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

November 17th, 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

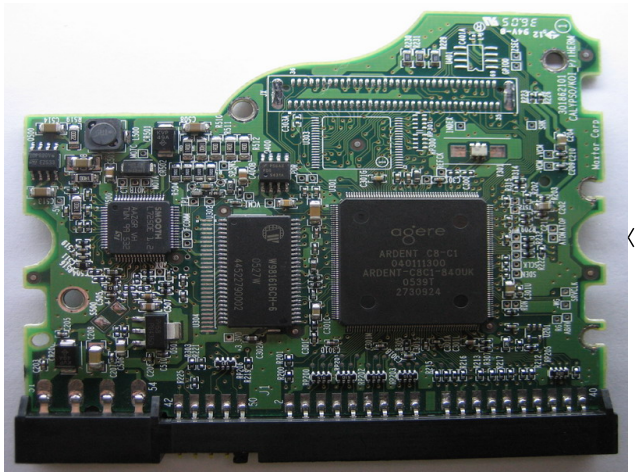
This work

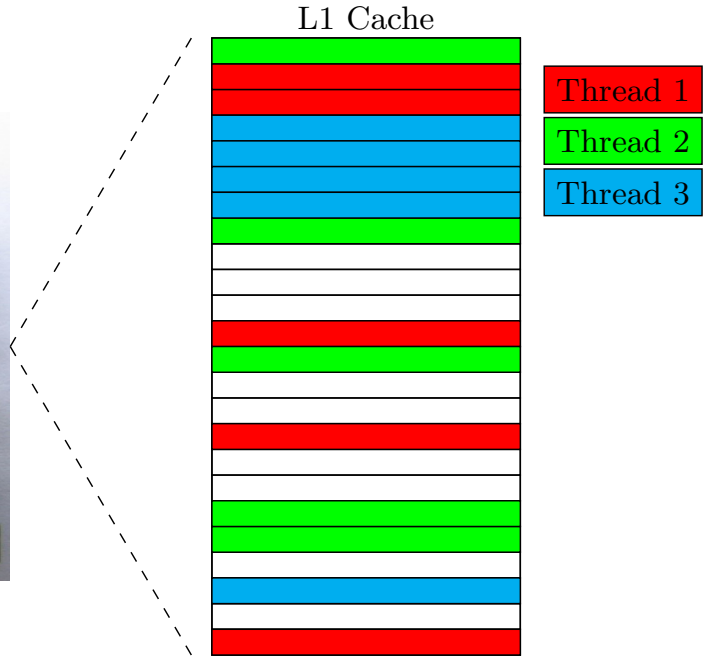
- We show that right-to-left sliding window method does not leak enough
- We show that left-to-right sliding window method does leak enough
- Two methods to extract information from square and multiply sequence
- Demonstrated real-world applicability by attacking Libgcrypt
- We analyze the reasons why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

Cache Timing Attacks

Cache (Timing) Attacks

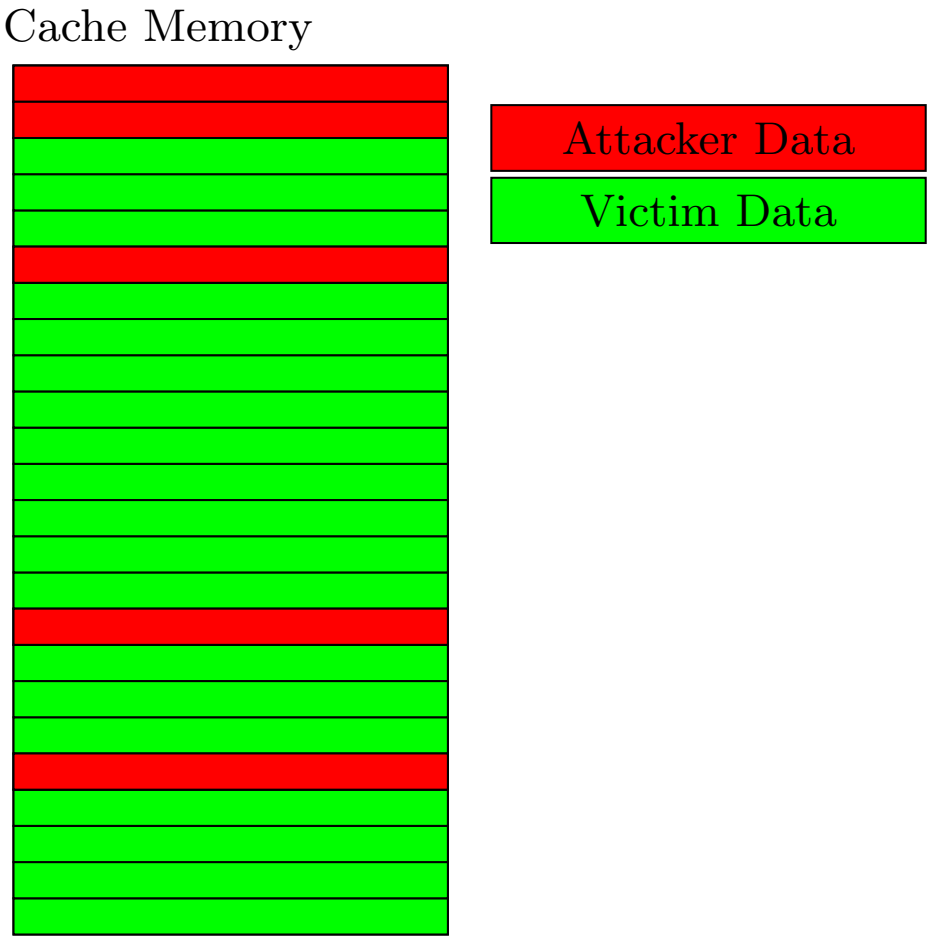
- Cache-memory: small, fast memory shared among all threads.
- Bridge the gap between processor speed and memory speed.
- Data is stored in cache-lines, typically 64 Bytes.





