Nostradamus goes Quantum

Barbara Jiabao Benedikt[1] (✉), Marc Fischlin[1], and Moritz Huppert[1]

[1] Cryptoplexity, Technische Universität Darmstadt, Germany
{barbara_jiabao.benedikt,marc.fischlin}@tu-darmstadt.de
moritz.huppert@proton.me

Abstract. In the Nostradamus attack, introduced by Kelsey and Kohno (Eurocrypt 2006), the adversary has to commit to a hash value y of an iterated hash function H such that, when later given a message prefix P, the adversary is able to find a suitable "suffix explanation" S with $\mathsf{H}(P\|S) = y$. Kelsey and Kohno show a herding attack with $2^{2n/3}$ evaluations of the compression function of H (with n bits output and state), locating the attack between preimage attacks and collision search in terms of complexity. Here we investigate the security of Nostradamus attacks for quantum adversaries. We present a quantum herding algorithm for the Nostradamus problem making approximately $\sqrt[3]{n} \cdot 2^{3n/7}$ compression function evaluations, significantly improving over the classical bound. We also prove that quantum herding attacks cannot do better than $2^{3n/7}$ evaluations for random compression functions, showing that our algorithm is (essentially) optimal. We also discuss a slightly less tight bound of roughly $2^{3n/7-s}$ for general Nostradamus attacks against random compression functions, where s is the maximal block length of the adversarially chosen suffix S.

Keywords: Hash function, herding attack, lower bound, Nostradamus, quantum, Grover

1 Introduction

Hash functions serve as a versatile tool in cryptography, thus coming with several security requirements like collision resistance, preimage resistance, or second preimage resistance. In 2006 Kelsey and Kohno [KK06] introduced a new kind of attack and security property for iterated hash functions H, based on a compression function h with state and output size n. The attack requires the adversary to first commit to a hash value y_{trgt} and later, after given a message prefix P, to find a message suffix S such that $\mathsf{H}(P\|S) = y_{\text{trgt}}$. The attack is known under the technical term *chosen-target forced-prefix* (CTFP) preimage attack, but is often referred to by the more picturesque title *Nostradamus attack*. The latter is via the connection to forecasting scenarios: The hash value can be seen as a commitment to a allegedly correct prediction of some event P in the future, which the attacker aims to attest by finding a suitable suffix S.

1.1 Herding Attacks

The so-called herding attack of Kelsey and Kohno [KK06] is a Nostradamus attack with roughly $\mathcal{O}(2^{2n/3})$ evaluations of the compression function h.[1] This is still far from the birthday bound for collision resistance, but it is clearly better than a preimage search with $\mathcal{O}(2^n)$ evaluations. The herding attack can be divided into two phases:

Offline phase: In the offline phase the adversary first determines the target hash value y_{trgt} and builds a diamond structure, which is a hash tree of height k. The tree connects 2^k distinct leaves to the root value y_{trgt} via iterating h on different message blocks. Kelsey and Kohno discuss that the overall effort to build such a tree is $\mathcal{O}(2^{(n+k)/2})$. Blackburn et al. [BSU12] later pointed out a flaw in the analysis and gave a bound of $\mathcal{O}(\sqrt{k} \cdot 2^{(n+k)/2})$.

Online phase: In the online phase the adversary is then presented the prefix P. It searches for a linking message part m_{link} to one of the leaves in the diamond structure, such that m_{link} and the message blocks on the tree path m_{path} yield the suffix $S = m_{\text{link}} \| m_{\text{path}}$. Kelsey and Kohno discuss that this step requires $\mathcal{O}(2^{n-k})$ evaluations of h.

Choosing $k = \frac{n}{3}$ then yields an overall effort of $\mathcal{O}\left(\sqrt{n} \cdot 2^{2n/3}\right)$ of both phases together.

We note that there are variations of the above fundamental attack, also discussed in [KK06]. One is to use expandable messages [Dea99,KS05] to accommodate variable-length suffixes when the message length is included in the padding. Such expandable messages can be used in combination with elongated diamond structures. These elongated structures significantly increase the suffix length, by a term 2^r for parameter r, but reduce the effort to roughly $2^{2n/3-2r/3}$. We do not look into such variations here, since our attacks already work well with basic diamond structures.

1.2 Quantum Herding Attack

By a result of Brassard et al. [BHT98] it is known that quantum computers facilitate the search for collisions in hash functions, reducing the effort from $\mathcal{O}(2^{n/2})$ in the classical setting to $\mathcal{O}(2^{n/3})$ in the quantum case. The algorithm itself is based on Grover's quantum search algorithm [Gro96]. The question we address in this work here is if quantum search or collision finding helps in improving herding attacks. Can we expect the same speed-up of a factor 2/3 in the exponent as in the collision case?

