

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

Full version

Daniel J. Bernstein<sup>2</sup>, Joachim Breitner<sup>3</sup>, Daniel Genkin<sup>3,4</sup>,  
Leon Groot Bruinderink<sup>1</sup>, Nadia Heninger<sup>3</sup>, Tanja Lange<sup>1</sup>,  
Christine van Vredendaal<sup>1</sup>, Yuval Yarom<sup>5</sup>

<sup>1</sup> Technische Universiteit Eindhoven, Netherlands

L.Groot.Bruinderink@tue.nl, tanja@hyperelliptic.org,  
c.v.vredendaal@tue.nl

<sup>2</sup> University of Illinois at Chicago, USA

djb@cr.yp.to

<sup>3</sup> University of Pennsylvania, USA

{joachim,danielg3,nadiah}@cis.upenn.edu

<sup>4</sup> University of Maryland, USA

<sup>5</sup> University of Adelaide and Data61, CSIRO, Australia  
yval@cs.adelaide.edu.au

**Abstract.** It is well known that constant-time implementations of modular exponentiation cannot use sliding windows. However, software libraries such as Libgcrypt, used by GnuPG, continue to use sliding windows. It is widely believed that, even if the complete pattern of squarings and multiplications is observed through a side-channel attack, the number of exponent bits leaked is not sufficient to carry out a full key-recovery attack against RSA. Specifically, 4-bit sliding windows leak only 40% of the bits, and 5-bit sliding windows leak only 33% of the bits.

In this paper we demonstrate a complete break of RSA-1024 as implemented in Libgcrypt. Our attack makes essential use of the fact that Libgcrypt uses the left-to-right method for computing the sliding-window expansion. We show for the first time that the direction of the encoding matters: the pattern of squarings and multiplications in left-to-right sliding windows leaks significantly more information about the exponent than right-to-left. We show how to extend the Heninger-Shacham algorithm for partial key reconstruction to make use of this information and obtain a very efficient full key recovery for RSA-1024. For RSA-2048 our attack is efficient for 13% of keys.

This document is the extended version of the CHES 2017 publication [5].

**Keywords:** left-to-right sliding windows, collision entropy, cache attack, Flush+Reload, RSA-CRT.

## 1 Introduction

Modular exponentiation in cryptosystems such as RSA is typically performed starting from the most significant bit (MSB) in a left-to-right manner. More efficient implementations use precomputed values to decrease the number of multiplications. Typically these windowing methods are described in a right-to-left manner, starting the recoding of the exponent from the least significant bit (LSB), leading to the potential disadvantage that the exponent has to be parsed twice: once for the recoding and once of the exponentiation.

This motivated researchers to develop left-to-right analogues of the integer recoding methods that can be integrated directly with left-to-right exponentiation methods. For example, the only method for sliding-window exponentiation in the Handbook of Applied Cryptography [17, Chap 14.6] is the left-to-right version of the algorithm. Doche [8] writes “To enable ‘on the fly’ recoding, which is particularly interesting for hardware applications” in reference to Joye and Yen’s [15] left-to-right algorithm.

Given these endorsements, it is no surprise that many implementations chose a left-to-right method of recoding the exponent. For example, Libgcrypt implements a left-to-right exponentiation with integrated recoding. Libgcrypt is part of the GnuPG code base [2], and is used in particular by GnuPG 2.x, which is a very popular implementation of the OpenPGP standard [7] for applications such as encrypted email and files. Libgcrypt is also used by various other applications; see [1] for a list of frontends.

It is known that exponentiation using sliding-window methods leaks information, specifically the pattern of squarings and multiplications, through cache-based side-channel attacks. However, it is commonly believed that for window width  $w$  only about a fraction  $2/(w+1)$  bits would leak: each window has 1 bit known to be 1, and each gap has on average 1 bit known to be 0, compared to  $w+1$  bits occupied on average by the window and the gap.

Libgcrypt 1.7.6, the last version at the time of writing this paper, resists the attacks of [10, 16], because the Libgcrypt maintainers accepted patches to protect against chosen-ciphertext attacks and to hide timings obtained from loading precomputed elements. However, the maintainers refused a patch to switch from sliding windows to fixed windows; they said that this was unnecessary to stop the attacks. RSA-1024 in Libgcrypt uses the CRT method and  $w = 4$ , which according to the common belief reveals only 40% of all bits, too few to use the key-recovery attack [12] by Heninger and Shacham. RSA-2048 uses CRT and  $w = 5$ , which according to the common belief reveals only 33% of all bits.

### 1.1 Contributions

In this paper we show that the common belief is incorrect for the left-to-right recoding: this recoding actually leaks many more bits. An attacker learning the location of multiplications in the left-to-right squarings-and-multiplications sequence can recover the key for RSA-1024 with CRT and  $w = 4$  in a search through fewer than 10000 candidates for most keys, and fewer than 1000000 candidates for practically all keys. Note that RSA-1024 and RSA-1280 remain

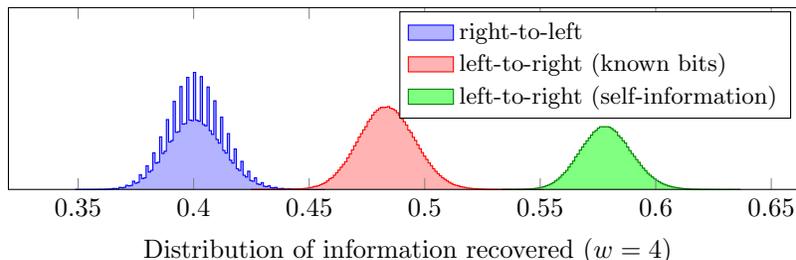


Fig. 1: The sequence of squares and multiplies of left-to-right windowed exponentiation contains much more information about the exponent than from exponentiation in the other direction, both in the form of known bits (red) and information-theoretic bits (green). Recovering close to 50% of the information about the key allows an efficient full key recovery attack.

widely deployed in some applications, such as DNSSEC. Scaling up to RSA-2048 does not stop our attack: we show that 13% of all RSA-2048 keys with CRT and  $w = 5$  are vulnerable to our method after a search through 2000000 candidates.

We analyze the reasons that left-to-right leaks more bits than right-to-left and extensive experiments show the effectiveness of this attack. We further improve the algorithm by Heninger and Shacham to make use of less readily available information to attack RSA-2048, and prove that our extended algorithm efficiently recovers the full key when the side channel leaks data with a *self-information* rate greater than  $1/2$ .

To illustrate the real-world applicability of this attack, we demonstrate how to obtain the required side-channel data (the pattern of squarings and multiplications) from the modular-exponentiation routine in Libcrypt version 1.7.6 using a Flush+Reload [27, 28] cache-timing attack that monitors the target’s cache-access patterns. The attack combines a small number of traces (at most 20) using the same secret RSA key, and does not depend on further front end details.

## 1.2 Targeted Software and Current Status

**Software and Hardware.** We target Libcrypt version 1.7.6, which is the latest version at the time of writing this paper. We compiled Libcrypt using

target	strategy
right-to-left	Rule 0 + Rule 3 (obvious effectiveness)
left-to-right	Rule 0 + Rule 3 (surprise: more effective)
left-to-right	Rules 0, 1, 2, 3 (fast attack against RSA-1024 $w = 4$ )
left-to-right	Rules 0, 1, 2, 3, X (many successes observed against RSA-2048 $w = 5$ )

Table 1: Attack strategies in strictly increasing order of effectiveness. See Section 3.1 for rules; Section 3.2 for analysis of Rules 0, 1, 2, 3; and Section 4 for computations.

GCC version 4.4.7 and the -O2 optimization level. We performed the attack on an HP-Elite 8300 desktop machine, running Centos 6.8 with kernel version 3.18.41-20. The machine has a 4-core Intel i5-3470 processor, running at 3.2 GHz, with 8 GiB of DDR3-1600 CL-11 memory.

**Current Status.** We have disclosed this issue to the Libgcrypt maintainers, who fixed the side-channel vulnerability in version 1.7.8, released June 29, 2017. The vulnerability has been assigned CVE-2017-7526.

## 2 Preliminaries

### 2.1 RSA-CRT

RSA signature key generation is done by generating two random primes  $p, q$ . The public key is then set to be  $(e, N)$  where  $e$  is a (fixed) public exponent and  $N = pq$ . The private key is set to be  $(d, p, q)$  where  $ed \equiv 1 \pmod{\phi(n)}$  and  $\phi(n) = (p-1)(q-1)$ . RSA signature of a message  $m$  is done by computing  $s = h(m)^d \pmod{N}$  where  $h$  is a padded cryptographically secure hash function. Signature verification is done by computing  $z = s^e \pmod{N}$  and verifying that  $z$  equals  $h(m)$ . A common optimization for RSA signatures is based on the Chinese Remainder Theorem (CRT). Instead of directly computing  $s = h(m)^d \pmod{N}$  directly, the signer computes  $s_p = h(m)^{d_p} \pmod{p}$ ,  $s_q = h(m)^{d_q} \pmod{q}$  (where  $d_p$  and  $d_q$  are derived from the secret key) and then combines  $s_p$  and  $s_q$  into  $s$  using the CRT. The computations of  $s_p$  and  $s_q$  work with half-size operands and have half-length exponents, leading to a speedup of a factor 2–4.

### 2.2 Sliding Window Modular Exponentiation

In order to compute an RSA signature (more specifically the values of  $s_p$  and  $s_q$  defined above), two modular exponentiation operations must be performed. A modular exponentiation operation gets as inputs base  $b$ , exponent  $d$ , and modulus  $p$  and outputs  $b^d \pmod{p}$ . A common method used by cryptographic implementations is the sliding window method, which assumes that the exponent  $d$  is given in a special representation, the windowed form. For a window size parameter  $w$ , the windowed form of  $d$  is a sequence of digits  $d_{n-1} \cdots d_0$  such that  $d = \sum_{i=0}^{n-1} d_i 2^i$  and  $d_i$  is either 0 or an odd number between 1 and  $2^w - 1$ .

**Algorithm 1** performs the sliding window exponentiation method, assuming that the exponent is given in a windowed form, in two steps: It first precomputes the values of  $b^1 \pmod{p}, b^3 \pmod{p}, \dots, b^{2^w-1} \pmod{p}$  for odd powers of  $b$ . Then, the algorithm scans the digits of  $d$  from the most significant bit (MSB) to the least significant bit (LSB). For every digit, the algorithm performs a squaring operation (**Line 6**) on the accumulator variable  $a$ . Finally, for every non-zero digit of  $d$ , the algorithm performs a multiplication (**Line 8**).

### 2.3 Sliding Window Conversion

The representation of a number  $d$  in (sliding) windows is not unique, even for a fixed value of  $w$ . In particular, the binary representation of  $d$  is a valid window form. However, since each non-zero digit requires a costly multiplication operation, it is desirable to reduce the number of non-zero digits in  $d$ 's sliding windows.

---

**Algorithm 1** Sliding window modular exponentiation.

---

**Input:** Three integers  $b$ ,  $d$  and  $p$  where  $d_n \cdots d_1$  is a windowed form of  $d$ .

**Output:**  $a \equiv b^d \pmod{p}$ .

```

1: procedure MOD_EXP( $b, d, p$ )
2:    $b_1 \leftarrow b, b_2 \leftarrow b^2 \pmod{p}, a \leftarrow 1$ 
3:   for  $i \leftarrow 1$  to  $2^{w-1} - 1$  do            $\triangleright$  precompute table of small powers of  $b$ 
4:      $b_{2i+1} \leftarrow b_{2i-1} \cdot b_2 \pmod{p}$ 
5:   for  $i \leftarrow n$  to 1 do
6:      $a \leftarrow a \cdot a \pmod{p}$ 
7:     if  $d_i \neq 0$  then
8:        $a \leftarrow a \cdot b_{d_i} \pmod{p}$ 
9:   return  $a$ 
10: end procedure

```

---

**Right-to-Left Sliding Windows.** One approach to computing  $d$ 's sliding windows (with fewer of non-zero digits) scans  $d$ 's binary representation from the least significant bit (LSB) to the most significant bit (MSB) and generates  $d$ 's sliding windows from the least significant digit (right) to the most significant digit (left). For every clear bit, a zero digit is appended to the left of the windowed form. For each set bit, a non-zero digit is appended whose value is the  $w$ -bit integer ending at the current bit. The next  $w - 1$  digits in the windowed form are set to be zero digits. The scan resumes from the leftmost bit unused so far. Finally, any leading zeroes in the window form are truncated.

