More on sliding right

Joachim Breitner

DFINITY Foundation, joachim@dfinity.org

Abstract

This text can be thought of an "external appendix" to the paper *Sliding right into disaster: Left-to-right sliding windows leak* by Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal and Yuval Yarom [1, 2], and goes into the details of

- an alternative way to find the knowable bits of the secret exponent, which is complete and can (in rare corner cases) find more bits than the rewrite rules in Section 3.1 of [1],
- an algorithm to calculate the collision entropy H that is used in Theorem 3 of [1], and
- a proof of Theorem 3.

All errors, typos and irrelevancies in the present paper are purely Joachim Breitner's fault, and not of any of the other authors of [1, 2].

The Haskell implementation of algorithm can be found at https://github.com/nomeata/slidingright.

1 Knowing all knowable bits

Section 3.1 of [1] describes how to learn some bit 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 these rules are not able to recover *all* known bits. Consider, for example, the bit string

$$0001001000100010001000101000001000 1,$$

producing the sequence

$$\texttt{sssmsssmsssssmsssssmsssssmsssssmssmsssssssmssssm}$$

which is first converted to

$$\texttt{xxx}\underline{\texttt{x}}\texttt{xx}\underline{\texttt{x}}\texttt{xxx}\underline{\texttt{x}}\texttt{xxx}\underline{\texttt{x}}\texttt{xxx}\underline{\texttt{x}}\texttt{xxx}\underline{\texttt{x}}\texttt{xx}\underline{\texttt{x}}\texttt{xxxxxx}\underline{\texttt{x}}\texttt{xxx}\underline{\texttt{x}}$$

and then rules 0–3 recover these bits:

$$\texttt{xxx}\underline{\texttt{1}}\texttt{xx}\underline{\texttt{1}}\texttt{0xx}\underset{a}{\underline{\texttt{1}}}\texttt{xxx}\underset{\uparrow}{\underline{\texttt{1}}}\texttt{xxx}\underset{b}{\underline{\texttt{1}}}\texttt{xxx}\underset{c}{\underline{\texttt{1}}}\texttt{xxx}\underset{d}{\underline{\texttt{1}}}\texttt{x}\underset{e}{\underline{\texttt{1}}}\texttt{00xxx}\underline{\texttt{1}}\texttt{xxx}\underline{\texttt{1}}$$

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 the bits which finds all knowable bits.

1.1 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. xxxx1xxxxx. For $w = 4$, this corresponds to one of four cases:

Case 1: x1xx1xxxxx, 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 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 $M = \{k_0, k_1, \ldots, k_n\}$ be the positions of the multiplications, with $k_0 > \cdots > k_M$. 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 $b_i \in \{0, 1, \mathrm{x}\}$, defined as follows:

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

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

Theorem 1 *This algorithm is correct and complete.*

This means not only that the input sequence have bit i set according to b_i, but also that if $b_i = \mathrm{x}$, then there are two input sequence 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. Completeness, 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:	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
m^-:	4 3 3 3 3 3 1 1 1
m^+:	4 3 3 3 3 3 1 3 3
b_i:	1xx11x101x101x101x101x101000xx10xx1.

1.2 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

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

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.

2 Calculating the collision entropy of a leak

From Theorem 3 of [1] 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.

2.1 A Mealy automaton

We can model the sequence of squares and multiplies performed by Algorithm 2 in [1] as a mealy automaton with input alphabet $\Sigma = \{0,1\}$ and output alphabet $\Gamma^* = \{\mathrm{S},\mathrm{M}\}^*$ (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 $S = \{0,\ldots,2^{w-1}-1\}$ 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		$2 \cdot x + 1$
$2^{w-2} \le x < 2^{w-1}$	0	$\mathrm{SM}(2 \cdot x + 1)$	0