As a very fundamental result we first argue that quantum collision search gives an easy attack with $\mathcal{O}(2^{n/2})$ evaluations of h, without the need to construct a diamond structure. Namely, pick an arbitrary target value y_{trgt} and, once receiving the prefix P, use Grover's search algorithm to find the linking message block $m_{\text{link}} \in \{0,1\}^B$ of $B \gg n$ bits. If we assume that the hash function is

[1] Unless stated otherwise, all bounds refer to expected numbers of evaluations.

approximately regular, then there are roughly $t = 2^{B-n}$ such message blocks mapping to the target value. But then Grover's algorithm requires $\mathcal{O}(\sqrt{2^B/t}) = \mathcal{O}(2^{n/2})$ quantum evaluations of h to find m_{link}. This already improves over the classical bound (and requires no storage for the diamond structure).

The next, more elaborate attempt is to replace the collision search to create the diamond structure in the attack of Kelsey and Kohno [KK06] by a quantum algorithm. The improvement may, however, be less expedient than envisioned at first, because the original diamond structure generation throws many values in parallel and then "sieves" for a sufficient number of simultaneous collisions. For the quantum case we proceed step by step. Nonetheless, we show that with this approach we indeed achieve an improvement factor of 2/3 in the exponent compared to the classical attack, requiring $\mathcal{O}(2^{4n/9})$ compression function evaluation.

Our main result is an enhanced version of the quantum attack with a diamond structure. We show that we can actually build a diamond structure more efficiently if we wisely re-use some of the previous evaluations when searching for collisions. Optimizing the parameters we achieve an attack with $\mathcal{O}(\sqrt[3]{k} \cdot 2^{3n/7})$ evaluations of h. The bounds for the attacks are displayed in Figure 1.

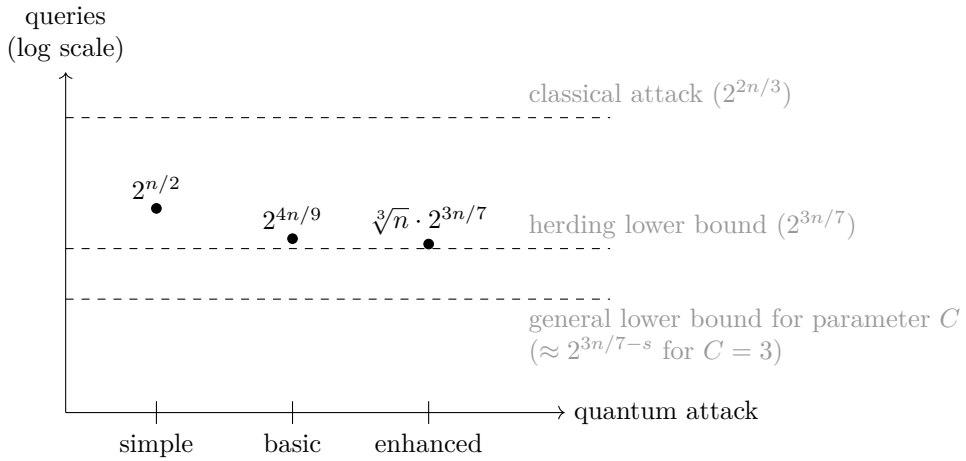

Fig. 1: Upper and lower bounds on expected number of compression function evaluations for quantum attacks (neglecting constants). The simple attack is the straightforward quantum attack, the basic attack uses a diamond structure formed by basic quantum collision search, and the enhanced attack optimizes generation of diamond structure. The parameter s denotes the maximal number of message blocks in the adversarial suffix S, n is the hash function's output size, and the parameter C denotes the number of multicollisions for the lower bound.

We have implemented our attacks in IBM's Qiskit software development kit.[2] The classical simulation of such quantum algorithms in Qiskit, however,

[2] Available via `https://git.rwth-aachen.de/marc.fischlin/quantum-nostradamus`.

restricts the number of available qubits. Therefore, we were only able to run our algorithms against a toy hash function with very limited block and output length $B = n = 8$, based on similar attempts in [ECBP+22,RBL+21]. For this hash function our experiments confirm that the enhanced attack is superior to the basic attack in terms of actual run time, mainly in the offline phase, with almost equal statistics in the online phase. Still, due to the restricted choice of n these results must be taken with prudence. This is the more true as the simple attack with Grover's algorithm outperforms both algorithms for $n = 8$, presumably because it does not require the additional overhead for the herding step.