For example, let  $w = 3$ , and  $d = 181$ , which is 1 0 1 1 0 1 0 1 in binary. The windows are underlined. This yields the sliding window form 10030005.

**Left-to-Right Windowed Form.** An alternative approach is the left-to-right windowed form, which scans the bits of  $d$  the most to least significant bit and the windowed form is generated from the most significant digit to the least significant one. Similar to the right-to-left form, for every scanned clear bit a zero digit is appended to the right of the windowed form. When a set bit is encountered, since we require from digits to be odd, the algorithm cannot simply set the digit to be the  $w$ -bit integer starting at the current bit. Instead, it looks for the longest integer  $u$  that has its most significant bit at the current bit, terminates in a set bit, and its number of bits  $k$  is at most  $w$  bits long. The algorithm sets the next  $k - 1$  digits in the windowed form to be zero, sets the subsequent digit to be  $u$  and resumes the scan from the next bit unused so far. As before, leading zeroes in the sliding window form are truncated.

Using the  $d = 181$  and  $w = 3$  example, the left-to-right sliding windows are 1 0 1 1 0 1 0 1 and the corresponding windowed form is 500501

**Left-to-Right vs. Right-to-Left.** While both the methods produce a windowed form whose average density (the ratio between the non-zero digits and the total form length) is about  $1/(w + 1)$ , generating the windowed form using the right-to-left method guarantees that every non-zero digit is followed by at least  $w - 1$  zero digits. This is contrast to the left-to-right method, where two non-zero

---

**Algorithm 2** Left-to-right sliding window modular exponentiation.

---

**Input:** Three integers  $b$ ,  $d$  and  $p$  where  $d_n \cdots d_1$  is the binary representation of  $d$ .**Output:**  $a \equiv b^d \pmod{p}$ .

```

1: procedure MOD_EXP( $b, d, p$ )
2:    $b_1 \leftarrow b, b_2 \leftarrow b^2, a \leftarrow 1, z \leftarrow 0$ 
3:   for  $i \leftarrow 1$  to  $2^{w-1} - 1$  do           ▷ precompute table of small odd powers of  $b$ 
4:      $b_{2i+1} \leftarrow b_{2i-1} \cdot b_2 \pmod{p}$ 
5:    $i \leftarrow n$ 
6:   while  $i \neq 1$  do                               ▷ main loop for computing  $b^d \pmod{p}$ 
7:      $z \leftarrow z + \text{COUNT\_LEADING\_ZEROS}(d_i \cdots d_1)$ 
8:      $i \leftarrow i - \text{COUNT\_LEADING\_ZEROS}(d_i \cdots d_1)$ 
9:      $l \leftarrow \min(i, w)$ 
10:     $u \leftarrow d_i \cdots d_{i-l+1}$ 
11:     $t \leftarrow \text{COUNT\_TRAILING\_ZEROS}(u)$ 
12:     $u \leftarrow \text{SHIFT\_RIGHT}(u, t)$            ▷ remove trailing zeroes by shifting  $u$  to the right
13:    for  $j \leftarrow 1$  to  $z + l - t$  do
14:       $a \leftarrow a \cdot a \pmod{p}$ 
15:       $a \leftarrow a \cdot b_u \pmod{p}$            ▷ notice that  $u$  is always odd
16:       $i \leftarrow i - l$ 
17:       $z \leftarrow z - t$ 
18:    return  $a$ 
19: end procedure

```

---

digits can be as close as adjacent. As explained in [Section 3](#), such consecutive non-zero digits can be observed by the attacker, aiding key recovery for sliding window exponentiations using the left-to-right windowed form.

## 2.4 GnuPG’s Sliding Window Exponentiation

While producing the right-to-left sliding window form requires a dedicated procedure, the left-to-right form can be generated “on-the-fly” during the exponentiation algorithm, combining the generation of the expansion and the exponentiation itself in one go. Consequently, the left-to-right sliding window form [17, Algorithm 14.85], shown in [Algorithm 2](#), is the prevalent method used by many implementations, including GnuPG.

Every iteration of the main loop ([Line 6](#)) constructs the next non-zero digit  $u$  of the windowed form by locating the location  $i$  of leftmost set bit of  $d$  which was not previously handled ([Line 8](#)) and then removing the trailing zeroes from  $d_i \cdots d_{i-w+1}$ . It appends the squaring operations needed in order to handle the zero windowed form digits preceding  $u$  ([Line 13](#)) before performing the multiplication operation using  $u$  as the index to the precomputation table (thus handling  $u$ ), and keeping track of trailing zeroes in  $z$ .

- Rule 0:**  $\underline{x} \rightarrow \underline{1}$
- Rule 1:**  $\underline{1x^i \underline{1x}^{w-i-1}} \rightarrow \underline{1x^i \underline{1}0^{w-i-1}}$  for  $i = 0, \dots, w - 2$
- Rule 2:**  $\underline{\text{xxx11}} \rightarrow \underline{1\text{xx11}}$
- Rule 3:**  $\underline{1x^i \text{x}^{w-1} \underline{1}} \rightarrow \underline{10^i \text{x}^{w-1} \underline{1}}$  for  $i > 0$

Fig. 2: Rules to deduce known bits from a square-and-multiply sequence

### 3 Sliding Right versus Sliding Left Analysis

In this section, we show how to recover some bits of the secret exponents, assuming that the attacker has access to the square-and-multiply sequence performed by Algorithm 2. We show that more bits can be found by applying this approach to the square-and-multiply sequence of the left-to-right method compared to that of the right-to-left method. At high level, our approach consists of two main steps. In the first step, we show how to directly recover some of the bits of the exponent by analyzing the sequence of squaring and multiplication operations performed by Algorithm 2. This step shows that we are capable of directly recovering an average of 48% of the bits of  $d_p$  and  $d_q$  for 1024-bit RSA with  $w = 4$ , the window size used by Libcrypt for 1024-bit RSA. However, the number of remaining unknown bits required for a full key recovery attack is still too large to brute force. In Section 3.4 we show that applying a modified version of the techniques of [12] allows us to recover the remaining exponent bits and obtain the full private key, if at least 50% of the bits are recovered.

#### 3.1 Analyzing the Square and Multiply Sequence

Assume the attacker has access to the sequence  $S \in \{s, m\}^*$  corresponding to the sequence of square and multiply operations performed by Algorithm 2 executed on some exponent  $d$ . Notice that the squaring operation (Line 13) is performed once per bit of  $d$ , while the multiplication operation is performed only for some exponent bits. Thus, we can represent the attacker’s knowledge about  $S$  as a sequence  $s \in \{0, 1, \underline{1}, \text{x}, \underline{\text{x}}\}^*$  where 0, 1 indicate known bits of  $d$ ,  $\text{x}$  denotes an unknown bit and the positions of multiplications are underlined. For  $y \in \{0, 1, \underline{1}, \text{x}, \underline{\text{x}}\}$  we denote by  $y^i$  the  $i$ -times repetition of  $y$  times.

Since at the start of the analysis all the bits are unknown, we convert  $S$  to the sequence  $s$  as follows: every  $sm$  turns into a  $\underline{\text{x}}$ , all remaining  $s$  into  $\text{x}$ . As a running example, the sequence of squares and multiplies  $S = \text{smssssssmsmsssssm}$  is converted into  $D_1 = \underline{\text{xxxxxxxxxxxxxx}}$ .

To obtain bits of  $d$  from  $S_1$ , the attacker applies the rewrite rules in Figure 2.

**Rule 0: Multiplication bits.** Because every digit in the windowed form is odd, a multiplication always happens at bits that are set.

Applied to  $D_1$  we obtain  $D_2 = \underline{1xxxxxx11xxxx1}$ .

**Rule 1: Trailing zeros.** The algorithm tries to include as many set bits as possible in one digit of the windowed form. So when two multiplications are

fewer than  $w$  bits apart, we learn that there were no further set bits available to include in the digit corresponding to the second multiplication. Rule 1 sets the following bits to zero accordingly.

Applied to  $D_2$  we obtain  $D_3 = \underline{1xxxxx1100x1}$ .

**Rule 2: Leading one.** If we find two immediately consecutive multiplications, it is clear that as the algorithm was building the left digit, there were no trailing zeroes in  $u = d_i \cdots d_{i-l+1}$ , i.e.  $t = 0$  in [Line 11](#). This tells us the precise location of  $d_i$ , which we know is set.

Applied to  $D_3$  we obtain  $D_4 = \underline{1xxx1xx1100x1}$ .

**Rule 3: Leading zeroes.** Every set bit of  $d$  is included in a non-zero digit of the windowed form, so it is at most  $w - 1$  bits to the left of a multiplication. If two consecutive multiplications are more than  $w$  bits apart, we know that there are zeroes in between.

Applied to  $D_4$  we obtain  $D_5 = \underline{10001xx1100x1}$ .

**Larger Example.** Consider the bit string

0100001111100101001100110101001100001100011111100011100100001001.

The corresponding sequence of square and multiply operations (using  $w = 4$ ) evolves as follows as we apply the rules:

xx  
x1xxxxxxxx1xxx1xxxx1xxx1xx1xxxx11xxxx1xxxxxx1x1xxxx1xx1xxxxxxxx1  
x100xxxx1xxx1xxxx1xxx1xx10xxx11000xx10xxxxx1x100xxx1xx10xxxxxx1  
x100xxxx1xxx1xxxx1xxx1xx101xx11000xx10xxxxx1x100xxx1xx10xxxxxx1  
x10000xxx1xxx10xxx1xxx1xx101xx11000xx1000xxx1x100xxx1xx10000xxx1.

Out of the 64 bits, 34 become known through this analysis.

**Iterative Application.** The previous examples shows that by applying rules iteratively, we can discover a few more bits. In particular, for a window where a leading one is recovered (Rule 2), one may learn the leading bit of the preceding window. Iterating Rule 2 in the example above gives 3 more known leading bits:

x10000xxx1xxx101xx11xx11x101xx11000xx1000xxx1x100xxx1xx10000xxx1.

This iterative behavior is hard to analyze and occurs rarely in practice. Therefore the following analysis disregards it. Note that the algorithm of [Section 3.4](#) does use the additional bits.

**Completeness.** The iterative application of these rules recovers almost all knowable bits, which is sufficient for our application. We present an alternative approach that is less intuitive and direct, but is complete in the sense that it recovers all knowable bits, in [Section C](#).

### 3.2 Analyzing Recovery Rules

In this section we analyze the number of bits we are theoretically expected to recover using Rules 0–3 described in the previous section. The analysis applies to general window size  $w$  and the bit string length  $n$ . There are a number of techniques that can be used here. O’Connor [19] modeled recoding rules using regular languages in order to study the resulting weight distribution.

**Renewal processes with rewards.** We model the number of bits recovered as a renewal reward process [23]. A renewal process is associated with interarrival times  $\underline{X} = (X_1, X_2, \dots)$  where the  $X_i$  are independent, identically distributed and non-negative variables with a common distribution function  $F$  and mean  $\mu$ . Let

$$S_n = \sum_{i=1}^n X_i, \quad n \in \mathbb{N},$$

where  $\underline{S} = (0, S_1, S_2, \dots)$  is the sequence of arrival times and

$$N_t = \sum_{n=1}^{\infty} \mathbf{1}(S_n \leq t), \quad t \in \mathbb{R}^+$$

is the associated counting process. Now let  $\underline{Y} = (Y_1, Y_2, \dots)$  be an i.i.d. sequence associated with  $\underline{X}$  in the sense that  $Y_i$  is the reward for the interarrival  $X_i$ . Note that even though both  $\underline{X}$  and  $\underline{Y}$  are i.i.d.,  $X_i$  and  $Y_i$  can be dependent. Then the stochastic process

$$R_t = \sum_{i=1}^{N_t} Y_i, \quad t \in \mathbb{R}^+,$$

is a renewal reward process. The function  $r(t) = \mathbb{E}(R_t)$  is the renewal reward function. We can now state the *renewal reward theorem* [18]. Since  $\mu_X < \infty$  and  $\mu_Y < \infty$  we have for the renewal reward process

$$\begin{aligned} R_t/t &\rightarrow \mu_Y/\mu_X \text{ as } t \rightarrow \infty \text{ with probability 1,} \\ r(t)/t &\rightarrow \mu_Y/\mu_X \text{ as } t \rightarrow \infty. \end{aligned}$$

This is related to our attack in the following way. The  $n$  bit locations of the bit string form an interval of integers  $[1, n]$ , labeling the leftmost bit as 1. We set  $X_1 = b + w - 1$ , where  $b$  is the location of the first bit set to 1, that is, the left boundary of the first window. Then the left boundary of the next window is independent of the first  $b + w - 1$  bits. The renewal process examines each window independently. For each window  $X_i$  we gain information about *at least* the multiplication bit. This is the reward  $Y_i$  associated with  $X_i$ . The renewal reward theorem now implies that for bit strings of length  $n$ , the expected number of recovered bits will converge to  $\frac{n\mu_Y}{\mu_X}$ .