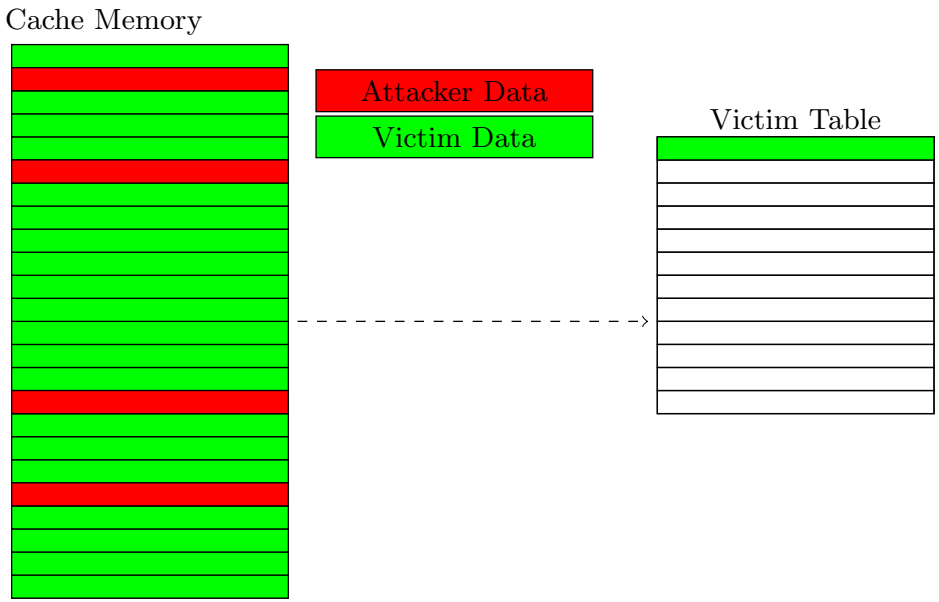
Cache (Timing) Attacks

- Attacker fills specific cache lines with his data.



Cache (Timing) Attacks

- Attacker notices that victim uses some part of cache.
- Learns cache-line of data used by victim.



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 $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$ for $d_i = 0$ or odd $1 \leq d_i \leq 2^w - 1$
- In general, compute b^d mod p as follows:

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 $d_i = 0$ or odd $1 \leq d_i \leq 2^w - 1$
- In general, compute $b^d \bmod p$ as follows:
  1. Precompute small, **odd** powers of $b \bmod p$ (i.e. $b \bmod p, b^3 \bmod p, \dots, b^{2^w-1} \bmod p$).

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 $d_i = 0$ or odd $1 \leq d_i \leq 2^w - 1$
- 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 $d_i = 0$ or odd $1 \leq d_i \leq 2^w - 1$
- 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. $a = a \cdot a \bmod p$ (Square)
  5. If $d_i \neq 0$:
  6. $a = a \cdot b^{d_i} \bmod p$ (Multiply)
  7. Return a
- This leaks a Square and Multiply Sequence
- For sufficiently large w, too many options to try

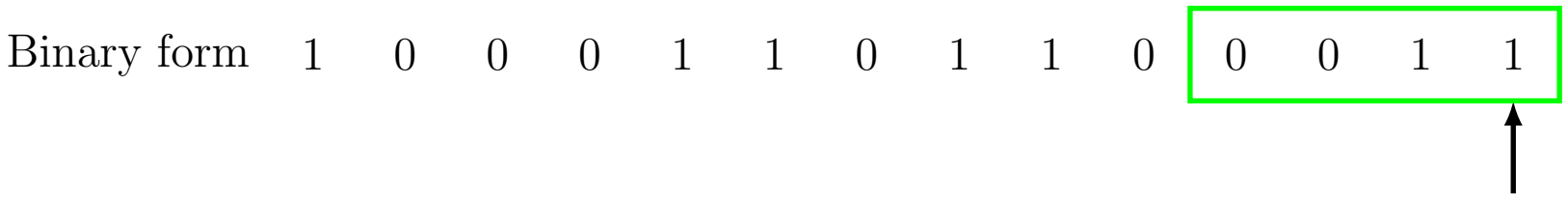
Sliding-window form

- 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

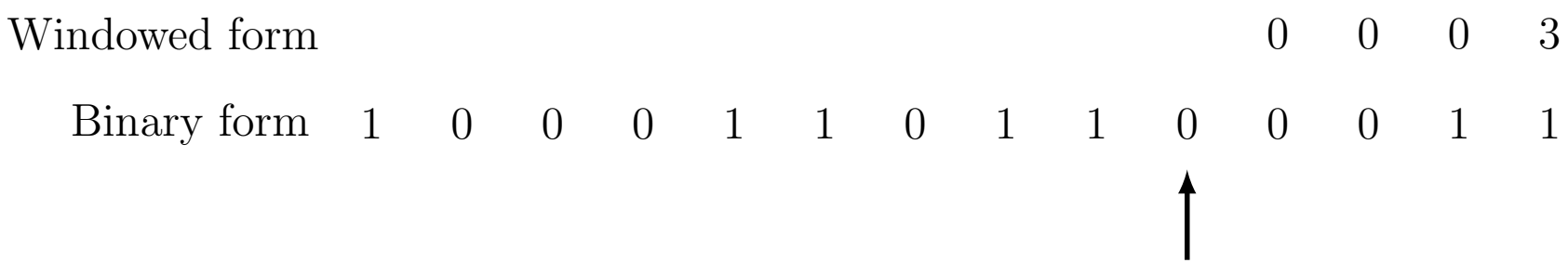
- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

