (((1^2)^2)^2 \times c^{13}$
- $m \leftarrow (((m^2)^2)^2 \times c^{14}$
- $m \leftarrow (((m^2)^2)^2 \times c^{7}$

- SCA security?
  - Secure against SPA (square–multiply sequence is independent of exponent)
  - Leakage of temporal window values (loaded table address) yields key recovery
    - Prime+Probe, address bit DPA, collision analysis, etc.
    - *Leakage/security of operand loading should be considered*
Many windowed exponentiation in OSS employ dummy load

- All operands in precomputation table are accessed in every multiplication

Operand loading in GMP
(addr is temporal window value)

Function LoadOperand(addr);

\[ \text{mask} \leftarrow \neg (i = \text{addr}); \]

\[ s \leftarrow \text{or} (s, \neg \text{mask}), \text{and} (\text{table}[i], \text{mask}); \]

return \( s \)

Windowed exponentiation + dummy load seems sufficient to counter known remote timing/cache attacks

But how about power/EM analyses?
Overview of proposed attack

Step 1: Profiling (DL training phase)

Step 2: Temporal window value estimation (DL attack phase)

Step 3: Partial key exposure attack

DL-SCA

Step 1: Acquire traces for NN training and training NN

Step 2: Temporal value inference from attack traces by NN inference

- We develop very efficient methodology (specify what to learn) via in-depth analyses on implementation

Step 3: Full-key recovery via secret key leakage obtained in Step 2

- Estimated secret exponents may not be completely correct
- New partial key exposure attack dedicated to our methodology
An operand loading consists of one true load and \(2^w-1\) dummy loads

- Value of register \(s\) is changed only when true load
  - Possibility of distinguishing true/dummy load by its physical side-channels
- Order of true and dummy loads fully depends on temporal window value
  - True/dummy load sequence is one-hot coding of temporal window value

Proposed methodology: **One truth prevails**

![Function LoadOperand(addr);](function)

<table>
<thead>
<tr>
<th>Temporal window value</th>
<th>1101</th>
<th>1110</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square-multiply sequence</td>
<td>SSSSM</td>
<td>SSSSM</td>
<td>SSSSM</td>
</tr>
<tr>
<td>True/dummy load sequence</td>
<td>DDDDDDDDDDTDD</td>
<td>DDDDDDDDDDTD</td>
<td>DDDDDDDDTDDDD</td>
</tr>
</tbody>
</table>

Example of \(d = (110111100111)_2\) and \(w = 4\)

Distinguishing true/dummy load yields temporal window value recovery
How to distinguish true/dummy load: DL-SCA

Employ two-classification NN to distinguish true/dummy load

Training phase:
- Train NN using traces labeled as true or dummy load (from profiling device)

Attack phase:
- Perform $2^w$ two-classifications to distinguish true/dummy load
- Estimate load operation with highest probability of true load as the true load
  (Take argmax of NN outputs)

NN inference is reduced to two-classification from $2^w$-classification
- Improve NN accuracy and reduce learning cost, which yields efficient attack
New partial key exposure attack

- Heninger–Shacham attack: Random bit leak
  - Inapplicable to our scenario

- Henecka et al.’s attack: Random bit flip
  - Computational cost grows exponentially by maximum length of consecutive bit errors

- Our attack: $w$-bit wise error
  - Utilize heuristics and priority deque to correct errors in $w$-bit wise manner
  - Heuristics determine cost of each key candidate due to inconsistent bit obtained from side-channels
    - Unlikely candidates are efficiently prone

\[
p = \cdots \text{XX}1, \quad q = \cdots \text{XX}1, \quad d_p, d_q
\]
Experimental evaluation

- Evaluate accuracy of temporal window value estimation on 1,024-bit RSA–CRT implementation with GMP
  - 1,024-bit RSA–CRT = $128 \times 2$ temporal window value estimations ($w = 4$)
  - Training trace dataset: 61,440,000 EM traces for true and dummy loads
  - Profiling and target device: ARM Cortex-M4 with 168 MHz frequency

![EM trace of true load](image1)

![EM trace of dummy load](image2)

Convolutional NN used in experiment
(6 convolutional layer followed by 2 fully connected layers)
Result (without partial key exposure attack)

- Evaluate test phase accuracy (attack success rate) using 24 different secret keys
  - We estimated 48 exponents, $48 \times 128$ temporal window values, and $48 \times 128 \times 16$ true/dummy loads ($w = 4$)
  - Success rate is sufficient to break exponent-blinded RSA–CRT if multiple traces are available

<table>
<thead>
<tr>
<th>True/dummy load</th>
<th>Temporal window value</th>
<th>Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed DL-SCA</td>
<td>99.94%</td>
<td>99.82%</td>
</tr>
<tr>
<td>Template attack</td>
<td>79.17%</td>
<td>4.16%</td>
</tr>
<tr>
<td>$2^w$-classification NN</td>
<td>N/A</td>
<td>11.53%</td>
</tr>
</tbody>
</table>

- Number of estimation errors is at most two

<table>
<thead>
<tr>
<th># Errors</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 &gt;</td>
<td>0</td>
</tr>
</tbody>
</table>
Generate 100 random RSA–CRT secret keys with $w$-bit-wise errors and apply proposed partial key exposure attack

- Simulate errors included in secret keys estimated by proposed DL-SCA

Proposed attack can recover full key with 100% success rate

- A few seconds when # errors is 2
- A dozen of seconds when it is 10
- Success rate of Henecka et al.’s attack was at most 80%
  - Our attack is well-calibrated for our DL-SCA ($w$-bit-wise error)
Concluding remarks

- New DL-SCA and partial key exposure attack on RSA–CRT
  - Applicable to practical implementations with windowed exponentiation and dummy load as hiding countermeasure
    - Utilized in, for example, GMP, OpenSSL, libgcrypt, and Botan
  - Experimentally confirmed full-key recovery of 1,024- and 2,048-bit RSA–CRT
  - Countermeasure: randomizations of initial register value and loading order
    (See our paper for concrete algorithm)

- DL can offer strong attacks even if detail of implementation is not known, but can achieve stronger attack if it is available
Many existing DL-SCAs focus on binary exponentiation
- Left-to-right, Montgomery ladder, square–multiply always, etc.
- Two-classification NN is used to directly estimate secret exponent
  - Its feasibility and accuracy have been studied

Natural extension to windowed exponentiation: $2^w$-classification NN
- But its feasibility is unclear in general
  - $2^w$-classification NN would be more difficult task than two-classification
  - Hiding countermeasure would make classification more difficult
    - $2^w$-classification on WolfSSL EdDSA implementation in [WCBP20], but it neither employs hiding countermeasure nor protects operand loading
  - In our experiment, $2^w$-classification NN achieved only 11.52% accuracy on Gnu MP implementation, which would be insufficient for key recovery