**Recovered bit probabilities.** In the remainder of this section we analyze the expected number of bits that are recovered (the reward) in some number of bits (the renewal length) by the rules of Section 3.1. Then by calculating the probability of each of these rules' occurrence, we can compute the overall number of recovered bits by using the renewal reward theorem. Note that Rule 0 (the bits set to  $\underline{1}$ ) can be incorporated into the other rules by increasing their recovered bits by one.

**Some observations.** We begin with some intuition on bitstring probabilities. The first observation is that in a uniformly random bitstring, each bit follows a Bernoulli distribution and is independent from all other bits. This means also that

at every point the probability that the next  $x$  bits are zeroes (followed by a 1) is  $\mathbb{P}[x] = 1/2^{x+1}$ . The average reward  $R_t$  is the average number of windows  $N_t$  times average reward  $\mu_y$  per window. At the left boundary of a new window, before sliding back, the first bit is a one, while the others are uniformly distributed. The effective window has length  $w$  with probability  $1/2$  (last bit is a 1),  $w - 1$  with probability  $1/2^2$  of length  $w - 1$  (last bits are 10) and so on. For each of the rules in the previous section we will analyze the probability of the rule applying, the average renewal length, and the average reward gained.

**Rule 1: Trailing zeroes.** The first rule applies to short windows. Recall that we call a window a “short window” whenever the length between two multiplications is less than  $w - 1$ .

Let  $0 \leq j \leq w - 2$  denote the length between two multiplications. (A length of  $w - 1$  is a full-size window.) The probability of a short window depends on these  $j$  bits, as well as  $w - 1$  bits after the multiplication: the multiplication bit should be the right-most 1-bit in the window. The following theorem (which we prove in Appendix B) gives the probability of a short window.

**Theorem 1.** *Let  $X$  be an interarrival time. Then the probability that  $X = w$  and we have a short window with reward  $Y = w - j$ ,  $0 \leq j \leq w - 2$  is*

$$p_j = \frac{1 + \sum_{i=1}^j 2^{2i-1}}{2^{j+w}}$$

We see in the proof that the bits  $y_{w-j-1}, \dots, y_{w-2}$  can take any values. Also since bit  $y_{w-j-2} = 0$  is known, we have a renewal at this point where future bits are independent.

**Rule 2: Leading one.** As explained in Section 3.1, this rule means that when after renewal an ultra-short window occurs (a 1 followed by  $w - 1$  zeroes) we get an extra bit of information about the previous window. The exception to this rule is if the previous window was also an ultra-short window. In this case the 1 of the window is at the location of the multiplication bit we would have learned and therefore we do not get extra information. As seen in the previous section, an ultra-short window occurs with probability  $p_0 = 1/2^w$ . If an ultra-short window occurs after the current window with window-size  $1 \leq j \leq w - 1$ , we therefore recover  $(w - j) + 1$  bits (all bits of the current window plus 1 for the leading bit) with probability  $p_j p_0$  and  $(w - j)$  with probability  $p_j(1 - p_0)$ .

**Rule 3: Leading zeroes.** The last way in which extra bits can be recovered is the leading zeroes. If a window of size  $w - d$  is preceded by more than  $d$  zeroes, then we can recover the excess zeroes. Let  $X_0$  be a random variable of the length of a bit string of zeros until the first 1 is encountered. Then  $X_0$  is geometrically distributed with  $p = 1/2$ . So  $\mathbb{P}[X_0 = k] = (1/2)^k \cdot (1/2) = (1/2)^{k+1}$ . This distribution has mean  $\mu_X = 1$ .

Let  $X_w$  be a random variable representing the length of the bit string from the first 1 that was encountered until the multiplication bit. For general window

length of  $w$ , we have

$$\mathbb{P}[X_w = k] = \begin{cases} \frac{1}{2^{w-1}} & k = 1 \\ \frac{1}{2^{w-k+1}} & k > 1 \end{cases}$$

Now the distribution of the full bit string is the sum of the variables  $X_0$  and  $X_w$ . We have that  $\mathbb{P}[X_0 + X_w = k] = \sum_{i=1}^{\min(k,w)} \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i]$ . Notice that this rule only recovers bits if the gap between two multiplications is at least  $w - 1$ . This means that these cases are independent of Rule 1.

There is a small caveat in this analysis: the renewal length is unclear. In the case that we have a sequence of zeroes followed by a short window of size  $j < w$ , we are implicitly conditioning on the  $w - j$  bits that follow. This means we cannot simply renew after the the  $\underline{1}$  and since we also cannot distinguish between a short and regular window size, we also cannot know how much information we have on the bits that follow.

We solve this by introducing an upper and lower bound. For the upper bound the recovered bits remains as above and the renewal at  $X_0 + w$ . This is an obvious upper bound. This means that for a sequence of zeroes followed by a short window of size  $j$ , we assume a probability of 1 of recovering information on the  $w - j$  bits that follow the sequence. We get an average recovered bits of

$$\bar{R} = \sum_{k=w}^{\infty} \sum_{i=1}^{\min(k,w)} (k - i + 1) \cdot \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i],$$

and a renewal length of

$$\bar{N} = \sum_{k=w}^{\infty} \sum_{i=1}^{\min(k,w)} (k + w - i) \cdot \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i].$$

For the lower bound we could instead assume a probability of 0 of recovering information on the  $w - j$  bits. We can however get a tighter bound by observing that the bits that follow this rule are more likely a 0 than a 1 and we are more likely to recover a 0 at the start of a new window than we are a 0. Therefore bound the renewal at  $X_0 + X_w$  and throw away the extra information. For the recovered bits of the lower bound this gives the following expectation

$$\underline{R} = \sum_{k=w}^{\infty} (k - w + 1) \cdot \mathbb{P}[X_0 + X_w = k],$$

and a renewal length of

$$\underline{N} = \sum_{k=w}^{\infty} \sum_{i=1}^{\min(k,w)} k \cdot \mathbb{P}[X_w = i] \cdot \mathbb{P}[X_0 = k - i].$$

We summarize the results in Table 2.

Table 2: Summary of bit information recovered from different patterns

Rule	Pattern	Probability	Renewal length	Recovered bits
1	Window size $1 < j < w$	$p_j(1 - p_0)$	$w$	$(w - j)$
		$p_j p_0$	$w$	$(w - j) + 1$
2	Window size $j = 1$	$p_0$	$w$	$w$
3	ub window size $> w$	$\mathbb{P}[X + Y = k]$	$\overline{N}$	$\overline{R}$
	lb window size $> w$	$\mathbb{P}[X + Y = k]$	$\underline{N}$	$\underline{R}$

From this, we can calculate the expected renewal length for fixed  $w$ , by summing over all possible renewal lengths with corresponding probabilities. We can do the same for the expected number of recovered bits per renewal. Finally, we are interested in the expected total number of recovered bits in a  $n$ -bit string. We calculate this by taking an average number of renewals (by dividing  $n$  by the expected renewal length) and multiply this with the number of recovered bits per window. Since we have upper and lower bounds for both the renewal length and recovered bits for Rule 3, we also get lower and upper bounds for the expected total number of recovered bits.

**Recovered Bits for Right-to-Left.** The analysis of bit recovery for right-to-left exponentiation is simpler. The bit string is an alternation of  $X_0$  and  $X_w$  (see Rule 3), where  $X_w = w$  and  $X_0$  is geometrically distributed with  $p = 1/2$ . Therefore the expected renewal length  $N$  and the expected reward  $R$  are

$$N = \sum_{i=0}^{\infty} (w + i) \cdot \mathbb{P}[X_0 = i] = w + 1 \quad \text{and} \quad R = \sum_{i=0}^{\infty} (1 + i) \cdot \mathbb{P}[X_0 = i] = 2.$$

Then by the renewal reward theorem, we expect to recover  $\frac{2n}{w+1}$  bits.

### 3.3 Experimental Verification

To experimentally validate our analysis, we sampled  $n$ -bit binary strings uniformly at random and used [Algorithm 2](#) to derive the square and multiply sequence. We then applied Rules 0–3 from [Section 3.1](#) to extract known bits.

**Case  $n = 512, w = 4$ .** [Figure 1](#) shows the total fraction of bits learned for right-to-left exponentiation compared to left-to-right exponentiation, for  $w = 4$ , over 1,000,000 experiments with  $w = 4$  and  $n = 512$ , corresponding to the our target Libgcrypt’s implementation for 1024-bit RSA. On average we learned 251 bits, or 49%, for left-to-right exponentiation with 512-bit exponents. This is between our computed lower bound of  $\beta_L = 245$  (from a renewal length of  $\underline{N} = 4.67$  bits and reward of 2.24 bits on average per renewal) and upper bound  $\beta_U = 258$  (from a renewal length of  $\overline{N} = 4.90$  bits and reward of 2.47 bits per renewal). The average number of recovered bits for right-to-left exponentiation is  $204 \approx \frac{2n}{w+1}$  bits, or 40%, as expected.

[Figure 3](#) shows the distribution of the number of new bits learned for each rule with left-to-right exponentiation by successively applying Rules 0–3 for 100,000 exponents. Both Rule 0 and Rule 3 contribute about  $205 \approx \frac{2n}{w+1}$  bits, which is equal both to our theoretical analysis and is also the number of bits learned from

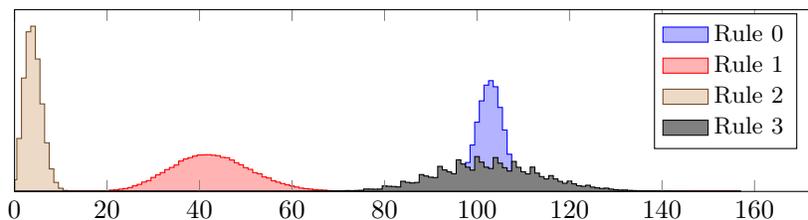


Fig. 3: We generated 100,000 random 512-bit strings and generated the square and multiply sequence with  $w = 4$ . We then applied Rules 0–3 successively to recover bits of the original string. We plot the distribution of the number of recovered bits in our experiments.

the right-to-left exponentiation. The spikes visible in Rule 3 are due to the fact that we know that any least significant bits occurring after the last window must be 0, and we credit these bits learned to Rule 3. The number of bits learned from this final step is equal to  $n \bmod w$ , leading to small spikes at intervals of  $w$  bits.

**Case  $n = 1024, w = 5$ .** For  $n = 1024$  and  $w = 5$ , corresponding to Libgcrypt’s implementation of 2048-bit RSA, we recover 41.5% of bits, or 425, on average using Rules 0–3. This is between our lower bound of  $\beta_L = 412$  (from a lower bound average renewal length of  $\bar{N} = 5.67$  bits, and expected 2.29 bits on average per renewal) and upper bound of  $\beta_U = 436$  (from an average renewal length of  $\bar{N} = 5.89$  bits with an average reward of 2.51 bits per renewal). Note that the reward per renewal is about the same as in the first case ( $n = 512, w = 4$ ), but the average renewal length is higher. This means that we win fewer bits for this case.

In Figure 4 we plot the frequency of the number of recovered bits per rule in 100,000 experiments, applied in order. Note again that both Rule 0 and Rule 3 contribute about  $\frac{2n}{w+1}$  bits, which is what we would expect in theory.