Windowed form



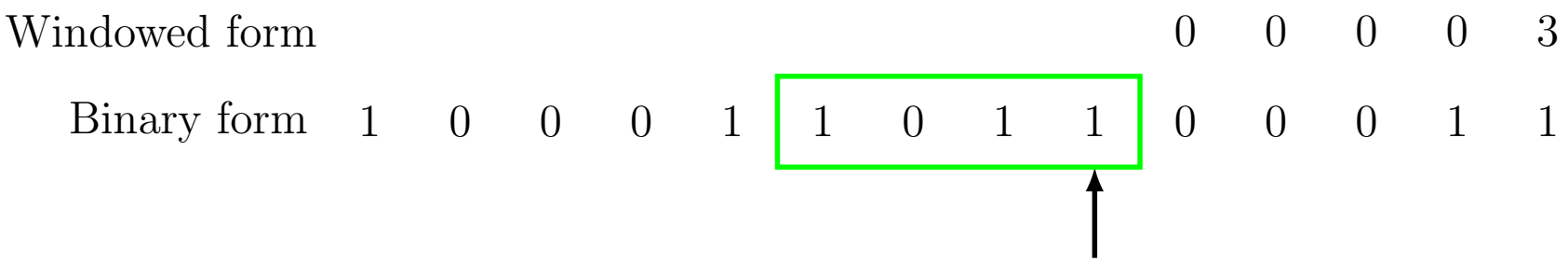
Sliding-window form

- 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



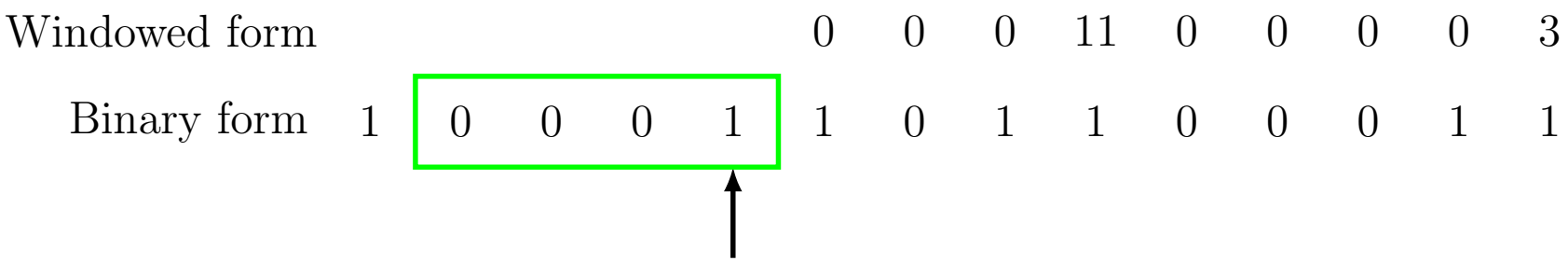
Sliding-window form

- 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



Sliding-window form

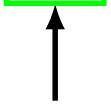
- How to compute sliding-window form $d_{n-1} \ldots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left



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	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



Sliding-window form

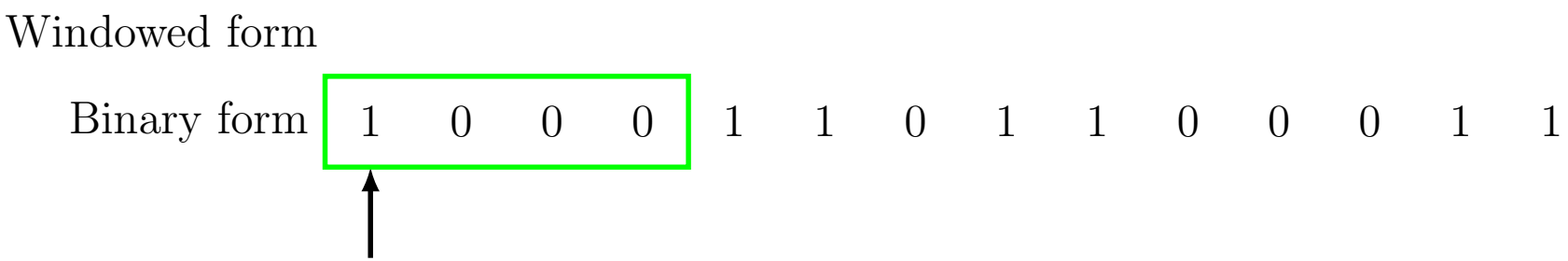
- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Right-to-left