Our attacks are primarily designed for iterated hash functions of the Merkle-Damgård type [Dam90,Mer90], such as SHA2 [Dan15]. The quantum herding attack is in principle also applicable to sponge-based hash functions [BDPV07], as we discuss in Section A. In this case, our simple quantum attack also yields a bound of $\mathcal{O}\left(2^{n/2}\right)$ for n-bit outputs. For the basic and enhanced attack the capacity c of the sponge becomes the relevant parameter for building the diamond structure with the hash collisions on the intermediate values, yielding the overall bounds $\mathcal{O}\left(2^{4c/9}\right)$ resp. $\mathcal{O}\left(\sqrt[3]{c} \cdot 2^{3c/7}\right)$. For SHA3 [Dwo15], however, we have $c = 2n$, such that the latter bounds are inferior to the one of the simple attack. For extendable output functions like SHAKE [Dwo15] the choice of the best attack depends on the relationship of n and c.

1.3 Quantum Lower Bounds

Can we go below the bound of $2^{3n/7}$ evaluations for our enhanced attack? We argue that for herding attacks this is impossible, at least generically. For this we use a lower bound of Liu and Zhandry [LZ19] for the quantum query complexity of finding C-collisions in random functions, i.e., C distinct values all mapping to the same function value. The bound states that one needs at least $\Omega(2^{n(1-\frac{1}{2^C-1})/2})$ queries to find such collisions. We argue below that a successful Nostradamus attack essentially allows to find such C-collisions such that the bound transfers to our scenario accordingly.

We first give a general lower bound for Nostradamus attacks for random function h, independently of how the adversary operates. The idea is as follows. Recall that the Nostradamus attacker first commits to the target hash value y_{trgt}, and in the second phase computes the suffix S for the given value P. We can hence repeat the second phase multiple times with different prefixes $P_1, P_2, \ldots$ to generate suffixes $S_1, S_2, \ldots$, such that all inputs $P_i \| S_i$ map to the same target hash values y_{trgt}. If we run the adversary sufficiently often, roughly $(C-1)^s$ times where s is the maximal number of blocks in the suffix S, then at some point we derive a C-collision. It follows that the Nostradamus attack must make at least $\Omega(2^{n(1-\frac{1}{2^C-1})/2})$ queries, divided by $(C-1)^s$. For $C = 3$ the bound simplifies to approximately $2^{3n/7-s}$.

We next argue that the factor $(C-1)^s$ can be avoided for herding attacks using a diamond structure. Specifically, we consider $C = 3$ and use a single solution S of the adversary together with the a specially crafted diamond structure

to form a 3-collision. Since we do not need repetitions the extra factor $(C-1)^s$ disappears. We hence turn a Nostradamus attack with diamond structures into a 3-collision finder, with the same number of compression function evaluations (up to constants). It follows from the bound of Liu and Zhandry [LZ19] for $C = 3$ that at least $\Omega\left(2^{3n/7}\right)$ quantum oracle queries are necessary for a random compression function. This shows that our attack is essentially optimal.

1.4 Related Work

Several other works have further refined the herding attack in the classical setting. Andreeva and Mennink [AM11] generalized the chosen-target forced-prefix attack and discuss the case that the part P may appear in the middle of the adversary's final output, $S_1 \| P \| S_2$. This covers for example attacks on zipper hash functions and similar constructs [ABDK09]. Kortelainen and Kortelainen [KK13] show how to remove the extra factor $\sqrt{k}$ when building the diamond structure, using a sophisticated construction. Weizman et al. [WDH17] subsequently improve the constant in the $\mathcal{O}$-notation for generating the diamond. We do not pursue such generalizations here.

Aiming at general improvements of quantum attacks for arbitrary hash functions, Chailloux et al. [CNS17] discuss different memory-time trade-offs for finding collisions and a preimage in a list of values. Using variations of Grover's algorithm they show collision attacks with $\widetilde{\mathcal{O}}\left(2^{2n/5}\right)$ evaluations, and multi-target preimage attacks with $\widetilde{\mathcal{O}}\left(2^{3n/7}\right)$ evaluations. Both algorithms only use $\mathcal{O}(n)$ quantum memory, and can be parallelized. Parallelization of multi-target preimage search, also in realistic communication models, has also been considered in [BB17]. They show that, with realistic communication models, they can find a preimage in a list of t values with p processors with $\mathcal{O}(\sqrt{2^n/pt^{1/2}})$ evaluation steps (where the bound improves from $t^{1/2}$ to t for models with free communication). We do not aim to optimize memory usage for our algorithms, which already store the diamond structure, but focus on the number of hash evaluations here.