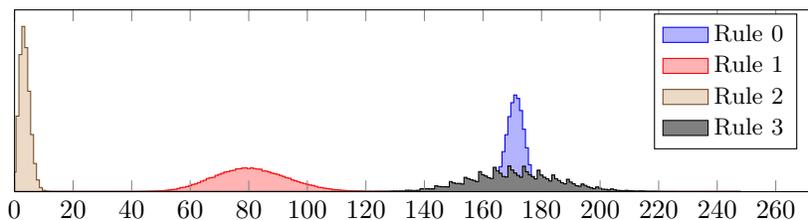


Fig. 4: We generated 100,000 random 1024-bit strings and generated the square and multiply sequence with  $w = 5$ . We then applied Rules 0–3 successively to recover bits of the original string and plot the distribution of the number of new bits recovered for each rule. Compared to the  $w = 4$  case, we learn a lower fraction of bits in this case.

### 3.4 Full RSA Key Recovery from Known Bits

Once we have used the recovered sequence of squares and multiplies to derive some information about the bits of the Chinese remainder theorem coefficients  $d_p = d \bmod (p - 1)$  and  $d_q = d \bmod (q - 1)$ , we can use a modified version of the branch and prune algorithm of Heninger and Shacham [12] to recover the remaining unknown bits of these exponents to recover the full private key.

The algorithm will recover the values  $d_p$  and  $d_q$  from partial information. In order to do so, we use the integer forms of the RSA equations

$$\begin{aligned} ed_p &= 1 + k_p(p - 1) \\ ed_q &= 1 + k_q(q - 1) \end{aligned}$$

which these values satisfy for positive integers  $k_p, k_q < e$ .

**RSA Coefficient Recovery.** As described in [13, 29],  $k_p$  and  $k_q$  are initially unknown, but are related via the equation  $(k_p - 1)(k_q - 1) \equiv k_p k_q N \bmod e$ . Thus we need to try at most  $e$  pairs of  $k_p, k_q$ . In the most common case,  $e = 65537$ . As described in [29], incorrect values of  $k_p, k_q$  quickly result in no solutions.

**LSB-Side Branch and Prune Algorithm.** At the beginning of the algorithm, we have deduced some bits of  $d_p$  and  $d_q$  using Rules 0–3. Given candidate values for  $k_p$  and  $k_q$ , we can then apply the approach of [12] to recover successive bits of the key starting from the least significant bits. Our algorithm does a depth-first search over the unknown bits of  $d_p, d_q, p$ , and  $q$ . At the  $i$ th least significant bit, we have generated a candidate solution for bits  $0 \dots i - 1$  of each of our unknown values. We then verify the equations

$$\begin{aligned} ed_p &= 1 + k_p(p - 1) \bmod 2^i \\ ed_q &= 1 + k_q(q - 1) \bmod 2^i \\ pq &= N \bmod 2^i \end{aligned} \tag{1}$$

and prune a candidate solution if any of these equations is not satisfied.

**Analysis.** Heuristically, we expect this approach to be efficient when we know more than 50% of bits for  $d_p$  and  $d_q$ , distributed uniformly at random. [12, 20] We also expect the running time to grow exponentially in the number of unknown bits when we know many fewer than 50% of the bits. From the analysis of Rules 0–3 above, we expect to recover 48% of the bits. While the sequence of recovered bits is not, strictly speaking, uniformly random since it is derived using deterministic rules, the experimental performance of the algorithm matched that of a random sequence.

**Experimental Evaluation for  $w = 4$ .** We ran 500,000 trial key recovery attempts for randomly generated  $d_p$  and  $d_q$  with 1024-bit RSA and  $w = 4$ . For a given trial, if the branching process passed 1,000,000 candidates examined without finding a solution, we abandoned the attempt. Experimentally, we recover more than 50% of the bits 32% of the time, and successfully recovered the key in 28% of our trials. For 50% or 512 bits known, the median number of examined

candidates was 182,738. We recovered 501 bits on average in our trials using Rules 0–3; at this level the median number of candidates was above 1 million.

**Experimental Evaluation for  $w = 5$ .** We experimented with this algorithm for 2048-bit RSA, with  $w = 5$ . The number of bits that can be derived unconditionally using Rules 0–3 is around 41% on average, below the threshold where we expect the Heninger-Shacham algorithm to terminate for 1024-bit exponents. The algorithm did not yield any successful key recovery trials at this size.

## 4 RSA Key Recovery from Squares and Multiplies

The sequence of squares and multiplies encodes additional information about the secret exponent that does not translate directly into knowledge of particular bits. In this section, we give a new algorithm that exploits this additional information by recovering RSA keys directly from the square-and-multiply sequence. This gives a significant speed improvement over the key recovery algorithm described in Section 3.4, and brings an attack against  $w = 5$  within feasible range.

### 4.1 Pruning from Squares and Multiplies

Our new algorithm generates a depth-first tree of candidate solutions for the secret exponents, and prunes a candidate solution if it could not have produced the ground-truth square-and-multiply sequence obtained by the side-channel attack. Let  $\text{SM}(d) = s$  be the deterministic function that maps a bit string  $d$  to a sequence of squares and multiplies  $s \in \{\text{s}, \text{m}\}^*$ .

In the beginning of the algorithm, we assume we have ground truth square-and-multiply sequences  $s_p$  and  $s_q$  corresponding to the unknown CRT coefficients  $d_p$  and  $d_q$ . We begin by recovering the coefficients  $k_p$  and  $k_q$  using brute force as described in Section 3.4. We will then iteratively produce candidate solutions for the bits of  $d_p$  and  $d_q$  by generating a depth-first search tree of candidates satisfying Equations 1 starting at the least significant bits. We will attempt to prune candidate solutions for  $d_p$  or  $d_q$  at bit locations  $i$  for which we know the precise state of Algorithm 2 from the corresponding square and multiply sequence  $s$ , namely when element  $i$  of  $s$  is a multiply or begins a sequence of  $w$  squares. To test an  $i$ -bit candidate exponent  $d_i$ , we compare  $s' = \text{SM}(d_i)$  to positions 0 through  $i - 1$  of  $s$ , and prune  $d_i$  if the sequences do not match exactly.

### 4.2 Algorithm Analysis

We analyze the performance of this algorithm by computing the expected number of candidate solutions examined by the algorithm before it recovers a full key. Our analysis was inspired by the information-theoretic analysis of [20], but we had to develop a new approach to capture the present scenario.

Let  $p_s = \Pr[\text{SM}(d_i) = s]$  be the probability that a fixed square-and-multiply sequence  $s$  is observed for a uniformly random  $i$ -bit sequence  $d_i$ . This defines the probability distribution  $D_i$  of square-and-multiply sequences for  $i$ -bit inputs.

In order to understand how much information a sequence  $s$  leaks about an exponent, we will use the *self-information*, defined as  $I_s = -\log p_s$ . This is the analogue of the number of bits known for the algorithm given in Section 3.4. As

with the bit count, we can express the number of candidate solutions that generate  $s$  in terms of  $I_s$ :  $\#\{d \mid \text{SM}(d) = s\} = 2^i p_s = 2^i 2^{-I_s}$ . For a given sequence  $s$ , let  $I_i$  denote the self-information of the least significant  $i$  bits.

**Theorem 2 (Heuristic).** *If for the square-and-multiply sequences  $s_{p_i}$  and  $s_{q_i}$ , we have  $I_i > i/2$  for almost all  $i$ , then the algorithm described in Section 4.1 runs in expected linear time in the number of bits of the exponent.*

*Proof (Sketch).* In addition pruning based on  $s$ , the algorithm also prunes by verifying the RSA equations up to bit position  $i$ . Let  $\text{RSA}_i(d_p, d_q) = 1$  if  $(ed_p - 1 + k_p)(ed_q - 1 + k_q) = k_p k_q N \pmod{2^i}$  and 0 otherwise. For random (incorrect) candidates  $d_{p_i}$  and  $d_{q_i}$ ,  $\Pr[\text{RSA}_i(d_{p_i}, d_{q_i})] = 1/2^i$ .

As in [12], we heuristically assume that, once a bit has been guessed wrong, the set of satisfying candidates for  $d_{p_i}$  and  $d_{q_i}$  behave randomly and independently with respect to the RSA equation at bit position  $i$ .

Consider an incorrect guess at the first bit. We wish to bound the candidates examined before the decision is pruned. The number of incorrect candidates satisfying the square-and-multiply constraints and the RSA equation at bit  $i$  is

$$\begin{aligned} \#\{d_{p_i}, d_{q_i}\} &\leq \#\{d_{p_i} \mid \text{SM}(d_{p_i}) = s_{p_i}\} \cdot \#\{d_{q_i} \mid \text{SM}(d_{q_i}) = s_{q_i}\} \cdot \Pr[\text{RSA}_i(d_{p_i}, d_{q_i})] \\ &= 2^i 2^{-I_i} \cdot 2^i 2^{-I_i} \cdot 2^{-i} = 2^{i-2I_i} \leq 2^{i \cdot (1-2c)} \end{aligned}$$

with  $I_i/i \geq c$  for some  $c > 1/2$ .

In total, there are  $\sum_i \#\{d_{p_i}, d_{q_i}\} \leq \sum_i 2^{i \cdot (1-2c)} \leq 1/(1 - 2^{1-2c})$  candidates.

But *any* of the  $n$  bits can be guessed wrongly, each producing a branch of that size. Therefore, the total search tree has at most  $n \cdot (1 + \frac{1}{1-2^{1-2c}})$  nodes.

A similar argument also tells us about the expected size of the search tree, which depends on the *collision entropy* [22]

$$H_i = -\log \sum_{s \in \{s, m\}^i} p_s^2$$

of the distribution  $D_i$  of distinct square-and-multiply sequences. This is the log of the probability that two  $i$ -bit sequences chosen according to  $D_i$  are identical.

For our distribution  $D_i$ , the  $H_i$  are approximately linear in  $i$ . We can define the *collision entropy rate*  $H = H_i/i$  and obtain an upper bound for the expected number of examined solutions, which we prove in the full version of the paper:

**Theorem 3.** *The expected total number of candidate solutions examined by Algorithm 2 for  $n$ -bit  $d_p$  and  $d_q$  is*

$$E \left[ \sum_i \#\{d_{p_i}, d_{q_i}\} \right] \leq n \left( 1 + \frac{1 - 2^{n \cdot (1-2H)}}{1 - 2^{1-2H}} \right).$$

See Section E for the proof.

**Entropy calculations.** We calculated the collision entropy rate by modeling the leak as a Markov chain. For  $w = 4$ ,  $H = 0.545$ , and thus we expect Algorithm

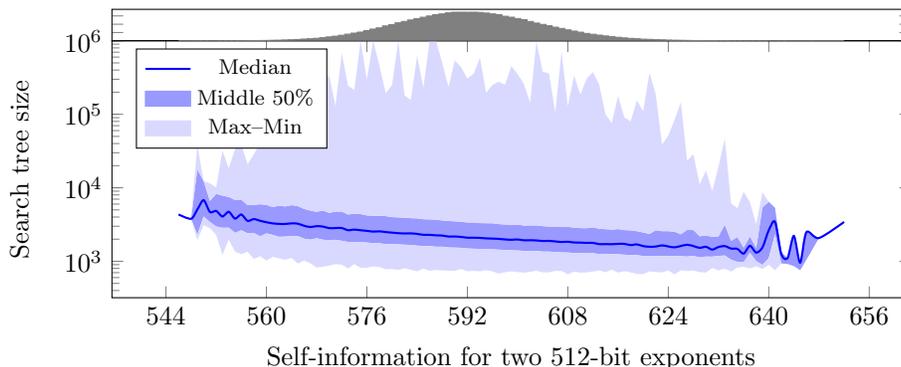


Fig. 5: We attempted 500,000 key recovery trials for randomly generated 1024-bit RSA keys with  $w = 4$ . We plot the distribution of the number of candidates tested by the algorithm against the self-information of the observed square-and-multiply sequences, calculated using the algorithm in Section C.2 and measured in bits. The histogram above the plot shows the distribution of self-information across all the trials.

2 to comfortably run in expected linear time. For  $w = 5$ ,  $H = 0.461$ , and thus we expect the algorithm to successfully terminate on some fraction of inputs. We give more details on this computation in Section D.

#### 4.3 Experimental Evaluation for $w = 4$

We ran 500,000 trials of our sequence-pruning algorithm for randomly generated  $d_p$  and  $d_q$  with 1024-bit RSA and plot the distribution of running times in Figure 5. For a given trial, if the branching process passed 1,000,000 candidates examined without finding a solution, we abandoned the attempt. For each trial square-and-multiply sequence  $s$ , we computed the number of bit sequences that could have generated it. From the average of this quantity over the 1 million exponents generated in our trial, the collision entropy rate in our experiments is  $H = 0.547$ , in line with our analytic computation above. The median self-information of the exponents generated in our trials was 295 bits; at this level the median number of candidates examined by the algorithm was 2,174. This can be directly compared to the 251 bits recovered in Section 3, since the self-information in that case is exactly the number of known bits in the exponent.