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

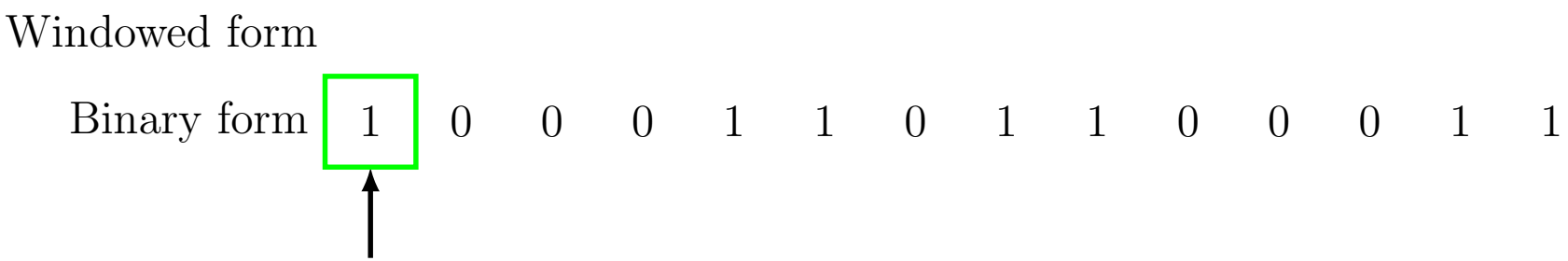
Sliding-window form

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



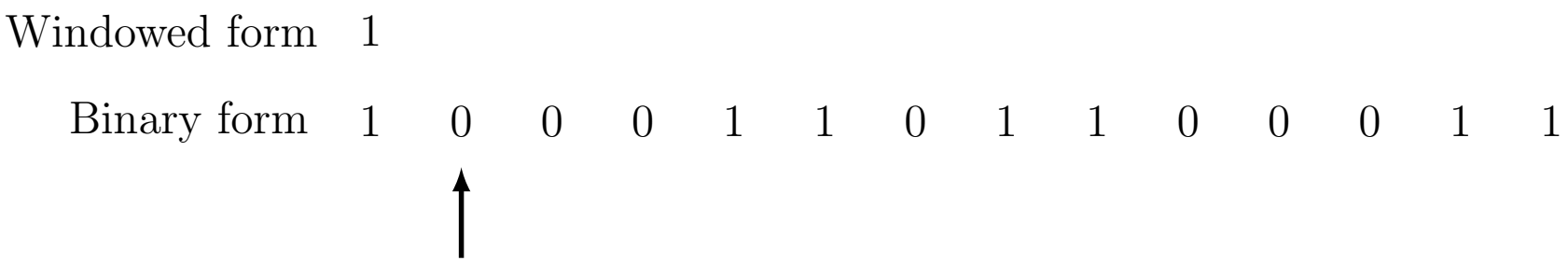
Sliding-window form

- 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



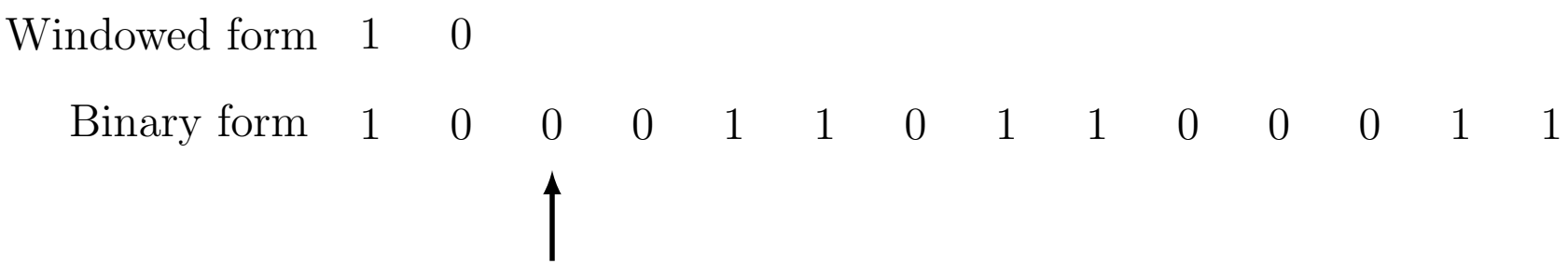
Sliding-window form

- 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



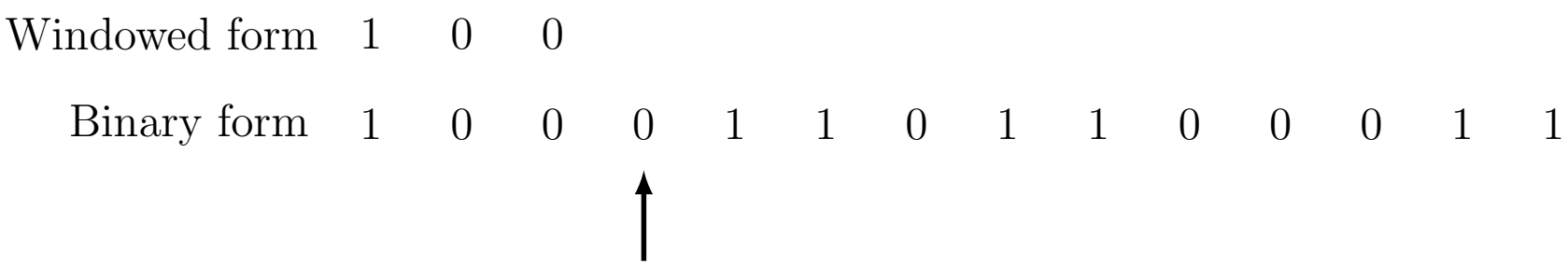
Sliding-window form

- 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



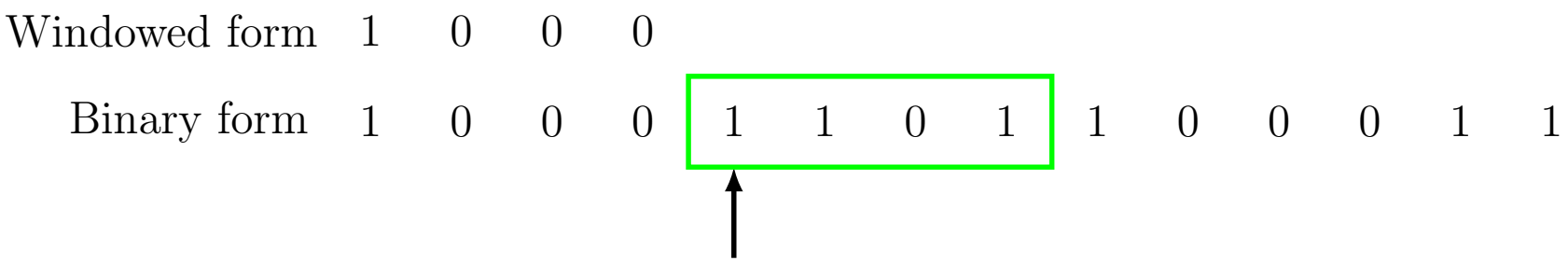
Sliding-window form

- 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



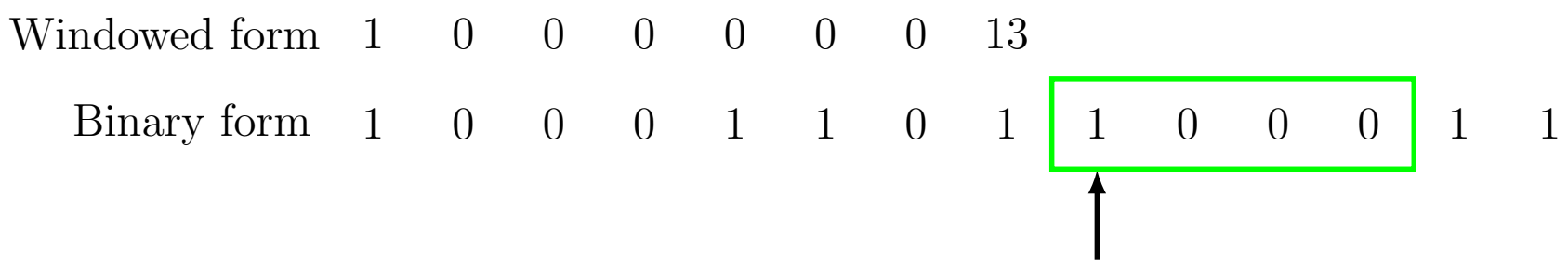
Sliding-window form

- 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



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



Sliding-window form

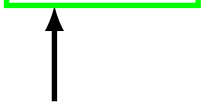
- How to compute sliding-window form $d_{n-1} \dots d_0$ s.t. $d = \sum_{i=0}^{n-1} d_i 2^i$
- Example with $w = 4$, $d = 9059 = 10001101100011$
- Left-to-right

Windowed form	1	0	0	0	0	0	0	13	1					
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	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 indices after non-zero index
- Left-to-right: two non-zero indices 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

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

$$D_1 = \underline{x}xxxxxx\underline{xx}xxxxx\underline{x}$$

Applying bit recovery rules

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

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