Dedicated quantum attacks against specific hash function, improving over the generic bounds, have gained more attention recently. The work of Hosoyamada and Sasaki [HS20] discusses collision-finding attacks against AES-MMO and Whirlpool. Refined collision and preimage attacks on AES-like hash functions have been presented subsequently by Dong et al. [DSS+20,DZS+21], Florez Gutierrez et al. [FGLN+20], as well as Ni et al. [NDJY21]. Hosoyamada and Sasaki [HS21] devised dedicated quantum collision attacks against reduced versions of SHA-256 and SHA-512. Wang et al. [WLG+22] present preimage attacks on 4-round versions of Keccak. While neither of these works considers the Nostradamus attack, the results and methods may be also useful to devise improved Nostradamus attacks against specific hash functions. In this work here, however, we are interested in the complexity of generic attacks.

2 Preliminaries

2.1 Hash Functions

Analogously to [KK06] we consider hash functions $\mathsf{H} : \{0,1\}^* \to \{0,1\}^n$ based on the Merkle-Damgård construction with a compression function $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$, where B is the size of the message blocks and n the size of the final hash value and the intermediate hash states. We implicitly assume a (public) initialization vector IV, and for some input $m = m_1 m_2 \dots m_\ell$, $m_i \in \{0,1\}^B$, aligned to block length B, we define the iterated compression function as

$$h^*(m) := y_\ell, \text{ where } y_0 = IV, y_i = h(m_i, y_{i-1}) \text{ for } i = 1, 2 \dots, \ell.$$

For non-aligned inputs m we assume that the hash function uses a form of suffix-padding function pad which only depends on the input length, such that $m \| \mathsf{pad}(|m|) \in (\{0,1\}^B)^*$. For example, for Merkle-Damgård hash functions like SHA2 one appends $10^d \| \langle |m| \rangle$ for a sufficient number d of 0-bits, where $\langle i \rangle$ is a fixed-length binary encoding of the integer i. We assume that the padding extends the message by at most one block.

We assume additionally that the compression function for any fixed intermediate value $y \in \{0,1\}^n$, given as $h_y(\cdot) := h(\cdot, y)$, is surjective and sufficiently close to regular. More specifically, we assume that this function is *β-balanced*, i.e., $|h_y^{-1}(h_y(m))| \geq \beta \cdot 2^{B-n}$ holds for all $m \in \{0,1\}^B$. We note that collisions would be easier to find for the compression function if the preimage sets are significantly skewed [BK04]. Indeed, Bellare and Kohno [BK04] also define a more fine grained balance notion for hash functions resp. compression functions. If $h_y(\cdot)$ is β-balanced according to our notion, then it has a balance factor of at least $1 - 2\log_{2^n} \beta$ according to their notion. The simpler balance notion here suffices to give precise performance guarantees when using Grover's quantum algorithm:

Definition 1. *A compression function $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$ for $B \geq n$ of an iterated hash function H is called β-balanced (where $\beta \in (0,1]$) if for any $y \in \{0,1\}^n$ and any $z \in \{0,1\}^n$ the number of preimages satisfies $|h_y^{-1}(z)| \geq \beta \cdot 2^{B-n}$ for the function $h_y(\cdot) = h(\cdot, y)$.*

We observe that, by definition, $\beta > 0$. This means that a β-balanced compression function satisfies $|h_y^{-1}(z)| \geq \beta \cdot 2^{B-n} > 0$ for any y, z. In other words, any image z has a preimage under $h_y(\cdot)$. This in particular means that the function $h_y(\cdot)$ is surjective for any y. Let us stress that we only need the β-balance property for the formal analysis. Our attacks may still succeed if the function is less balanced. This also complies with our implementation results where our example hash function is not even surjective.

We briefly discuss that, if we assume h to be random, then it is β-balanced for $\beta = \frac{1}{2}$ with overwhelming probability for $B \gg n$. For this note that for any fixed image z, the probability that z has less than $\beta \cdot 2^{B-n}$ of the expected number 2^{B-n} of preimages, is at most $\exp(-2^{B-n}/8)$ by the Chernoff bound,

and thus double exponentially small. Hence, the probability that there exists any z among the 2^n images violating the bound is still double exponentially small. This means that for a random h we can almost surely assume that each value $z \in \{0,1\}^n$ is hit by at least $\beta \cdot 2^{B-n}$ preimages.