#### 4.4 Experimental Evaluation for $w = 5$

We ran 500,000 trials of our sequence-pruning algorithm for 2048-bit RSA and  $w = 5$  with randomly generated  $d_p$  and  $d_q$  and plot the distribution of running times in Figure 6. 8.6% of our experimental trials successfully recovered the key before hitting the panic threshold of 1 million tries. Increasing the allowed tree size to 2 million tries allowed us to recover the key in 13% of trials. We experimentally estimate a collision entropy rate  $H = 0.464$ , in line with our analytic computation. The median self-information for the exponents generated

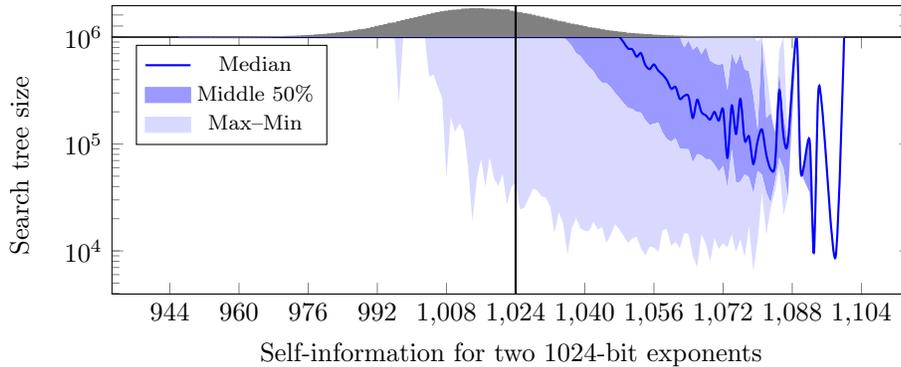


Fig. 6: We attempted 500,000 key recovery trials for randomly generated 2048-bit RSA keys with  $w = 5$ , and plot the distribution of search tree size by the self-information. The vertical line marks the 50% rate at which we expect the algorithm to be efficient.

in our trials is 507 bits, significantly higher than the 425 bits that can be directly recovered using the analysis in Section 3.

## 5 Attacking Libcrypt

In the previous section we showed how an attacker with access to the square-and-multiply sequence can recover the private RSA key. To complete the discussion we show how the attacker can obtain the square-and-multiply sequence.

### 5.1 The Side-Channel Attack

To demonstrate the vulnerability in Libcrypt, we use the Flush+Reload attack [28]. The attack, which monitors shared memory locations for access by a victim, consists of two phases. The attacker first evicts a monitored location from the cache, typically using the x86 `cflush` instruction. He then waits for a short while, before measuring the time to access the monitored location. If the victim has accessed the memory location during the wait, the contents of the memory location will be cached, and the attacker's access will be fast. Otherwise, the attacker causes the contents to be retrieved from the main memory and his access takes longer. By repeating the attack, the attacker creates a trace of the victim accesses to the monitored location over time. Flush+Reload has been used in the past to attack modular exponentiation [3, 28], as well as other cryptographic primitives [4, 6, 14, 21, 27] and non-cryptographic software [11, 31].

Mounting the Flush+Reload attack on Libcrypt presents several challenges. First, as part of the defense against the attack of Yarom and Falkner [28], Libcrypt uses the multiplication code to perform the squaring operation. While this is less efficient than using a dedicated code for squaring, the use of the same code means that we cannot identify the multiply operations by probing a

separate multiply code. Instead we probe code locations that are used between the operations to identify the call site to the modular reduction.

The second challenge is achieving a sufficiently high temporal resolution. Prior side-channel attacks on implementations of modular exponentiation use large (1024–4096 bits) moduli [9, 10, 16, 28, 30], which facilitate side-channel attacks [25]. In this attack we target RSA-1024, which uses 512-bit moduli. The operations on these moduli are relatively fast, taking a total of less than 2500 cycles on average to compute a modular multiplication. To be able to distinguish events of this length, we must probe at least twice as fast, which is close to the limit of the Flush+Reload attack and would result in a high error rate [3]. We use the amplification attack of Allan et al. [3] to slow down the modular reduction. We target the code of the subtraction function used as part of the modular reduction. The attack increases the execution time of the modular reduction to over 30000 cycles.

Our third challenge is that even with amplification, there is a chance of missing a probe [3]. To reduce the probability of this happening, we probe two memory locations within the execution path of short code segments. The likelihood of missing both probes is small enough to allow high-quality traces.

Overall, we use the Flush+Reload attack to monitor seven victim code locations. (See Appendix A for details.) The monitored locations can be divided into three groups. To distinguish between the exponentiations Libgcrypt performs while signing, we monitor locations in the entry and exit of the exponentiation function. We also monitor a location in the loop that precomputes the multipliers to help identifying these multiplications. To trace individual modular multiplications, we monitor locations within the multiply and the modular reduction functions. Finally, to identify the multiplication by non-zero multipliers, we monitor locations in the code that conditionally copies the multiplier and in the entry to the main loop of the exponentiation. The former is accessed when Libgcrypt selects the multiplier before it performs the multiplication. The latter is accessed after the multiplication when the next iteration of the main loop starts. We repeatedly probe these locations once every 10000 cycles, allowing for 3–4 probes in each modular multiply or square operation.

## 5.2 Results

To mount the attack, we use the `FR-trace` software, included in the Mastik toolkit [26]. `FR-trace` provides a command-line interface for performing the Flush+Reload and the amplification attacks we require for recovering the square-and-multiply sequences of the Libgcrypt exponentiation. `FR-trace` waits until there is activity in any of the monitored locations and collects a trace of the activity. Figure 7 shows a part of a collected activity trace.

Recall that the Flush+Reload attack identifies activity in a location by measuring the time it takes to read the contents of the location. Fast reads indicate activity. In the figure, monitored locations with read time below 100 cycles indicate that the location was active during the sample.

Because multiplications take an average 800 cycles, whereas our sample rate is once in 10000 cycles, most of the time activity in the multiplication code is

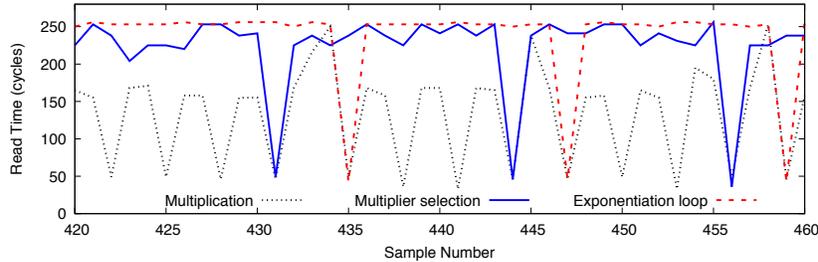


Fig. 7: Libgrypt Activity Trace.

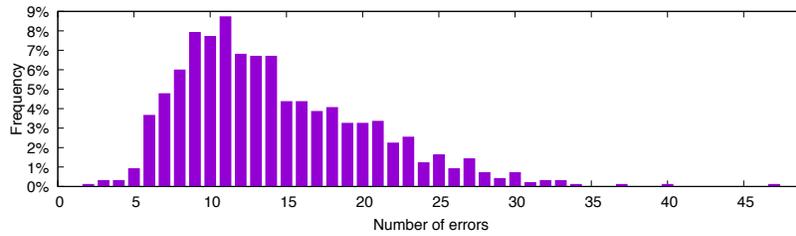


Fig. 8: Distribution of the number of errors in captured traces.

contained within a single sample. In [Figure 7](#) we see the multiplication operations as “dips” in the multiplication trace (dotted black).

Each multiplication operation is followed by a modular reduction. Our side-channel amplification attack “stretches” the execution of the modular reduction and it spans over more than 30000 cycles. Because none of the memory addresses traced in the figure is active during the modular reduction, we see gaps of 3–4 samples between periods of activity in any of the other monitored locations.

To distinguish between multiplications that use one of the precomputed multipliers and multiplications that square the accumulator by multiplying it with itself, we rely on activity in the multiplier selection and in the exponentiation loop locations. Before multiplying with a precomputed multiplier, the multiplier needs to be selected. Hence we would expect to see activity in the multiplier selection location just before starting the multiply, and due to the temporal granularity of the attack we are likely to see both events in the same sample. Similarly, after performing the multiplication and the modular reduction, we expect to see activity in the beginning of the main exponentiation loop. Again, due to attack granularity, this activity is likely to occur within the same sample as the following multiplication. Thus, because we see activity in the multiplier selection location during sample 431 and activity in the beginning of the exponentiation loop in the following multiplication (sample 435), we can conclude that the former multiplication is using one of the precomputed multipliers.

In the absence of errors, this allows us to completely recover the sequence of square and multiplies performed and with it, the positions of the non-zero digits in the windowed representation of the exponent.

However, capture errors do occur, as shown in [Figure 8](#). To correct these, we capture multiple traces of signatures using the same private key. On average there are 14 errors in a captured trace. We find that in most cases, manually aligning traces and using a simple majority rule is sufficient to recover the complete square-and-multiply sequence. In all of the cases we have tried, combining twenty sequences yielded the complete sequence.

## Acknowledgments

Author list is in alphabetical order; see <https://www.ams.org/profession/leaders/culture/CultureStatement04.pdf>. We thank Guenaëlle de Julis for pointing out an error in Algorithm 2 and Luke O'Connor for pointing us to his related work.

Yuval Yarom performed part of this work as a visiting scholar at the University of Pennsylvania. This work was supported by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005; by the Commission of the European Communities through the Horizon 2020 program under project number 645622 (PQCRYPTO) and project number 645421 (ECRYPT-CSA); by the National Science Foundation under grants 1314919, 1408734, 1505799, 1513671, 1319880 and 14-519; by a gift from Cisco; by an Endeavour Research Fellowship from the Australian Department of Education and Training; by the 2017-2018 Rothschild Postdoctoral Fellowship; by the Blavatnik Interdisciplinary Cyber Research Center (ICRC); by the Check Point Institute for Information Security; by the Israeli Centers of Research Excellence I-CORE program (center 4/11); by the Leona M. & Harry B. Helmsley Charitable Trust; by the Warren Center for Network and Data Sciences; by the financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology; and by the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

Permanent ID of this document: 8016c16382e6f3876aa03bef6e4db5ff. Date: 2017.06.28.