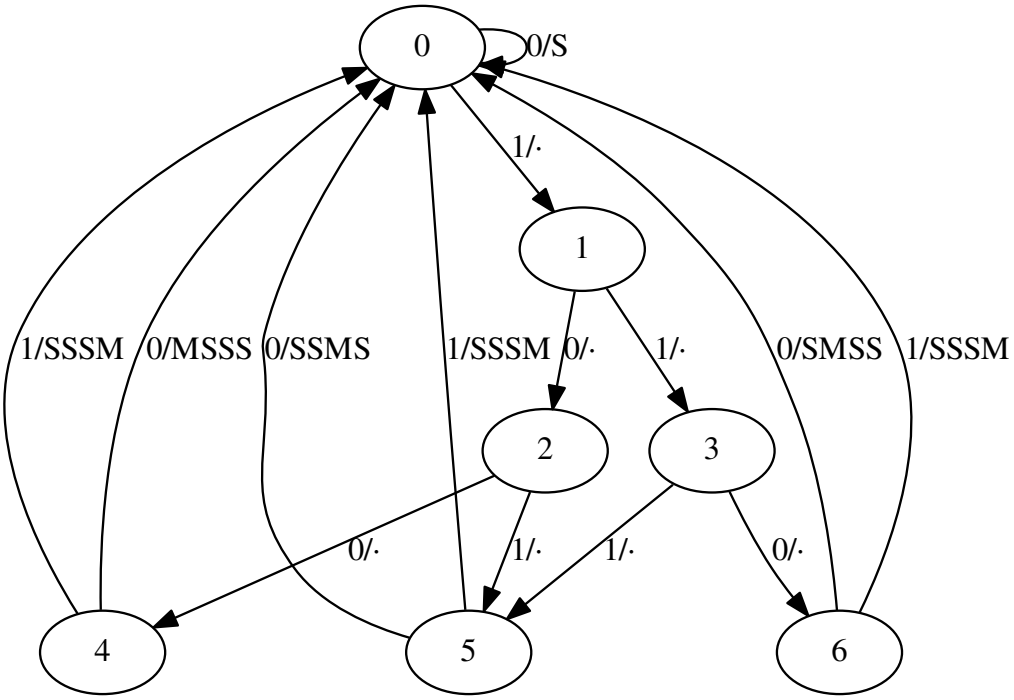

Figure 1: Mealy automaton, $w = 4$

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

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

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 1.

2.2 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.