2.2 Quantum Collision Finding

In the following chapter we will see that the Nostradamus attack is based on finding collisions of the compression function h. Therefore we will introduce a generalization of Grover's algorithm as $\mathsf{Grover}(F, y_0)$ and the specific quantum model, which is required for the efficient collision finding algorithm from [BHT98]:

Theorem 2 (Grover, [BBHT98])**.** *Let $F : X \to Y$ be a function, $N := |X|$ be the cardinality of the domain, $y_0 \in Y$ be fixed and*

$$t = |F^{-1}(y_0)| = |\{x \in X | F(x) = y_0\}|$$

be the cardinality of the preimage of y_0 under F. If $t \geq 1$, then the algorithm $\mathsf{Grover}(F, y_0)$ outputs an $x \in X$ with $F(x) = y_0$ after $\mathcal{O}\left(\sqrt{N/t}\right)$ expected evaluations of F.

We consider an adversary with access to a (local) quantum computer and assume that the compression function h is quantum accessible, i.e., the adversary can implement h efficiently on its quantum computer. This allows the adversary to query this h-oracle with arbitrary superposition of the inputs, akin to the quantum random oracle model [BDF+11]. This enables us to also use the algorithm $\mathsf{Grover}(F, y_0)$ in cases where the function F depends on the compression function h.

Note that Boyer et al. [BBHT98] discuss that the above theorem even holds if the number t of solutions is not known in advance. In our attacks against hash functions we will later take advantage of the fact the compression function h is β-balanced (where β is usually assumed to be constant). Hence, using Grover's algorithm for searching a preimage for some value z among the $N = 2^B$ many inputs to h, of which at least $t \geq \beta \cdot 2^{B-n}$ solutions map to z, takes an expected number $\mathcal{O}\left(\sqrt{2^B/\beta 2^{B-n}}\right) = \mathcal{O}\left(\beta^{-1/2} \cdot 2^{n/2}\right)$ of function evaluations. For constant β this equals $\mathcal{O}\left(2^{n/2}\right)$. We also point out that in most of our attacks we apply Grover's algorithm multiple times, in sequential order, such that the expected number of total evaluations of F is given by the sum of the expected evaluations of the individual calls, by the linearity of expectations.

3 Classical Nostradamus Attack

In this section we review the idea of Kelsey and Kohno [KK06] for the Nostradamus attack.

Attack. The Nostradamus attack lets the adversary first commit to a target hash value y_{trgt}. Then it receives a prefix P where we assume for simplicity that P is aligned to block length and that the number ℓ_P of blocks of P is known in advance. If this was not the case the adversary could simply guess the actual length of P and append 0's if necessary. The task of the adversary is now to find a suffix S such that $\mathsf{H}(P\|S) = y_{\text{trgt}}$.

We have to specify how the prefix P is chosen in the attack. Kelsey and Kohno [KK06] assume that the prefix is chosen randomly from a specified set. To avoid assumptions about the distribution of P we ask the adversary to succeed for any given value P. The latter matches the fact that known attacks indeed achieve this. We specify this by quantifying over all possible prefixes P, demanding the adversary to win in all cases. We also require the adversary to succeed with probability 1 (over the internal randomness). Since we allow the adversary to run in expected time, we can always enforce this by iterating till success, compensating for smaller success probabilities by larger run times:

Definition 3 (Successful Nostradamus Attack)**.** *A successful Nostradamus attack $\mathcal{A}$ for an iterated hash function H based on compression function $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$ consists of two algorithms $(\mathcal{A}^{off}, \mathcal{A}^{onl})$ such that, for any $\ell_P \in \mathbb{N}$, any $P \in \{0,1\}^{\ell_P \cdot B}$, the following experiment $\boldsymbol{Exp}^{Nostr}_{\mathsf{H},\mathcal{A},P}(\lambda)$ always returns 1:*

$\boldsymbol{Exp}^{Nostr}_{\mathsf{H},\mathcal{A},P}(\lambda)$*:*

$(st, y_{trgt}) \leftarrow\!\!\$\ \mathcal{A}^{off}(1^\lambda, \ell_P)$ // *offline*

$S \leftarrow \mathcal{A}^{onl}(st, P)$ // *online*

return $[\mathsf{H}(P\|S) = y_{trgt}]$

Note that we do not make any stipulations on the run time of adversary $\mathcal{A}$. In this sense there is always the trivial attack which executes an exhaustive search. We are interested in more efficient attacks, of course. The parameter ℓ_P usually enters the run time of the adversary, but only mildly compared to the search for the initial state and for the suffix.