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

$$D_2 = \underline{1}xxxxxx\underline{11}xxxx\underline{1}$$

Applying bit recovery rules

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

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

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

$$D_2 = \underline{1}xxxxxx\underline{11}xxxx\underline{1}$$

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

$$D_3 = \underline{1}xxxxxx\underline{11}000x\underline{1}$$

Applying bit recovery rules

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

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

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

$$D_2 = \underline{1}xxxxxx\underline{11}xxxx\underline{1}$$

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

$$D_3 = \underline{1}xxxxxx\underline{11}000x\underline{1}$$

- **Rule 2: Leading one** $xxx\underline{11} \rightarrow 1xx\underline{11}$

$$D_4 = \underline{1}xxx1xx\underline{11}000x\underline{1}$$

Applying bit recovery rules

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

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

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

$$D_2 = \underline{1}xxxxxx\underline{11}xxxx\underline{1}$$

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

$$D_3 = \underline{1}xxxxxx\underline{11}000x\underline{1}$$

- **Rule 2: Leading one** $xxx\underline{11} \rightarrow 1xx\underline{11}$

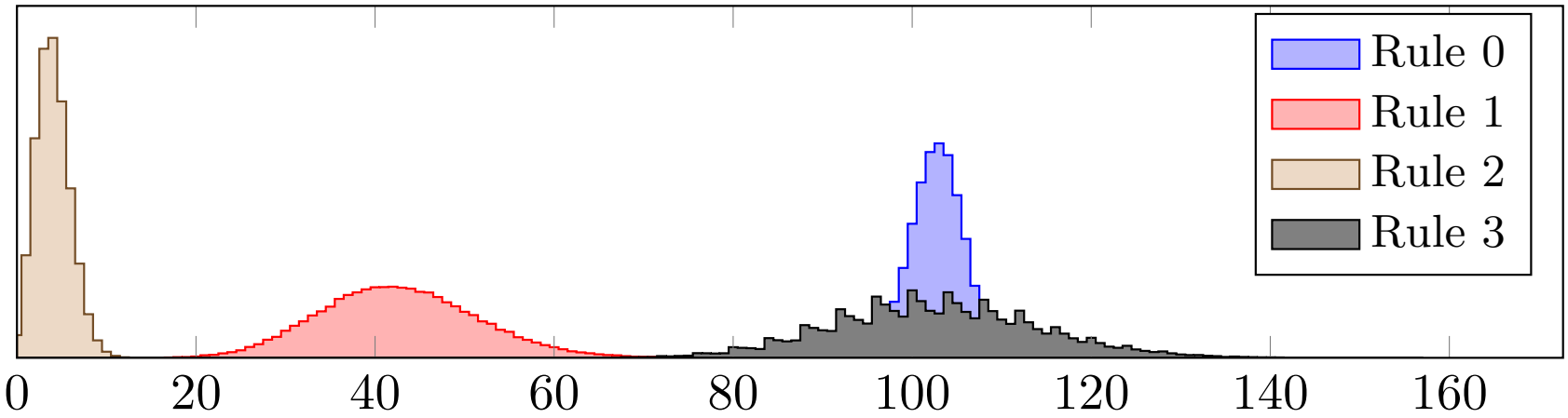
$$D_4 = \underline{1}xxx1xx\underline{11}000x\underline{1}$$