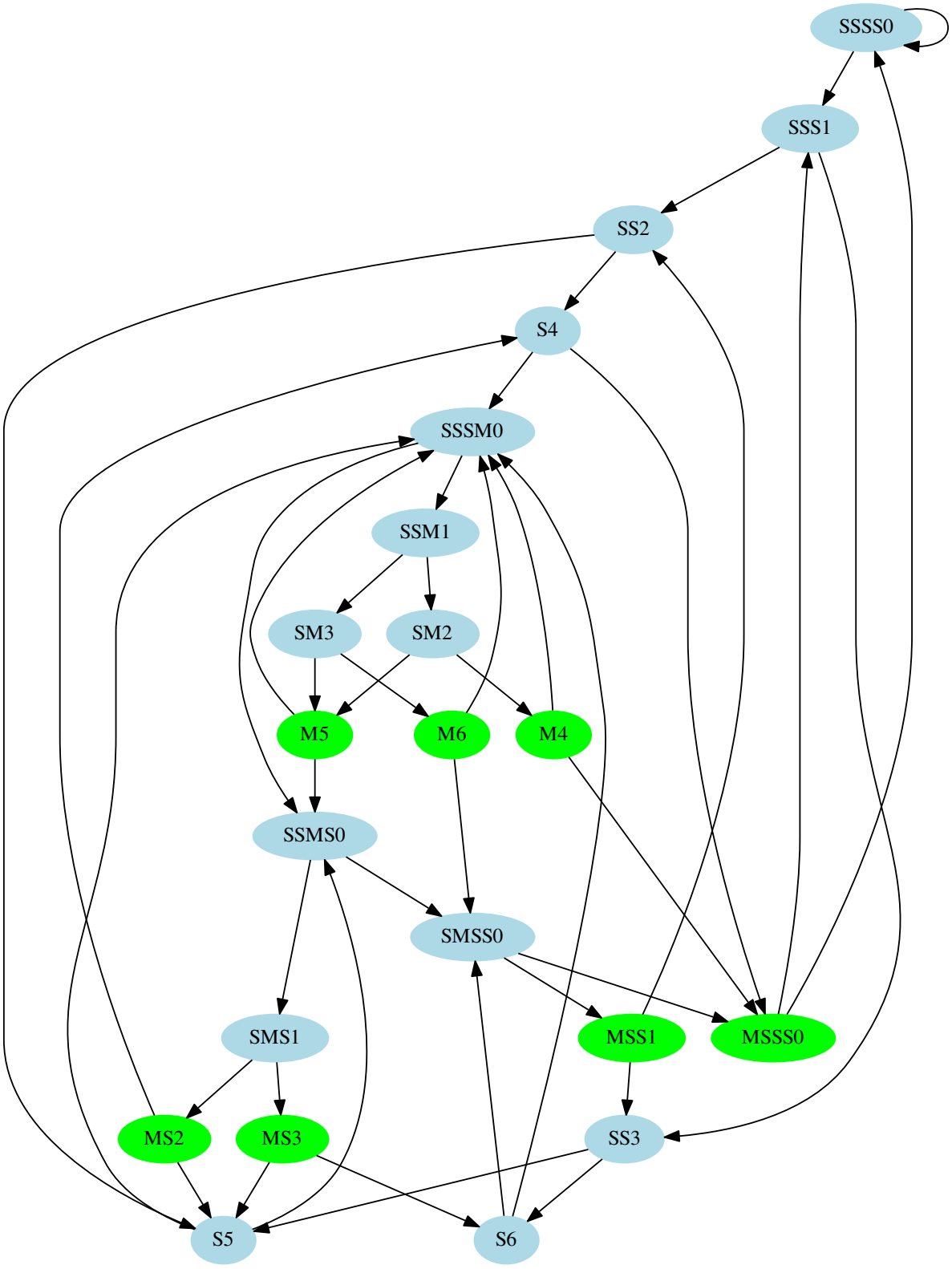

Figure 2: Markov chain, $w = 4$

In all generality: Given a Mealy automaton with state space S, input alphabet Σ, output alphabet Γ^*, output function $\rho : S \times \Sigma \to \Gamma^*$ and transition function $\delta : S \times \Sigma \to S$, 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 γ_1 of a state $(\gamma_1 \overline{\gamma_2}, s)$ is visible. We restrict the Markov chain to the states reachable from a sensibly chosen start state; in our case that is $(\mathrm{M}^w, 0)$.

This process turns the mealy automaton from Figure 1 into the Markov chain shown in Figure 2. 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.

2.3 Collision entropy of a Markov chain

The final step is to calculate the collision entropy of a hidden Markov chain. To that end, we construct a Markov chain that simulates the evolution of two independent instances of the given Markov chain, find the stationary distribution of still-colliding states and calculate the probability of the two Markov chains producing different output in the next step. More theoretical details and generalizations of this novel approach can be found in [3].

We start with a Markov chain with state space S, transition probability $p : S \to S \to \mathbb{R}$ and visible state $\rho :: S \to \Gamma$.

The product Markov chain has state space $S \times S$ and transition probability

$$p\big((s_1, s_2), (s_1', s_2')\big) = p(s_1, s_1') \cdot p(s_2, s_2').$$

We find a probability distribution $P : C \to \mathbb{R}$ on the colliding states

$$C := \{(s_1, s_2) \in S \times S \mid \rho(s_1) = \rho(s_2)\}$$

that is stationary in the sense that if the product Markov chain starts in this distribution, takes a step and ends up in a state in C again, then it is again in this distribution. In other words, it satisfies the equation

$$P(s) = \frac{\sum\limits_{s' \in C} P(s') \cdot p(s', s)}{\sum\limits_{s', s \in C} P(s') \cdot p(s', s)}.$$

for all $s \in C$. The denominator re-normalizes the distribution, as the next state may not be in C. We can find this distribution as the eigenvector with the largest eigenvalue of the transition matrix of the product Markov chain with all transitions outside of C set to zero. (In contrast to the usual stationary distribution of a Markov chain, the eigenvalue will be smaller than 1.)