## Bibliography

- [1] GnuPG Frontends, . URL [https://www.gnupg.org/related\\_software/frontends.html](https://www.gnupg.org/related_software/frontends.html).
- [2] GNU Privacy Guard, . URL <https://www.gnupg.org>.
- [3] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *32nd Annual Computer Security Applications onference (ACSAC)*, Los Angeles, CA, US, December 2016.
- [4] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. “Ooh aah. . . , just a little bit”: A small amount of side channel can go a long way. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 75–92, Busan, KR, September 2014.
- [5] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, 2017. (accepted).
- [6] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload – A cache attack on the BLISS lattice-based signature scheme. In *CHES*, volume 9813 of *LNCS*, pages 323–345. Springer, 2016.
- [7] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. OpenPGP message format. RFC 4880, November 2007.
- [8] Christophe Doche. Exponentiation. In *Handbook of Elliptic and Hyperelliptic Curve Cryptography.*, pages 144–168. Chapman and Hall/CRC, 2005. doi: 10.1201/9781420034981.pt2.
- [9] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *CRYPTO 2014*, pages 444–461 (vol. 1). Springer, 2014. Extended version: Cryptology ePrint Archive, Report 2013/857.
- [10] Daniel Genkin, Lev Pachmanov, Itamar Pipman, and Eran Tromer. Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation. In *Workshop on Cryptographic Hardware and Embedded Systems (CHES) 2015*, pages 207–228, 2015. Extended version: Cryptology ePrint Archive, Report 2015/170.
- [11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*, pages 897–912, Washington, DC, US, August 2015.
- [12] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In *CRYPTO*, volume 5677 of *LNCS*, pages 1–17. Springer, 2009.
- [13] Mehmet Sinan İnci, Gorka Irazoqui, Berk Gülmezoğlu, Thomas Eisenbarth, and Berk Sunar. Cache attacks enable bulk key recovery on the cloud. In

- Cryptographic Hardware and Embedded Systems (CHES)*, Santa Barbara, CA, US, August 2016.
- [14] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 299–319, Gothenburg, Sweden, September 2014.
  - [15] Marc Joye and Sung-Ming Yen. Optimal left-to-right binary signed-digit recoding. *IEEE Trans. Computers*, 49(7):740–748, 2000. doi: 10.1109/12.863044.
  - [16] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P) 2015*. IEEE, 2015.
  - [17] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, October 1996.
  - [18] J.J. Moder and S.E. Elmaghraby. *Handbook of operations research: models and applications*. Number v. 1. Van Nostrand Reinhold Co., 1978.
  - [19] Luke O’Connor. *An Analysis of Exponentiation Based on Formal Languages*, pages 375–388. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-48910-8.
  - [20] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. *A Coding-Theoretic Approach to Recovering Noisy RSA Keys*, pages 386–403. Springer, 2012.
  - [21] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In *Cryptographers’ track at the RSA Conference on Topics in Cryptology (CT-RSA)*, pages 3–21, San Francisco, CA, USA, April 2015.
  - [22] Alfréd Rényi. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 547–561, Berkeley, Calif., 1961.
  - [23] Sheldon Mark Ross. *Stochastic processes*. Probability and mathematical statistics. Wiley, 1983. ISBN 0-471-09942-2.
  - [24] Maciej Skórski. Shannon entropy versus Rényi entropy from a cryptographic viewpoint. IMACC 2015, pages 257–274. Springer, 2015. doi: 10.1007/978-3-319-27239-9\_16.
  - [25] Colin D. Walter. Longer keys may facilitate side channel attacks. In *Selected Areas in Cryptography (SAC)*, pages 42–57, Waterloo, ON, CA, August 2004.
  - [26] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit. <http://cs.adelaide.edu.au/~yval/Mastik/Mastik.pdf>, September 2016.
  - [27] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, February 2014.
  - [28] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *25th USENIX Security Symposium*, pages 719–732, San Diego, CA, US, 2014.
  - [29] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In *Cryptographic Hardware and Embedded Systems (CHES)*, pages 346–367. Springer, 2016.

- [30] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM Conference on Computer and Communications Security (CCS)*, pages 305–316, Raleigh, NC, US, October 2012.
- [31] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-Tenant side-channel attacks in PaaS clouds. In *Computer and Communications Security (CCS)*, Scottsdale, AZ, US, 2014.

## A Monitored Libcrypt Code Locations

Table 3: Side-Channel attack on Libcrypt — Monitored locations.

Location	Comment
mpi-pow.c:403	Entry to the exponentiation function
mpi-pow.c:564	Precomputation of the multipliers
mpi-pow.c:780	Exit from the exponentiation function
_gcry_mpih_mul_1+72	Part of the multiplication code
mpih-div.c:354	Part of the modular reduction code
_gcry_mpi_set_cond+64	A location in the conditional copy of multiplier (accessed before multiplications)
mpi-pow.c:602	Entry to the exponentiation main loop (accessed after multiplications)

As discussed in [Section 5](#), we use the Flush+Reload attack to monitor seven victim code location. These are summarized in [Table 3](#). To invoke the attack we use the following command line:

```
FR-trace -H -s 10000 -c 100000 -i 500 -p 2 -f libcrypt.so.20
-m mpi-pow.c:602 -m _gcry_mpih_mul_1+0x48 -m mpih-div.c:354
-m mpi-pow.c:403 -m mpi-pow.c:564 -m _gcry_mpi_set_cond+0x40
-m mpi-pow.c:780 -t _gcry_mpih_submul_1 -t _gcry_mpih_submul_1+64
```

The `-m` arguments specify locations to monitor. `-t` specify memory locations targeted by the amplification attack. Both targeted locations are within the code of the function `_gcry_mpih_submul_1`, which subtracts large integers. It is used at the heart of the calculation of the modular reduction.

## B Proof of Theorem 1

**Theorem 3.** *Let  $X$  be an interarrival time. Then the probability that  $X = w$  and we have a short window with reward  $Y = w - j$ ,  $0 \leq j \leq w - 2$  is*

$$p_j = \frac{1 + \sum_{i=1}^j 2^{2i-1}}{2^{j+w}}$$

*Proof.* To prove this theorem, we first make two observations. The first observation is that given a substring of the multiplication chain, the attack recovers a

bit independently from bits that are more than  $w - 1$  positions away. Given a set bit in the bit string, its position in the window that contains it determines whether or not it can be recovered. This only depends on the values of the  $w - 1$  bits before it and after it. This means that after a renewal, we can calculate the probability of seeing a string of  $j > 0$  zeroes followed by a short window by looking at the set of size  $j + w$  bitstrings.

The second observation is that this probability  $p_j$  is equal to  $\frac{|C_j|}{2^{j+w}}$  where  $|C_j|$  is equal to the number of elements in

$$C_j = \{x_0, \dots, x_{j-1} \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0, 1\}; 0 \leq k \leq j-1, 0 \leq \ell \leq w-2\},$$

that is, the set of  $j + w$  bitstrings which after the side-channel attack will show an  $\underline{1}$  at position  $j$ . We will now prove our theorem by induction. For the base case, let  $j = 0$ , then the  $w - 1$  bits after the multiplication should be the all-zero bitstring. This means  $C_0 = 1$  and  $p_0 = \frac{1}{2^w}$ .

Suppose now have for  $j > 0$

$$C_j = 1 + \sum_{i=1}^j 2^{2i-1},$$

and therefore we have the following recursive formula

$$C_j = C_{j-1} + 2^{2j+1}.$$

Then by conditioning on the first bit  $x_0$  we get

$$\begin{aligned} C_{j+1} &= \{x_0 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0, 1\}\} \\ &= \{0 | x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0, 1\}\} \\ &\quad \cup \{1 | x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0, 1\}\} \end{aligned}$$

Since when  $x_0 = 0$ , we get the same conditions as in the case of a window of size  $j$ , we have

$$|\{0 | x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell \in \{0, 1\}\}| = |C_j|.$$

In the case that  $x_0 = 1$ , we have to count the set

$$\{1 | x_1 \dots x_j \underline{1} y_0 \dots y_{w-2} | x_k, y_\ell\}$$

and this will always have multiplication bit  $\underline{1}$  at position  $j + 1$  if  $y_0, \dots, y_{w-j-2}$  are an all-zero string, which leaves  $2^j * 2^{j+1} = 2^{2j+1}$  possibilities for the remaining bits.

This means that  $C_{j+1} = C_j + 2^{2j+1} = 1 + \sum_{i=1}^{j+1} 2^{2i-1}$ , from which our theorem follows.  $\square$

## C Learning all learnable bits

Section 3.1 describes how to learn some bits of the secret exponent, given the sequence of square and multiply operations performed by Algorithm 2, using four simple rewrite rules. It turns out that one can learn further bits using a stateful algorithm. Consider, for example, the bit string

00010010001000100010001010000010001,

producing the sequence

ssssmsssmsssmsssmsssmsssmsssmsssmsssmsssmsssmsssm

which is first converted to

xxx~~xxxxxxx~~xxx~~xxxxxxx~~xxx~~xxxxxxx~~xxx~~xxxxxxx~~

and then rules 0–3 recover these bits:

xxx~~1~~xx~~1~~0xx~~1~~xxx~~1~~xxx~~1~~xxx~~1~~x~~1~~00xxx~~1~~xxx~~1~~  
a ↑ b c d e

But every bit sequence leading to the above sequence of squares and multiplies must have a 1 in the position marked with an arrow.

To see why the most significant bit of the multiplier for multiplication  $b$  must be there, consider the alternatives. It cannot be one bit earlier, as then it would be included in the window for  $a$ . It also cannot be later, i.e. a multiplication with 3 or 1, as then the same would hold for  $c$ , and hence  $d$ , which is not possible, as otherwise the multiplication at  $d$  would include the 1 at  $e$ .

These missed opportunities are rare and do not significantly affect the efficiency of the pruning, but in the interest of completeness, we provide an algorithm to recover these additional bits and prove that it finds all bits whose values can be known with certainty.

### C.1 Possible window widths

We can find out all knowable bits if we keep track of the possible widths of the multiplier of each observed multiplication. Consider a multiplication at position  $i = 5$ , i.e. `xxx1xxxx`. For  $w = 4$ , this corresponds to one of four cases:

- Case 1: `x1xx1xxxx`, multiplier width: 4
- Case 2: `xx1x10xxxx`, multiplier width: 3
- Case 3: `xxx1100xxx`, multiplier width: 2
- Case 4: `xxxx1000xx`, multiplier width: 1

The key idea is now that **multipliers do not overlap**. More precisely, if the multiplier of the multiplication at position  $i$  has width  $m_i$  and the multiplier of the multiplication at position  $j$  has width  $m_j$  (with  $i > j$ ), then

$$i + m_i - w \geq j + m_j.$$

From this, we can derive a simple linear-time algorithm that calculates the possible multiplier widths for each multiplication. We define

$m_i^-$  to be the smallest possible width of the multiplication at position  $i$ , and  $m_i^+$  to be the largest possible width of the multiplication at position  $i$ .

Let  $M = \{k_0, k_1, \dots, k_n\}$  be the positions of the multiplications, with  $k_0 > \dots > k_M$ . Then

$$m_{k_i}^- = \begin{cases} 1, & \text{if } i = n \\ \max(1, k_{i+1} + m_{k_{i+1}}^- + w - k_i), & \text{otherwise, and} \end{cases}$$

$$m_{k_i}^+ = \begin{cases} w, & \text{if } i = 0 \\ \min(w, k_{i-1} + m_{k_{i-1}}^+ - w - k_i), & \text{otherwise.} \end{cases}$$

Note that  $m^-$  can be calculated going from right to left, while  $m^+$  can be calculated going from left to right.

Given  $m^-$  and  $m^+$ , we can calculate all the knowable bits of the input sequence. Let us represent this by  $b_i \in \{0, 1, x\}$ , defined as follows:

$$b_i = \begin{cases} 1, & \text{if } i \in M \\ 1, & \text{else if } j + m_j^- - 1 = i = j + m_j^+ - 1 \text{ for some } j \in M \\ x, & \text{else if } j + m_j^- - 1 \leq i \leq j + m_j^+ - 1 \text{ for some } j \in M \\ 0 & \text{otherwise.} \end{cases}$$

The first case corresponds to Rule 0 in Section 3.1 (last bit), the second to Rule 2 (first bit) and the last case to Rules 1 and 3 (trailing and leading zeroes).

**Theorem 4.** *This algorithm is correct and complete.*

This means not only that the input sequence has bit  $i$  set according to  $b_i$ , but also that if  $b_i = x$ , then there are two input sequences that produce the given sequence of squares and multiplies and differ at bit  $i$ . In other words: All knowable bits are known.

*Proof (Sketch).* Correctness follows by construction of the algorithm. The algorithm is complete because for every unknown bit ( $x$ ) we can construct two sequences, one with a 1 and one with a 0 in that spot, that map to the same square-and-multiply sequence.

**Example.** For the input above, with  $w = 4$ , we learn

```

Input: xxxxxxxxxx
m^-:   4 3 3 3 3 3 1 1 1
m^+:   4 3 3 3 3 3 1 3 3
b_i:   1xx11x101x101x101x101x101000xx10xx1.
    
```

## C.2 Self-information

Calculating  $m^+$  and  $m^-$  not only allows us to read off all knowable bits, it also allows us to calculate the number of input sequences that produce the given sequence of squares and multiplies.

We define the function  $f(i, z)$  to be the number of colliding sequences for the output including the window at position  $k_i$ , under the additional constraint that the earliest 1 occurs at position  $z$ . This function is recursively defined (and admits to a straight-forward efficient dynamic programming implementation). The base case is  $f(i, z) = 1$  for  $i > n$ , otherwise we have

$$f(i, z) = \sum_{m=m_{k_i}^-}^{m_{k_i}^+} [k_i + m - 1 \leq z] \cdot 2^{\max(0, m-2)} \cdot f(i+1, k_i + m - 1 - w)$$

where

- the sum iterates over all possible multiplier widths at this position,
- the characteristic function selects those where the first 1 of the window is after  $z$ ,
- the power of two takes into account the unknown bits in this window,
- and the recursive call goes to the next window, updating the constraint on the next allowed 1.

