Sliding right into disaster - Left-to-right sliding windows leak

Daniel J. Bernstein, Joachim Breitner, Daniel Genkin,
Leon Groot Bruinderink, Nadia Heninger, Tanja Lange,
Christine van Vredendaal and Yuval Yarom

September 28th, 2017

Side-channel attacks on RSA

- Side-channel attacks on RSA: modular exponentiation
- Constant-time implementations cannot use sliding windows
- Common belief: sliding windows do not leak enough for key recovery

This work

- We show that right-to-left sliding window method does not leak enough

This work

- We show that right-to-left sliding window method does not leak enough
- We show that left-to-right sliding window method does leak enough
- Two methods to extract information from square and multiply sequence
- Demonstrated real-world applicability by attacking Libgcrypt
- We analyze the reasons why left-to-right leaks more than right-to-left

RSA

RSA signatures

Keygen:

- Public key (e, N) where $N = pq$ for primes p, q
- Secret key (d, p, q) where $ed \equiv 1 \bmod \phi(N)$ and $\phi(N) = (p-1)(q-1)$

RSA signatures

Keygen:

- Public key (e, N) where $N = pq$ for primes p, q
- Secret key (d, p, q) where $ed \equiv 1 \bmod \phi(N)$ and $\phi(N) = (p-1)(q-1)$

Sign and verify:

- Let H be a padded secure hash-function
- Signature: s of message m: $s = H(m)^d \bmod N$
- Verification: compute $z = s^e \bmod N$ and verify $z \stackrel{?}{=} H(m)$

RSA signatures

Keygen:

- Public key (e, N) where $N = pq$ for primes p, q
- Secret key (d, p, q) where $ed \equiv 1 \bmod \phi(N)$ and $\phi(N) = (p-1)(q-1)$

Sign and verify:

- Let H be a padded secure hash-function
- Signature: s of message m: $s = H(m)^d \bmod N$
- Verification: compute $z = s^e \bmod N$ and verify $z \stackrel{?}{=} H(m)$

CRT:

- Common optimization based on Chinese Remainder Theorem (CRT)
- Compute $s_p \equiv H(m)^{d_p} \bmod p$ and $s_q \equiv H(m)^{d_q} \bmod q$
- Combine to s using CRT

Sliding-window method

- Implement modular exponentiation using sliding-windows
- Window size w, sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$ for odd $0 \leq d_i \leq 2^w - 1$
- In general, compute $b^d \bmod p$ as follows:

Sliding-window method

- Implement modular exponentiation using sliding-windows
- Window size w, sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$ for odd $0 \le d_i \le 2^w - 1$
- In general, compute $b^d \bmod p$ as follows:
  1. Precompute small, **odd** powers of $b \bmod p$ (i.e. $b \bmod p, b^3 \bmod p, \ldots, b^{2^w-1} \bmod p$).

Sliding-window method

- Implement modular exponentiation using sliding-windows
- Window size w, sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$ for odd $0 \leq d_i \leq 2^w - 1$
- In general, compute $b^d \bmod p$ as follows:
  1. Precompute small, **odd** powers of $b \bmod p$ (i.e. $b \bmod p, b^3 \bmod p, \dots, b^{2^w-1} \bmod p$).
  2. Set $a = 1$
  3. For $i \leftarrow n-1$ to 0:
  4. $\quad a = a \cdot a \bmod p$ (Square)
  5. $\quad$ If $d_i \neq 0$:
  6. $\quad\quad a = a \cdot b^{d_i} \bmod p$ (Multiply)
  7. Return a

Sliding-window method

- Implement modular exponentiation using sliding-windows
- Window size w, sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$ for odd $0 \le d_i \le 2^w - 1$
- In general, compute $b^d \bmod p$ as follows:
  1. Precompute small, **odd** powers of $b \bmod p$ (i.e. $b \bmod p, b^3 \bmod p, \dots, b^{2^w-1} \bmod p$).
  2. Set $a = 1$
  3. For $i \leftarrow n-1$ to 0:
  4. $\quad a = a \cdot a \bmod p$ (Square)
  5. $\quad$ If $d_i \neq 0$:
  6. $\qquad a = a \cdot b^{d_i} \bmod p$ (Multiply)
  7. Return a
- This leaks a Square and Multiply Sequence
- For sufficiently large w, too many options to try

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$

Sliding-window form

- How to compute sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form														
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form										0	0	0	3
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form									0	0	0	0	3
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form						0	0	0	11	0	0	0	0	3
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form		0	0	0	1	0	0	0	11	0	0	0	0	3
Binary form	$\boxed{1}$	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form	1	0	0	0	1	0	0	0	11	0	0	0	0	3
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

- Leaking on average a fraction of $\frac{2}{w+1}$ bits

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form														
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form

Binary form $\boxed{1}$ 0 0 0 1 1 0 1 1 0 0 0 1 1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1													
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0												
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0											
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0	0										
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0	0	0	0	0	13						
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0	0	0	0	0	13	1					
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0	0	0	0	0	13	1	0	0	0		
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

Sliding-window form

- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0	0	0	0	0	13	1	0	0	0	0	3
Binary form	1	0	0	0	1	1	0	1	1	0	0	0	1	1

- Enables on-the-fly encoding and exponentiation
- Not obvious how many bits are leaking...