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

$$D_5 = \underline{1}0001xx\underline{11}000x\underline{1}$$

Results of using bit recovery rules

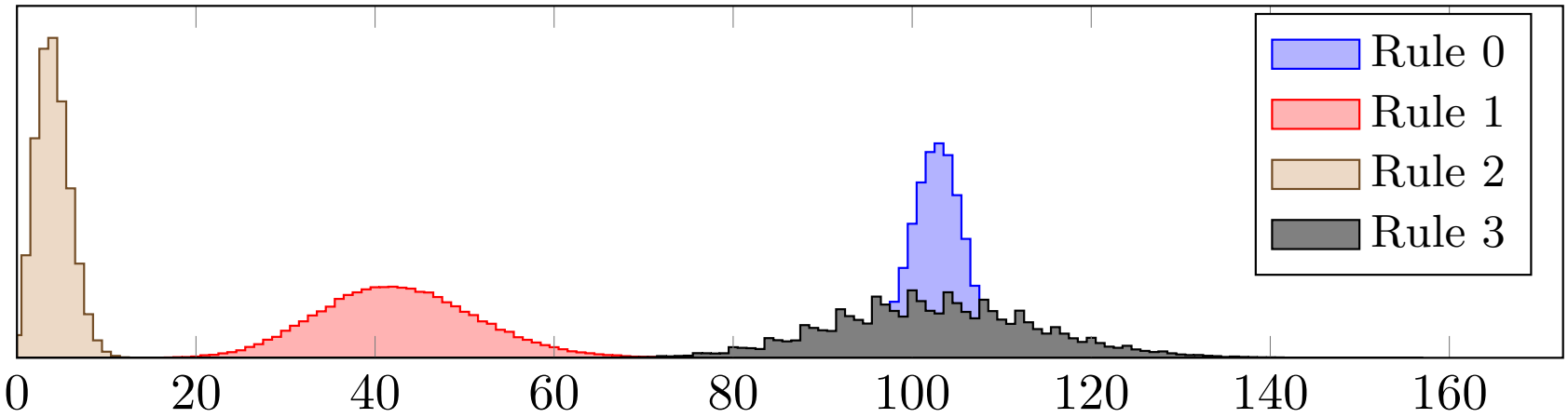
- Conform Libgcrypt's implementation of RSA-1024: $n = 512, w = 4$



Distribution of number of recovered bits per rule

Results of using bit recovery rules

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



Distribution of number of recovered bits per rule

- Is this the best we can do? No!

A small example where bitrecovery misses a bit of bits

- Example with $d = 10010001000100010001010000010001$ and $w = 4$
- $S = smsssmsssssmsssssmsssssmsssssmssmsssssssmssssm$

A small example where bitrecovery misses a bit of bits

- Example with $d = 10010001000100010001010000010001$ and $w = 4$
- $S = smsssmsssssmsssssmsssssmsssssmssmsssssssmssssm$
- After applying rules 0-3, we get:
  $\underline{1}xx\underline{1}0xx\underset{a}{\underline{1}}xxx\underset{\uparrow}{}\underset{b}{\underline{1}}xxx\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$

A small example where bitrecovery misses a bit of bits

- Example with $d = 10010001000100010001010000010001$ and $w = 4$
- $S = smsssmsssssmsssssmsssssmsssssmssmsssssssmssssm$
- After applying rules 0-3, we get:
  $\underline{1}xx\underline{1}0xx\underset{a}{\underline{1}}xxx\underset{\uparrow}{x}\underset{b}{\underline{1}}xxx\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$
- Let's try $\underset{\uparrow}{x} = 0$:
- $\underline{1}xx\underline{1}0xx\underset{a}{\underline{1}}x\underset{\uparrow}{0}x\underset{b}{\underline{1}}xxx\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$

A small example where bitrecovery misses a bit of bits

- Example with $d = 10010001000100010001010000010001$ and $w = 4$
- $S = smsssmsssssmsssssmsssssmsssssmssmsssssssmssssm$
- After applying rules 0-3, we get:
  $\underline{1}xx\underline{1}0xx\underset{a}{\underline{1}}\underset{\uparrow}{x}xx\underset{b}{\underline{1}}xxx\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$
- Let's try $\underset{\uparrow}{x} = 0$:
- $\underline{1}xx\underline{1}0xx\underset{a}{\underline{1}}x\underset{\uparrow}{0}x\underset{b}{\underline{1}}00x\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$

A small example where bitrecovery misses a bit of bits

- Example with $d = 10010001000100010001010000010001$ and $w = 4$
- $S = smsssmsssssmsssssmsssssmsssssmssmsssssssmssssm$
- After applying rules 0-3, we get:
  $\underset{a}{\underline{1}}xx\underline{1}0xx\underset{a}{\underline{1}}xxx\underset{b}{\underline{1}}xxx\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$
- Let's try $\underset{\uparrow}{x} = 0$:
- $\underset{a}{\underline{1}}xx\underline{1}0xx\underset{a}{\underline{1}}x0x\underset{b}{\underline{1}}00x\underset{c}{\underline{1}}00x\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$

A small example where bitrecovery misses a bit of bits

- Example with $d = 10010001000100010001010000010001$ and $w = 4$
- $S = smsssmsssssmsssssmsssssmsssssmssmsssssssmssssm$
- After applying rules 0-3, we get:
  $\underline{1}xx\underline{1}0xx\underset{a}{\underline{1}}xxx\underset{\uparrow}{x}\underset{b}{\underline{1}}xxx\underset{c}{\underline{1}}xxx\underset{d}{\underline{1}}x\underset{e}{\underline{1}}00xxx\underline{1}xxx\underline{1}$
- $\underset{\uparrow}{x}$ has to be a 1!
- These events are rare, but require a more iterative approach