The total number of colliding sequences is then  $f(0, l)$ , where  $l$  is the length of the input, and the self-information of the sequence is  $I_s = -\log p_s = -\log \frac{f(0, l)}{2^l}$ .

**Example.** For the example above, this yields 1408 possible input sequences, a self-information of 24.5, and hence 0.70 bits of self-information per bit of input. Note that 1408 is not simply two to the number of  $x$ : not all assignments to yield valid input sequences.

## D Calculating the collision entropy of a leak

From [Theorem 3](#) we learn that the expected search tree size depends on the *collision entropy rate*  $H$  of the square-and-multiply sequence. While we are not able to give a closed formula of  $H$  in the window width  $w$ , we can calculate it for a given  $w$ . The individual steps are

1. Create a Mealy automaton that converts the bits of the secret exponent into a sequence of squares and multiplies.
2. Turn this automaton into an equivalent Hidden Markov Chain, by delaying the output by  $w$  bits.
3. Calculate the collision entropy of the Markov chain by finding the stationary distribution of the colliding process.

Just as [Theorem 3](#) applies to partial information beyond sliding windows, we believe that this approach will be useful to analyze other information leaks.

### D.1 A Mealy automaton

We can model the sequence of squares and multiplies performed by [Algorithm 2](#) as a Mealy automaton with input alphabet  $\Sigma = \{0, 1\}$  and output alphabet

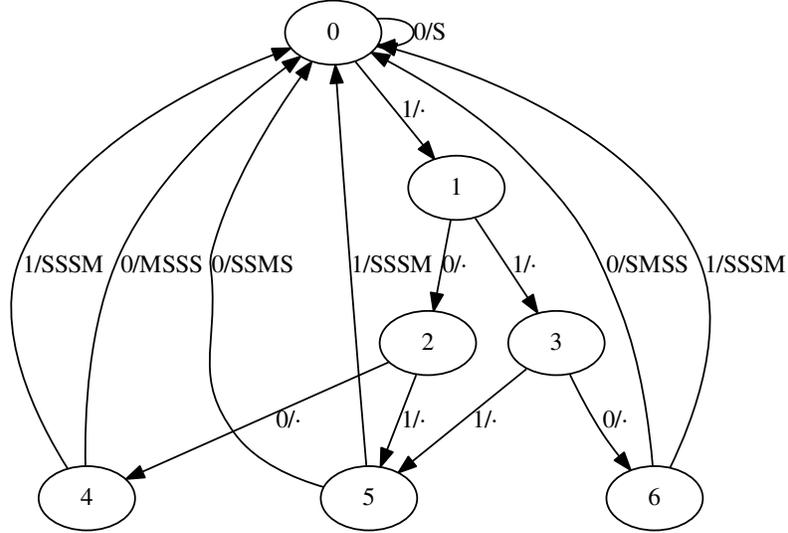


Fig. 9: Mealy automaton,  $w = 4$

$\Gamma^* = \{S, M\}^*$  (where M represents a combined square-and-multiply sequence). Because the algorithm may scan up to the next  $w$  bits, we can use the  $2^{w-1}$  states  $S = \{0, \dots, 2^{w-1} - 1\}$  to build the automaton in a straight-forward manner, where the state number directly encodes the bits scanned:

State	Input	Output	Next state
0	0	S	0
0	1		1
$0 < x < 2^{w-2}$	0		$2 \cdot x$
$0 < x < 2^{w-2}$	1		$2 \cdot x + 1$
$2^{w-2} \leq x < 2^{w-1}$	0	SM( $2 \cdot x + 1$ )	0

Once  $w$  bits are scanned into  $x$ , the sequence of squares and multiplies corresponding to  $x$  is printed, defined by

$$SM(x) := S^{w-i-1}MS^i,$$

where  $i$  is the position of the last 1 in  $x$ , and the algorithm continues again in state 0.

Some states are actually equivalent, such as 5 and 7 for  $w = 4$ , because the algorithm does not distinguish between the bit sequence 101 and 111, and a

standard automaton reduction algorithm can coalesce these nodes. For  $w = 4$ , this the resulting automaton is shown in [Figure 9](#).

## D.2 Mealy automaton to Markov chain

Such a Mealy automaton is not yet ideal to analyze, in particular due to the varying length of the output on each transitions. We would prefer to deal with a Markov chain where every state corresponds to one exactly output symbol. While this is obviously not the case for the automaton constructed above, which may have to scan  $w$  bits before producing output, we can make it so by delaying the output of the bits by  $w - 1$  steps.

In all generality: Given a Mealy automaton with state space  $S$ , input alphabet  $\Sigma$ , output alphabet  $\Gamma^*$ , output function  $\rho: S \times \Sigma \rightarrow \Gamma^*$  and transition function  $\delta: S \times \Sigma \rightarrow S$ , we can construct a Markov chain with state space  $S' := S \times \Gamma^*$  which has transitions

$$(\gamma_1 \overline{\gamma_2}, s) \longrightarrow (\overline{\gamma_2} \rho(s, i), \delta(s, i))$$

for  $\gamma_1 \in \Gamma$ ,  $\overline{\gamma_2} \in \Gamma^*$ ,  $s \in S$  and  $i \in \Sigma$  with probability  $\frac{1}{|\Sigma|}$ . This is a hidden state Markov chain, where only the first symbol  $\gamma_1$  of a state  $(\gamma_1 \overline{\gamma_2}, s)$  is visible. We restrict the Markov chain to the states reachable from a sensible chosen start state; in our case that is  $(M^w, 0)$ .

This process turns the Mealy automaton from [Figure 9](#) into the Markov chain shown in [Figure 10](#). The colors indicate the visible state of each node: green for M, blue for S.

At this point it becomes irrelevant which input bit (0 or 1) causes a transition, as we are interested in the distribution of the output.

## D.3 Collision entropy of a Markov chain

The final step is to calculate the collision entropy of a hidden Markov chain. To that end, we look at two independent instances of the Markov chain, find the stationary distribution of still-colliding states, which is a straight-forward eigenvector problem. The collision entropy can be read off the corresponding eigenvalue.

**Theorem 5.** *Consider a Markov chain  $(X_n)$  with state space  $S$ , observed through  $\rho: S \rightarrow \Gamma$ , and transition probability  $p: S \times S \rightarrow \mathbb{R}$ . Let  $C := \{(s_1, s_2) \in S \times S \mid \rho(s_1) = \rho(s_2)\}$ . The equation*

$$N \cdot \pi((s_1, s_2)) = \sum_{(s'_1, s'_2) \in C} \pi((s'_1, s'_2)) \cdot p(s'_1, s_1) \cdot p(s'_2, s_2)$$

for all  $(s_1, s_2) \in C$  describes an eigenvector  $\pi$  (which is a probability distribution on  $C$ ) with eigenvalue  $N$ . Then the collision entropy rate of the hidden Markov chain  $(\rho(X_n))$  is  $H = -\log N$ .

*Proof.* To model two independent executions of  $(X_n)$  we consider the product Markov chain  $(X_n^2)$  with state space  $S \times S$  and transition probabilities

$$p((s'_1, s'_2), (s_1, s_2)) = p(s'_1, s_1) \cdot p(s'_2, s_2).$$

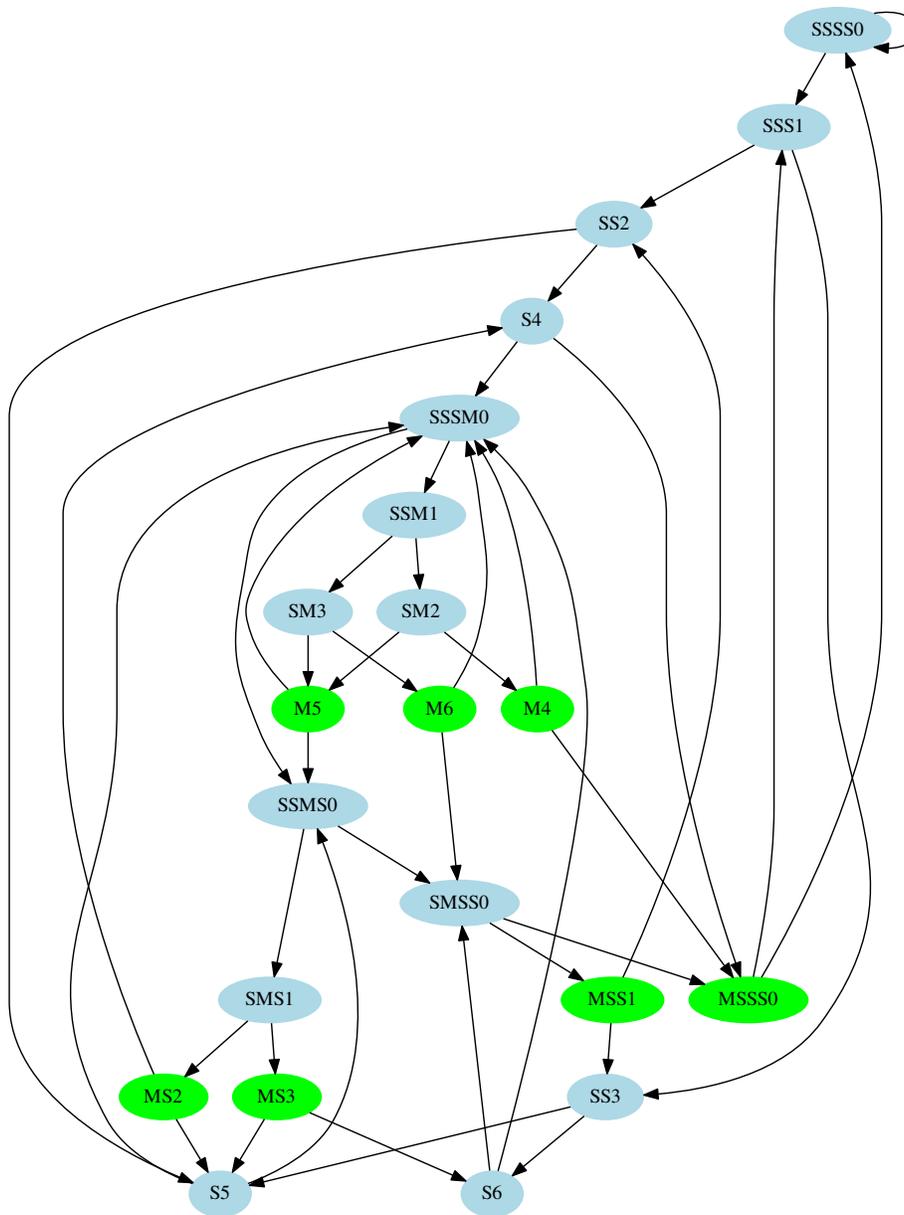


Fig. 10: Markov chain,  $w = 4$

The collision entropy rate of the original hidden Markov chain ( $\rho(X_n)$ ) is defined as

$$H = \lim_{n \rightarrow \infty} -\frac{1}{n} \cdot \log P(\rho(X_i) = \rho(X'_i), \forall i \leq n)$$

which we can write as

$$= \lim_{n \rightarrow \infty} -\frac{1}{n} \cdot \log P(X_i^2 \in C, \forall i \leq n).$$

By the Stolz-Cesàro theorem we have

$$\begin{aligned} &= \lim_{n \rightarrow \infty} -(\log P(X_i^2 \in C, \forall i \leq n+1) - \log P(X_i^2 \in C, \forall i \leq n)) \\ &= \lim_{n \rightarrow \infty} -\log P(X_{n+1}^2 \in C \mid X_i^2 \in C, \forall i \leq n) \\ &= -\log \lim_{n \rightarrow \infty} P(X_{n+1}^2 \in C \mid X_i^2 \in C, \forall i \leq n) \end{aligned}$$

Now we define

$$N := \lim_{n \rightarrow \infty} P(X_{n+1}^2 \in C \mid X_i^2 \in C, \forall i \leq n)$$

and for  $(s_1, s_2) \in C$

$$\pi((s_1, s_2)) := \lim_{n \rightarrow \infty} P(X_n^2 = (s_1, s_2) \mid X_i^2 \in C, \forall i \leq n).$$

From

$$\begin{aligned} &P(X_{n+1}^2 \in C \mid X_i^2 \in C, \forall i \leq n) \cdot P(X_{n+1}^2 = (s_1, s_2) \mid X_i^2 \in C, \forall i \leq n+1) \\ &= P(X_{n+1}^2 = (s_1, s_2) \mid X_i^2 \in C, \forall i \leq n) \\ &= \sum_{(s'_1, s'_2) \in C} P(X_{n+1}^2 = (s_1, s_2) \mid X_n^2 = (s'_1, s'_2) \wedge X_i^2 \in C, \forall i \leq n) \\ &\quad \cdot P(X_n^2 = (s'_1, s'_2) \mid X_i^2 \in C, \forall i \leq n) \\ &= \sum_{(s'_1, s'_2) \in C} P(X_{n+1}^2 = (s_1, s_2) \mid X_n^2 = (s'_1, s'_2)) \cdot P(X_n^2 = (s'_1, s'_2) \mid X_i^2 \in C, \forall i \leq n) \\ &= \sum_{(s'_1, s'_2) \in C} p((s'_1, s'_2), (s_1, s_2)) \cdot P(X_n^2 = (s'_1, s'_2) \mid X_i^2 \in C, \forall i \leq n) \end{aligned}$$

we obtain, by taking the limit  $n \rightarrow \infty$  on both sides, that  $N$  and  $\pi$  are indeed solutions to the equation in the theorem, and in particular,  $H = -\log N$ .