Offline Phase. The Nostradamus attack of Kelsey and Kohno [KK06] consists of an offline and an online phase. In the offline phase the adversary creates a (binary) tree structure (V, E), also referred to as a diamond structure. Each node $v \in V \subseteq \{0,1\}^n$ in the tree represents a hash state and each (directed and labeled) edge $e = (y_0, m, y_1) \in E \subseteq V \times \{0,1\}^B \times V$ represents a transition from one hash state to the next one via the message label, $h(m, y_0) = y_1$.[3]

The tree consists of 2^k *distinct* leaves where k is chosen appropriately. The algorithm of Kelsey and Kohno starts by sampling the leaf nodes, building up the tree level by level, by trying message blocks for each node in order to find

[3] Note that we simply identify a node with its hash value label. Formally, to make nodes with identical hash values distinct, we add a position in the tree to each node (given by the level and its order within the nodes of the same level) but usually omit mentioning the position value.

collisions. A level of the tree is a set of nodes with identical (edge) distance to the root node. These levels and its nodes get numbered by the distance, e.g., level k contains the leaf nodes $y_{k,1}, y_{k,2}, \ldots$ and level 0 the root node $y_{0,1}$. In general, node $y_{i,j}$ is the j^{th} node of level i. See Figure 2.

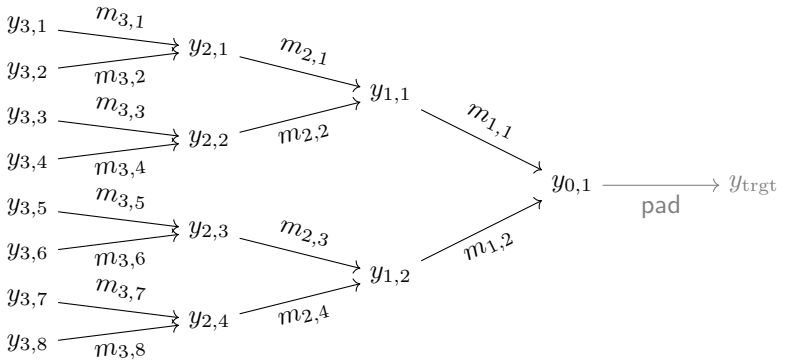

Fig. 2: Tree structure of height 3 in offline phase of classical Nostradamus attack, where $y_{\text{trgt}} = h(\mathsf{pad}(B(\ell_P + 4)), y_{0,1})$.

The original analysis in [KK06] argued that roughly $2^{\frac{n+k}{2}}$ compression function evaluations are necessary to form such a tree of height k. This, however, ignored that one node value $y_{i,j}$ may collide with multiple other values, such that we would not get a full binary tree. Luckily, in [BSU12] it is shown that we can capture this by an extra factor $\sqrt{k}$. That is, in [BSU12] it is proven that $\sqrt{k \cdot \ln 2} \cdot 2^{\frac{n-k}{2}}$ evaluations of the function h suffice per node, and $\Theta(\sqrt{k} \cdot 2^{\frac{n+k}{2}})$ in total for building the complete diamond structure. Afterwards the hash value $y_{\text{trgt}} = h(\mathsf{pad}(B \cdot (\ell_P + k + 1)), y_{0,1})$ can be computed and submitted, where $y_{0,1}$ is the root of the tree structure.

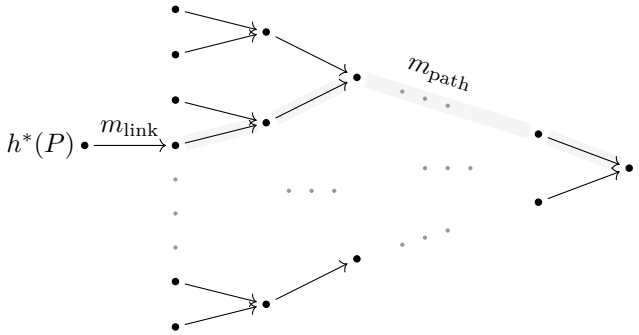

Fig. 3: Linking message m_{link} for given prefix P.

Online Phase. In the online phase the adversary uses the tree structure to construct the suffix S. For this the adversary searches for a linking message block

$m_{\text{link}} \in \{0,1\}^B$ such that $h(m_{\text{link}}, h^*(P))$ hashes to a leaf in the tree (see Figure 3). If such a message block m_{link} has been found, then the suffix S is given by $S := m_{\text{link}} \| m_{\text{path}}$, where m_{path} denotes the (concatenation of the) message labels on the path from the leaf to the root of the tree. See Figure 3. Note that by construction $P\|S$ consists of ℓ_P blocks in P, one linking block m_{link}, and the k blocks in m_{path}, such that the message is of bit length $B \cdot (\ell_P + k + 1)$. Together with the appended padding the hash function thus maps $P\|S$ to the pre-selected target value y_{trgt}.

Since there are 2^k randomly and independently sampled leaves in the tree one can expect to find such a linking message m_{link} with 2^{n-k} evaluations of h. Moreover [BSU12] proves that the height of the tree can be optimally chosen with $k \approx \frac{n}{3}$, so that the entire attack needs $\mathcal{O}(\sqrt{n} \cdot 2^{\frac{2n}{3}})$ evaluations of h.