The collision probability rate is now simply the probability of the product Markov chain remaining in C, and hence the collision entropy rate is:

$$H = -\log \sum_{s', s \in C} P(s') \cdot p(s', s).$$

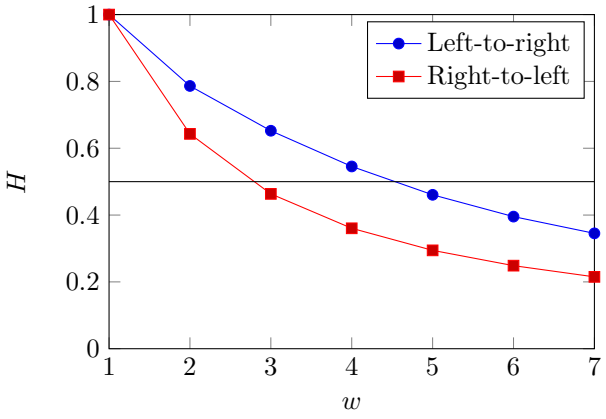

Figure 3: Collision entropy rate

2.4 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 3. We can see that for $w = 4$, compared to right-to-left, left-to-right hoists the collision entropy rate over the threshold of 0.5, and for $w = 5$ moves it very close.

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

2.5 Why not Shannon entropy?

In Theorem 3 and this section, we are are concerned with the collision entropy, also known as as Rényi entropy [5], 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 $p_x = P(X = x)$:

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

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 [6].

3 Proof of Theorem 3

We can prove Theorem 3 of [1] 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.

Assumption Our proof relies on these assumptions:

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

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

   holds to be $2^{-(n-i)}$.

2. In the above situation, we assume that the probability of the oracles for d_p and d_q to accept the guesses after the first wrongly guessed bit are independent.

3. The probability of the oracle accepting an n-bit suffix where the last i bits are correct is less than the probability of the oracle accepting a random $(n-i)$-bit suffix:

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

4. The probability of the oracle accepting a next bit is constant. In other words,

$$P(A_n(d_p, d_q)) = 2^{-nH}$$

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

It suffices if these assumptions hold for almost all n; if they do not hold for small n then the inequality in Theorem 3 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 $(x, y) \overset{n}{\neq} (x', y')$ 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.

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

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

where the predicate OK is 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,

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

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 $[\mathrm{OK}_n(d_p, d_q, d'_p, d'_q)] = 1$) or if not, when was the first bit that we guessed wrongly:

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

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

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

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

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

Plugged 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

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

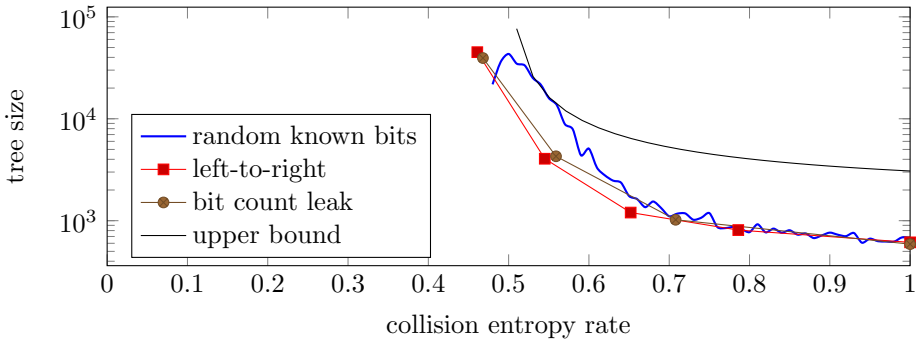

Figure 4: Validation of Theorem 3

Validation In order to empirically validate Theorem 3, we plot the predicted upper bound as well as the measured tree sizes in Figure 4, with pruning based on

- randomly known bits, similar to the setting of [4],
- left-to-right sliding window sequences of squares and multiply of varying window 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.

Acknowledgments

We would like to thank all the other authors of [1, 2] for the opportunity to collaborate on this paper. This work was done while the author was affiliated with the University of Pennsylvania, with support from by the National Science Foundation under grants 1319880 and 14-519.

References

[1] 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. Cryptology ePrint Archive, Report 2017/627, 2017. https://eprint.iacr.org/2017/627.

[2] 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*, volume 10529 of *Lecture Notes in Computer Science*, pages 555–576. Springer, 2017.

[3] Joachim Breitner and Maciej Skorski. Analytic formulas for renyi entropy of hidden markov models. *CoRR*, abs/1709.09699, 2017.

[4] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In *CRYPTO*, volume 5677 of *LNCS*, pages 1–17. Springer, 2009.

[5] 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.

[6] Maciej Skorski. A comprehensive comparison of shannon entropy and smooth renyi entropy. Cryptology ePrint Archive, Report 2014/967, 2014. https://eprint.iacr.org/2014/967.