Identifying sufficient assumptions on the Markov chain and  $\rho$  which ensure convergence of all limits and a unique solution to the eigenvalue problem remains future work.

#### D.4 Results

We have applied this process to both left-to-right and right-to-left sliding window leaks for various window widths, and obtain the following results, which are also plotted in [Figure 11](#). We can see that for  $w = 4$ , compared to right-to-left, left-to-right brings the collision entropy rate over the threshold of 0.5, and for  $w = 5$  moves it very close.

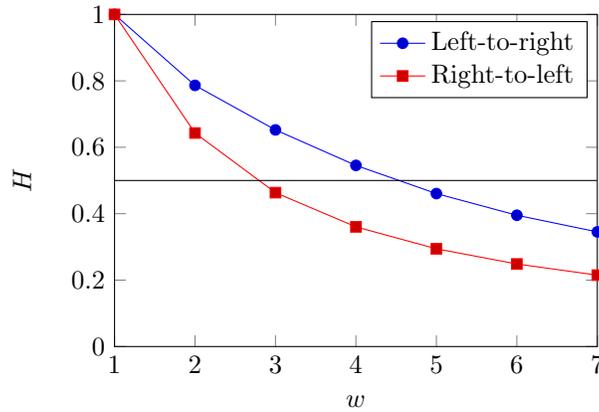


Fig. 11: Collision entropy rate

$w$	1	2	3	4	5	6	7
left-to-right	1.000	0.786	0.652	0.545	0.460	0.395	0.345
right-to-left	1.000	0.643	0.463	0.360	0.294	0.249	0.215

### D.5 Why not Shannon entropy?

In [Theorem 3](#) and this section, we are concerned with the collision entropy, also known as Rényi entropy [22], rather than the more common Shannon entropy. Why is this the case? Let us recall the definition of the Shannon entropy  $H_1$  and the collision entropy  $H_2$  for a random variable  $X$  with  $p_x = P(X = x)$ :

$$H_1 := - \sum_x p_x \cdot \log p_x \quad H_2 := - \log \sum_x p_x^2$$

We always have  $H_2 \leq H_1$ , and equality for an uniformly distributed  $X$ .

Note that  $\sum_x p_x^2 = P(X_1 = X_2)$ , the probability of two independent random variables with the same distribution as  $X$  to coincide, hence the name *collision entropy*. And it is exactly when there is a collision between the partial information from the actual secret key and from the guessed suffix when the pruning algorithm will carry on.

More on the difference between the various entropy measures in the context of cryptography can be found in [24].

## E Proof of [Theorem 3](#)

We can prove [Theorem 3](#) in a more general setting than just left-to-right leakage: We assume an oracle that, given a hypothesized suffix  $p'$  of  $d_p$  of length  $n$  tells us whether this is indeed a possible suffix. This framework allows us to model various forms of partial information about  $d_p$ , including randomly known bits and square and multiply sequences.

In notation:  $A_n(d_p, d'_p)$  holds if the oracle does not refute the  $n$  least significant bits of  $d'_p$  as a possible suffix of  $d_p$ .

**Assumptions.** Our proof makes the following assumptions:

1. If we guessed  $d'_p$  and  $d'_q$  wrong, and the least significant incorrectly guessed bit is at position  $i$ , then we assume that the RSA equation

$$\text{RSA}_n(d_p, d_q) := (ed_p - 1 + k_p)(ed_q - 1 + k_q) = k_p k_q N \bmod 2^n$$

holds with probability  $2^{-(n-i)}$ .

2. In the above situation, we assume that the oracles for  $d_p$  and  $d_q$  accept the guesses independently at random with the above probability after the first incorrectly guessed bit.
3. The probability that the oracle accepts an  $n$ -bit suffix where the last  $i$  bits are correct is less than the probability that the oracle accepts a random  $(n-i)$ -bit suffix:

$$P(A_n(d_p, d'_p) \mid \text{last } i \text{ bits of } d'_p \text{ are correct}) \leq P(A_{n-i}(d_p, d'_p))$$

4. The probability that the oracle accepts yet another bit, after having accepted  $n$  bits, is constant:

$$P(A_{n+1}(d_p, d'_p) \mid A_n(d_p, d'_p)) = 2^{-H}.$$

In this case

$$P(A_n(d_p, d'_p)) = 2^{-nH}$$

for some  $H \in [0, 1]$ . The constant  $H$  can be understood as the collision entropy rate of the oracle.

The proof requires that these assumptions hold for almost all  $n$ . If they do not hold for small  $n$ , then the inequality [Theorem 3](#) still holds up to a constant factor.

**Proof.** We use some additional notation: Let  $k$  be the key size, i.e. the lengths of both  $p, q \in \{0, 1\}^k$ . We use  $(x, y) \stackrel{n}{\neq} (x', y')$  to denote that the  $n$  least significant bits of  $x$  and  $x'$  and of  $y$  and  $y'$  are equal, and that the  $n$ th bit of either  $x$  and  $x'$  or  $y$  and  $y'$  differ.

Define the predicate OK to be true if the least significant  $n$  bits of  $d'_p$  and  $d'_q$  are potentially correct suffixes of the actual values  $d_p$  and  $d_q$ . In our case,

$$\text{OK}_n(d_p, d_q, d'_p, d'_q) := A_n(d_p, d'_p) \wedge A_n(d_q, d'_q) \wedge \text{RSA}_n(d'_p, d'_q).$$

where  $\text{RSA}_n(d_p, d_q) := (ed_p - 1 + k_p)(ed_q - 1 + k_q) = k_p k_q N \bmod 2^n$ .

In general, the size of a search tree with pruning is

$$S(d_p, d_q) := \sum_{n=0}^{k-1} \sum_{d'_p, d'_q \in \{0, 1\}^n} [\text{OK}_n(d_p, d_q, d'_p, d'_q)]$$

This notion of size counts all nodes of the tree that are not pruned, i.e. excludes the leaves where a contradiction is observed.

We can further split this sum up according to whether we guessed all bits correctly (in which case  $[\text{OK}_n(d_p, d_q, d'_p, d'_q)] = 1$ ) or if not, when was the first bit that we guessed incorrectly:

$$S(d_p, d_q) = \sum_{n=0}^{k-1} \left( 1 + \sum_{i=0}^n \sum_{d'_p, d'_q \in \{0,1\}^n} [(d'_p, d'_q) \stackrel{i}{\neq} (d_p, d_q) \wedge \text{OK}_n(d_p, d_q, d'_p, d'_q)] \right).$$

Reordering the sum to iterate over the position of the incorrectly guessed bit first, we get

$$S(d_p, d_q) = k + \sum_{i=0}^{k-1} \sum_{n=i}^{k-1} \sum_{d'_p, d'_q \in \{0,1\}^n} [(d'_p, d'_q) \stackrel{i}{\neq} (d_p, d_q) \wedge \text{OK}_n(d_p, d_q, d'_p, d'_q)].$$

At this point we use the assumption that once bit  $i$  of  $d_p$  or  $d_q$  has been guessed incorrectly, the following  $n - i$  bits pass the RSA check with uniform probability  $2^{-(n-i)}$  and furthermore the likelihood that the oracle accepts  $d'_p$  is independent of whether the oracle accepts  $d'_q$ . Formally, this means that, over all keys  $d_p$  and  $d_q$ , we have

$$\begin{aligned} \sum_{d'_p, d'_q \in \{0,1\}^n} P((d'_p, d'_q) \stackrel{i}{\neq} (d_p, d_q) \wedge \text{OK}_n(d_p, d_q, d'_p, d'_q)) \\ = \sum_{d'_p, d'_q \in \{0,1\}^{n-i}} 2^{-(n-i)} \cdot P(A_{n-i}(d_p, d'_p)) \cdot P(A_{n-i}(d_q, d'_q)) \end{aligned}$$

Substituting into the above, we get

$$\begin{aligned} E(S(d_p, d_q)) &= k + \sum_{i=0}^{k-1} \sum_{n=0}^{k-1-i} \sum_{d'_p, d'_q \in \{0,1\}^n} 2^{-n} \cdot P(A_n(d_p, d'_p)) \cdot P(A_n(d_q, d'_q)) \\ &\leq k + \sum_{i=0}^{k-1} \sum_{n=0}^{k-1} \sum_{d'_p, d'_q \in \{0,1\}^n} 2^{-n} \cdot P(A_n(d_p, d'_p)) \cdot P(A_n(d_q, d'_q)) \\ &= k \cdot \left( 1 + \sum_{n=0}^{k-1} \sum_{d'_p, d'_q \in \{0,1\}^n} 2^{-n} \cdot P(A_n(d_p, d'_p)) \cdot P(A_n(d_q, d'_q)) \right) \end{aligned}$$

By the last assumption this simplifies to

$$\begin{aligned} E(S(d_p, d_q)) &\leq k \cdot \left( 1 + \sum_{n=0}^{k-1} \sum_{d'_p, d'_q \in \{0,1\}^n} 2^{-n} \cdot 2^{-nH} \cdot 2^{-nH} \right) \\ &= k \cdot \left( 1 + \sum_{n=0}^{k-1} 2^{n \cdot (1-2H)} \right) \\ &= k \cdot \left( 1 + \frac{1 - 2^{k \cdot (1-2H)}}{1 - 2^{1-2H}} \right). \end{aligned}$$

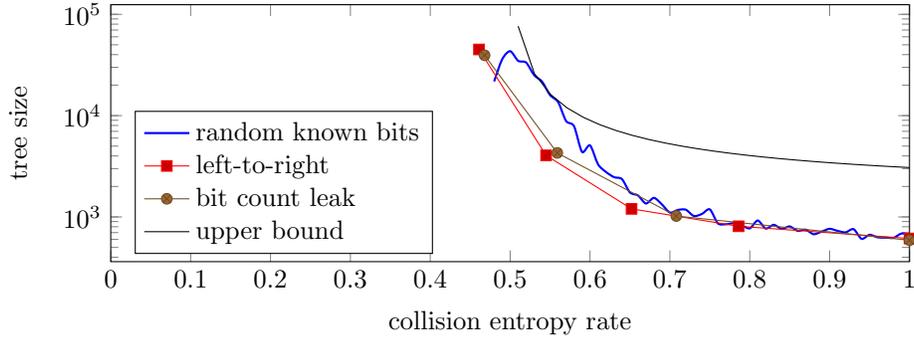


Fig. 12: We experimentally validated [Theorem 3](#) by computing the predicted upper bound and the measured tree sizes for 50 random 1024-bit keys.

**Validation.** In order to empirically validate [Theorem 3](#), we plot the predicted upper bound as well as the measured tree sizes in [Figure 12](#), with pruning based on

- randomly known bits, similar to the setting of [\[12\]](#),
- left-to-right sliding window sequences of squares and multiply of varying window width,
- a hypothetical leak where the attacker learns the number of set bits, but not their positions, per fixed window.

We find that the graph does not refute our theorem.

The graph measures the tree size for 50 randomly chosen 1024 bit keys. The search was aborted if the tree size exceeds 100,000 nodes. Note that close to  $H = 0.5$  and further left, the plots show the average search tree size over those attempts that finished within the limit.