Sliding Right versus Sliding Left Analysis

First observations

- Right-to-left: guaranteed $w - 1$ zero bits after non-zero bit
- Left-to-right: two non-zero bits can be as close as adjacent
- This allows for many more recovered bits from Square and Multiply sequence
- First method: deduce more known bits from 4 bit recovery rules
- Second method: uses knowledge not directly translatable to known bits

Applying bit recovery rules

- $d = 9059 \rightarrow S = smssssssssmsmssssssm$ with $w = 4$
- Convert $sm \rightarrow \underline{x}$, $s \rightarrow x$

$$D_1 = \underline{x}xxxxxx\underline{x}\underline{x}xxxx\underline{x}$$

Applying bit recovery rules

- $d = 9059 \rightarrow S = smsssssssmsmsssssm$ with $w = 4$
- Convert $sm \rightarrow \underline{x}$, $s \rightarrow x$

$$D_1 = \underline{x}xxxxxx\underline{x}\underline{x}xxxx\underline{x}$$

- **Rule 0: Multiplication bits $\underline{x} \rightarrow \underline{1}$**

$$D_2 = \underline{1}xxxxxx\underline{1}\underline{1}xxxx\underline{1}$$

Applying bit recovery rules

- $d = 9059 \rightarrow S = smsssssssmsmsssssm$ with $w = 4$
- Convert $sm \rightarrow \underline{x}$, $s \rightarrow x$

$$D_1 = \underline{x}xxxxxx\underline{x}\underline{x}xxxx\underline{x}$$

- **Rule 0: Multiplication bits** $\underline{x} \rightarrow \underline{1}$

$$D_2 = \underline{1}xxxxxx\underline{1}\underline{1}xxxx\underline{1}$$

- **Rule 1: Trailing zeros** $\underline{1}x^i\underline{1}x^{w-i-1} \rightarrow \underline{1}x^i\underline{1}0^{w-i-1}$

$$D_3 = \underline{1}xxxxxx\underline{1}\underline{1}000x\underline{1}$$

Applying bit recovery rules

- $d = 9059 \rightarrow S = smssssssssmsmssssssm$ with $w = 4$
- Convert $sm \rightarrow \underline{x}$, $s \rightarrow x$

$$D_1 = \underline{x}xxxxxx\underline{x}\underline{x}xxxx\underline{x}$$

- **Rule 0: Multiplication bits** $\underline{x} \rightarrow \underline{1}$

$$D_2 = \underline{1}xxxxxx\underline{1}\underline{1}xxxx\underline{1}$$

- **Rule 1: Trailing zeros** $\underline{1}x^i\underline{1}x^{w-i-1} \rightarrow \underline{1}x^i\underline{1}{\color{blue}0^{w-i-1}}$

$$D_3 = \underline{1}xxxxxx\underline{1}\underline{1}{\color{blue}000}x\underline{1}$$

- **Rule 2: Leading one** $xxx\underline{1}\underline{1} \rightarrow {\color{red}1}xx\underline{1}\underline{1}$

$$D_4 = \underline{1}xxx{\color{red}1}xx\underline{1}\underline{1}{\color{blue}000}x\underline{1}$$

Applying bit recovery rules

- $d = 9059 \rightarrow S = smsssssssmsmssssssm$ with $w = 4$
- Convert $sm \rightarrow \underline{x}$, $s \rightarrow x$

$$D_1 = \underline{x}xxxxxx\underline{x}\underline{x}xxxx\underline{x}$$

- **Rule 0: Multiplication bits** $\underline{x} \rightarrow \underline{1}$

$$D_2 = \underline{1}xxxxxx\underline{1}\underline{1}xxxx\underline{1}$$

- **Rule 1: Trailing zeros** $\underline{1}x^i\underline{1}x^{w-i-1} \rightarrow \underline{1}x^i\underline{1}0^{w-i-1}$

$$D_3 = \underline{1}xxxxxx\underline{1}\underline{1}000x\underline{1}$$

- **Rule 2: Leading one** $xxx\underline{1}\underline{1} \rightarrow 1xx\underline{1}\underline{1}$

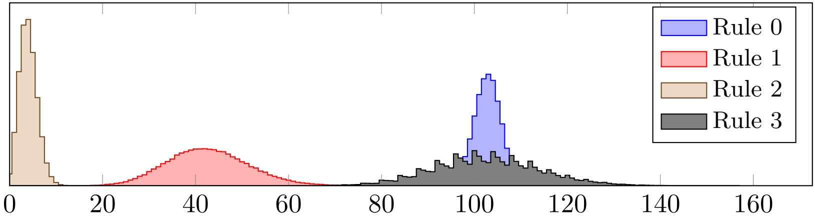
$$D_4 = \underline{1}xxx1xx\underline{1}\underline{1}000x\underline{1}$$