Revisiting multiplier locations

- Idea: keep track of the possible widths for each multiplication
- Suppose multiplication at position $i = 5$ and $w = 4$

Case 1: `x1xx1xxxxx`, multiplier width: 4
Case 2: `xx1x10xxxx`, multiplier width: 3
Case 3: `xxx1100xxx`, multiplier width: 2
Case 4: `xxxx1000xx`, multiplier width: 1

Revisiting multiplier locations

- Idea: keep track of the possible widths for each multiplication
- Suppose multiplication at position $i = 5$ and $w = 4$
- Case 1: `x1xx1xxxxx`, multiplier width: 4
  Case 2: `xx1x10xxxx`, multiplier width: 3
  Case 3: `xxx1100xxx`, multiplier width: 2
  Case 4: `xxxx1000xx`, multiplier width: 1
- Key idea: **multipliers do not overlap!**
- Retrieve smallest m^- and largest m^+ possible width for each multiplier

Revisiting multiplier locations

- Idea: keep track of the possible widths for each multiplication
- Suppose multiplication at position $i = 5$ and $w = 4$
- Case 1: x1xx1xxxxx, multiplier width: 4
  Case 2: xx1x10xxxx, multiplier width: 3
  Case 3: xxx1100xxx, multiplier width: 2
  Case 4: xxxx1000xx, multiplier width: 1
- Key idea: **multipliers do not overlap!**
- Retrieve smallest m^- and largest m^+ possible width for each multiplier
- Gives more known bits:

Input:	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
m^-:				4		3			3			3			3			3		1				1			1	
m^+:				4		3			3			3			3			3		1				3			3	
b_i:	1	x	x	1	1	x	1	0	1	x	1	0	1	x	1	0	1	x	1	0	1	0	0	0	x	x	1	0	x	x	1

Revisiting multiplier locations

- Idea: keep track of the possible widths for each multiplication
- Suppose multiplication at position $i = 5$ and $w = 4$
- Case 1: x1xx1xxxxx, multiplier width: 4
  Case 2: xx1x10xxxx, multiplier width: 3
  Case 3: xxx1100xxx, multiplier width: 2
  Case 4: xxxx1000xx, multiplier width: 1
- Key idea: **multipliers do not overlap!**
- Retrieve smallest m^- and largest m^+ possible width for each multiplier
- Gives more known bits:

Input:	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
m^-:				4		3			3			3			3			3		1				1			1	
m^+:				4		3			3			3			3			3		1				3			3	
b_i:	1	x	x	1	1	x	1	0	1	x	1	0	1	x	1	0	1	x	1	0	1	0	0	0	x	x	1	0	x	x	1	.

- Is this the best we can do? Yes!

Heninger-Shacham key-recovery attack

- Give random bits of $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, how to recover the remainders?

Heninger-Shacham key-recovery attack

- Give random bits of $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, how to recover the remainders?
- Branch and prune over candidate keys using RSA equations:

$$ed_p = 1 + k_p(p-1)$$
$$ed_q = 1 + k_q(q-1)$$

  with $k_p, k_q < e$.
- k_p, k_q initially unknown, but related via

$$(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$$

- Try at most e pairs of k_p, k_q
- Incorrect values for k_p, k_q quickly give no solutions

Heninger-Shacham key-recovery attack

- Give random bits of $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$, how to recover the remainders?
- At i'th least significant bit, we have generated candidate solutions for bits $0, \ldots, i-1$, and then verify that:

$$ed_p = 1 + k_p(p-1) \bmod 2^i$$
$$ed_q = 1 + k_q(q-1) \bmod 2^i$$
$$pq = N \bmod 2^i$$

- Prune a candidate solution if it does not satisfy equations
- Heuristically, this is efficient if $> 50\%$ of the bits are known

Applying Heninger-Shacham after bitrecovery rules

- Conform Libgcrypt's implementation of RSA-1024: $n = 512, w = 4$, we recovered $> 50\%$ of the bits in 32% of the time
- Conform Libgcrypt's implementation of RSA-2048: $n = 1024, w = 5$, 50% was out of reach

Applying Heninger-Shacham after bitrecovery rules

- Conform Libgcrypt's implementation of RSA-1024: $n = 512, w = 4$, we recovered $> 50\%$ of the bits in 32% of the time
- Conform Libgcrypt's implementation of RSA-2048: $n = 1024, w = 5$, 50% was out of reach
- Is this the best we can do? No!

Direct pruning from SM-sequence

- Idea: prune based on Square and Multiply sequence of candidates!
- Given $S_p, S_q \in \{s, m\}^*$ as ground truth sequence for d_p, d_q
- Let $SM(d) = \Sigma$ be the function that maps a bit string d to its Square and Multiply sequence $\Sigma \in \{s, m\}^*$