4 Quantum-based Nostradamus Attacks

We start by giving a straightforward quantum version of the Nostradamus attack, exploiting that one can find collisions for hash functions in about $2^{n/2}$ quantum steps. This already beats the classical bound of roughly $2^{2n/3}$ but we show afterwards that one can go lower with more advanced strategies. The first advanced attack follows the same structure as the classical herding attack but reduces the work load to approximately $2^{4n/9}$ by using Grover's algorithm. The second advanced algorithm then optimizes the step to build the diamond structure, resulting in a total number of roughly $2^{3n/7}$ compression function evaluations.

4.1 Simple Quantum Attack

We first describe a very simple quantum Nostradamus attack $\mathcal{A}_{\text{simple}}$. The algorithm receives as input the block length ℓ_P of the unknown prefix, picks a random value y (to which $P\|S$ will map under h^*) and applies the final iteration for the padding. In the online phase it simply runs Grover's algorithm to find a linking message block m_{link}:

$\mathcal{A}^{\text{off}}_{\text{simple}}(1^\lambda, \ell_P)$: // offline

1 : $y \leftarrow_{\$} \{0,1\}^n$
2 : $y_{\text{trgt}} \leftarrow h(\mathsf{pad}(B \cdot (\ell_P + 1)), y)$
3 : **return** (y, y_{trgt})

$\mathcal{A}^{\text{onl}}_{\text{simple}}(y, P)$: // online

1 : $p \leftarrow h^*(P)$
2 : $S \leftarrow \mathsf{Grover}(h_p, y) /\!/ h_p(\cdot) = h(\cdot, p)$
3 : **return** S

Proposition 4. *Let* H *be a hash function with a* β*-balanced compression function* $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$*. Algorithm* $\mathcal{A}_{simple}$ *mounts a successful Nostradamus attack against* h *with* $\mathcal{O}\left(\beta^{-1/2} \cdot 2^{n/2} + \ell_P + 1\right)$ *expected evaluations of* h *for any prefix* P *of block length* ℓ_P*.*

Note that for constant β (e.g., recall that $\beta = \frac{1}{2}$ works with overwhelming probability for random function h) we obtain approximately the bound $\mathcal{O}\left(2^{n/2}\right)$.

Proof. Correctness follows directly by construction and the fact that the compression function h is surjective according to the balance property: For the chosen value y there exists at least one preimage S such that

$$y = h_p(S) = h(S, p) = h^*(P\|S),$$

and $y_{\text{trgt}} = h(\mathsf{pad}(|P\|S|), y)$ by construction. As for performance, Theorem 2 implies that the search for S needs $\mathcal{O}\left(\sqrt{N/t}\right)$ evaluations in expectation where

$$N = |\{0,1\}^B| = 2^B \quad \text{and} \quad t = |h_p^{-1}(y)| = |\{m \in \{0,1\}^B | h_p(m) = y\}|.$$

Since the function h_p is β-balanced it follows that $t \geq \beta \cdot 2^{B-n}$, yielding an overall effort of $\mathcal{O}\left(\beta^{-\frac{1}{2}} \cdot 2^{n/2}\right)$ evaluations for Grover's search. The computation of y_{trgt} in the offline phase and the initial computation of p in the online phase need at most $\ell_P + 1$ additional evaluations of h. □

4.2 Basic Quantum Attack with Diamond Structure

This section shows that the construction of a diamond structure is rewarding and leads to a more efficient quantum attack. We first describe a basic version of this attack and optimize it in the next section. We present the algorithm by dividing into several sub algorithms, following the structure of the classical attack: In the offline phase we use algorithm $\mathsf{Diamond}_{\text{basic}}$ to build the diamond structure. The algorithm itself uses a collision finder Claw as a subroutine. In the online phase we once more use the Link algorithm to find the linking message m_{link}. Finally we put all algorithms together to derive our adversary.

We first describe the claw finding algorithm Claw, which is in fact the well-known BHT algorithm [BHT98] adapted to our setting. The algorithm takes as input a parameter ℓ and two functions h_y and $h_{y'}$ and returns m, m' such that $h_y(m) = h_{y'}(m')$. It does so by first sampling a random list of 2^ℓ input messages m_i for h_y, and uses Grover's algorithm to find the matching value m' for $h_{y'}$.