- **Rule 3: Leading zeros** $\underline{1}x^ix^{w-1}\underline{1} \rightarrow \underline{1}0^ix^{w-1}\underline{1}$

$$D_5 = \underline{1}0001xx\underline{1}\underline{1}000x\underline{1}$$

Results of using bit recovery rules

- Conform Libgcrypt's implementation of RSA-1024: $n = 512, w = 4$



Distribution of number of recovered bits per rule

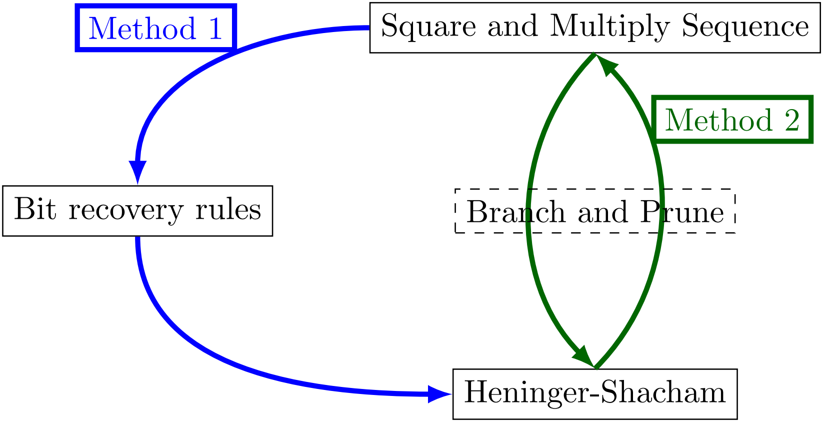
Results of using bit recovery rules

- Heninger-Shacham: branch and prune candidate solutions given partial information on RSA keys
- Requires $> 50\%$ known bits for efficient attack
- For $n = 512, w = 4$, we recover more than 50% of the bits in 32% of the time

Direct pruning from Square and Multiplies

- Bit recovery rules did not give enough known bits for $n = 1024, w = 5$ to succeed (conform RSA-2048)
- Method 2: directly branch and prune search tree of Heninger-Shacham from Square and Multiply sequence

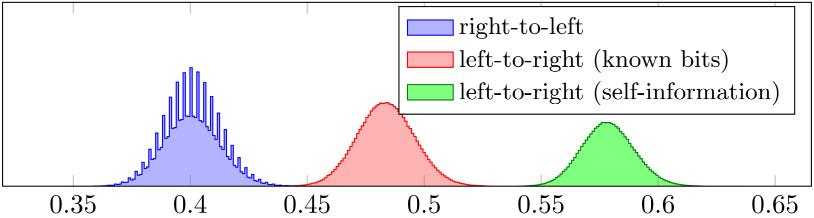
Direct pruning from Square and Multiplies



Recovery methods RSA Square and Multiply Sequence

Direct pruning from Square and Multiplies

- Summary of results for RSA-1024:

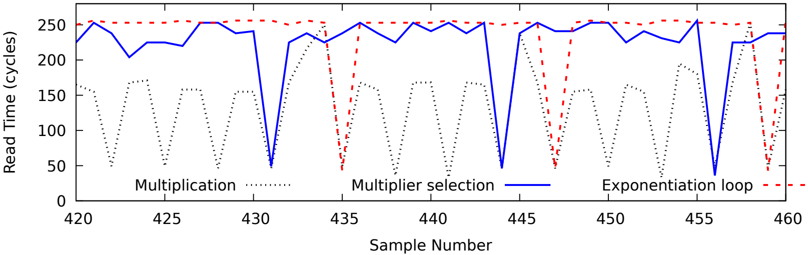


Distribution of information recovered ($w = 4$)

- Direct pruning allows to recover RSA-2048 bit keys 13% of the time

Attacking Libgcrypt

- Demonstrated vulnerability in Libgcrypt (fixed in version 1.7.8)
- Flush+Reload cache-attack using Mastik toolkit



Libgcrypt Activity Trace

A lot more in the paper!

- Theoretical analysis of bit-recovery rules using Renewal Reward processes
- Theoretical analysis of direct pruning using self-information and collision entropy
- More experimental results and details
- Full version online: *https://eprint.iacr.org/2017/627*

A lot more in the paper!

- Theoretical analysis of bit-recovery rules using Renewal Reward processes
- Theoretical analysis of direct pruning using self-information and collision entropy
- More experimental results and details
- Full version online: *https://eprint.iacr.org/2017/627*

Thank you for your attention
Questions?