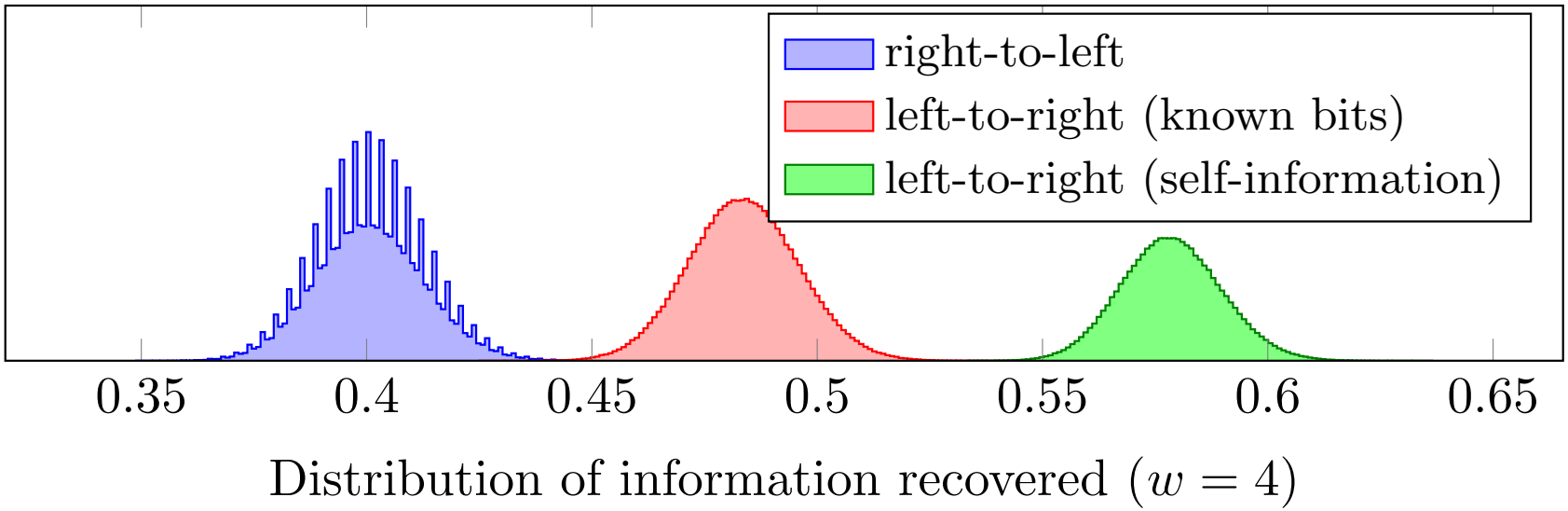
Direct pruning from SM-sequence

- Idea: prune based on Square and Multiply sequence of candidates!
- Given $S_p, S_q \in \{s, m\}^*$ as ground truth sequence for d_p, d_q
- Let $SM(d) = \Sigma$ be the function that maps a bit string d to its Square and Multiply sequence $\Sigma \in \{s, m\}^*$
- Basically the same as original Heninger-Shacham
- Also test i-bit candidate d_i by comparing $\Sigma' = SM(d_i)$ from position 0 through $i - 1$ of S_p, S_q
- Prune d_i if it does not match

Direct pruning from SM-sequence

- Idea: prune based on Square and Multiply sequence of candidates!
- Given $S_p, S_q \in \{s, m\}^*$ as ground truth sequence for d_p, d_q
- Let $SM(d) = \Sigma$ be the function that maps a bit string d to its Square and Multiply sequence $\Sigma \in \{s, m\}^*$
- Basically the same as original Heninger-Shacham
- Also test i-bit candidate d_i by comparing $\Sigma' = SM(d_i)$ from position 0 through $i - 1$ of S_p, S_q
- Prune d_i if it does not match
- Better results: uses information not directly translatable to bits
- Is this the best we can do? Yes!

Summary of results for RSA-1024

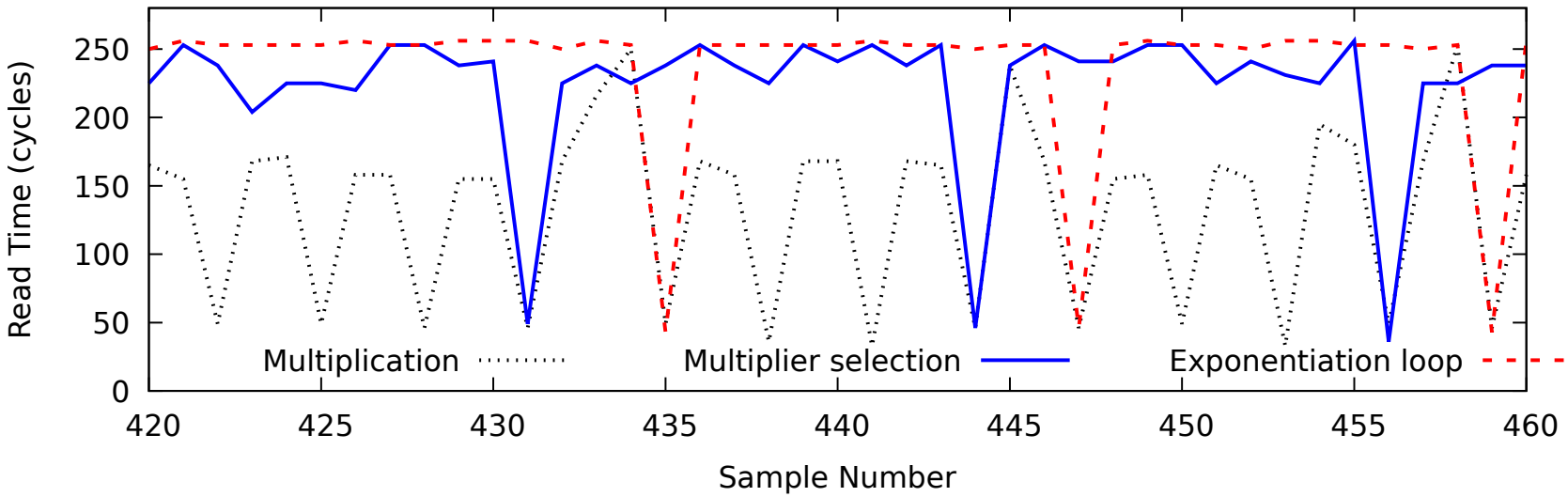


Distribution of information recovered ($w = 4$)

- 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*

Be careful when sliding right...

Questions?