$\mathsf{Claw}_\ell(h_y, h_{y'})$:

1 : $m_1, \ldots, m_{2^\ell} \leftarrow\!\!\$\, \{0,1\}^B$ such that $h_y(m_i) \neq h_y(m_j)$ for all $i \neq j$
2 : $m' \leftarrow \mathsf{Grover}(F, 1)$ // F as in Proposition 5
3 : $i \leftarrow \{1, \ldots, 2^\ell\}$ such that $h_y(m_i) = h_{y'}(m')$
4 : **return** (m_i, m')

Note that for a well-balanced compression function and for $\ell \leq n/3$, as we need below, the message sampling in the first step can be implemented by picking all the m_i's randomly, at once. A hash collision among the at most $2^\ell \leq 2^{n/3}$ hash values happens with sufficiently small probability, and in the rare case of a collision we repeat the process. On average we do not need to make more than two iterations for avoiding collisions. In the theorem below,

and also in our implementation, we nonetheless consider the general case of a β-balanced compression function, in which case we re-sample each m_i individually if it collides with a previous choice.

Proposition 5 ([BHT98]). *Let H be a hash function with β-balanced compression function $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$. For $\ell \in \mathbb{N}$, a set $m_1, \ldots, m_{2^\ell} \in \{0,1\}^B$ of distinct messages, and values $y, y' \in \{0,1\}^n$. Let $F : \{0,1\}^B \to \{0,1\}$ be the function defined by*

$$F(m) = 1 \quad :\Longleftrightarrow \quad \exists i \in \{1, \ldots, 2^\ell\} : h_y(m_i) = h_{y'}(m).$$

Then algorithm $\mathsf{Claw}_\ell(h_y, h_{y'})$ outputs messages $m, m' \in \{0,1\}^B$ with $h_y(m) = h_{y'}(m')$ and needs $\mathcal{O}\left(\beta^{-1} \cdot 2^\ell + \beta^{-\frac{1}{2}} \cdot 2^{\frac{n-\ell}{2}}\right)$ expected evaluations of h for $\ell < n$. In particular, $\mathsf{Claw}_{n/3}$ needs $\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}}\right)$ expected evaluations of h.

Note that for $\beta \leq 1$ we always have $\beta^{-1/2} = \mathcal{O}\left(\beta^{-1}\right)$ and from now on bound the factor $\beta^{-1/2}$ by the inverse of β.

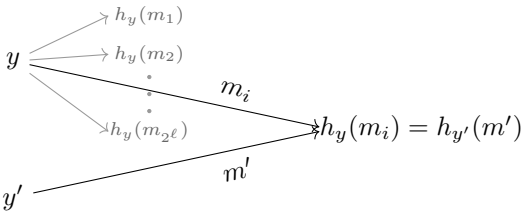

Fig. 4: Illustration of Algorithm $\mathsf{Claw}_\ell(h_y, h_{y'})$.

Proof. Let us first discuss how we generate the 2^ℓ message blocks m_i in the first step for general compression functions. For this we iterate through $i = 1, 2, 3, \ldots$ and for each i we repeatedly pick $m_i \leftarrow\!\!\$\, \{0,1\}^B$ randomly, until $h_y(m_i)$ is different from all the previous hash values. For the i-th step there are $2^n - (i-1)$ hash values unoccupied, and each image has at least $\beta \cdot 2^{B-n}$ preimages. Hence, we pick such a good preimage with probability at least

$$2^{-B} \cdot (2^n - i + 1) \cdot \beta \cdot 2^{B-n} \geq (2^n - 2^\ell) \cdot \beta \cdot 2^{-n} = (1 - 2^{\ell-n}) \cdot \beta \geq \tfrac{1}{2} \cdot \beta$$

for $\ell < n$. On average, we thus only need to sample $\mathcal{O}\left(\beta^{-1}\right)$ times for each of the 2^ℓ message blocks.

The proof now follows straightforwardly from Theorem 2, noting that the input size of function F equals $N = 2^B$, and since all 2^ℓ hash values are distinct and each hash value has at least $\beta \cdot 2^{B-n}$ preimages by the balance property of h, there are at least $t \geq \beta \cdot 2^\ell \cdot 2^{B-n}$ possible solutions. □

Given the claw finding algorithm we present our basic algorithm for deriving the diamond structure. The algorithm takes as input a parameter k determining

the height of the tree (V, E) which it outputs. The algorithm uses $\mathsf{Claw}_{n/3}$ to determine collisions for neighbored values:

$\mathsf{Diamond}_{\mathrm{basic}}(k)$:

1 : $(V, E) \leftarrow \emptyset$
2 : $y_{k,1}, y_{k,2}, \ldots, y_{k,2^k} \leftarrow\!\!\$\ \{0,1\}^n$ pairwise different // leaf nodes
3 : **for** $s = k, \ldots, 1$ **do** // for constructing level $s-1$
4 : $\quad V \leftarrow V \cup \{y_{s,1}, \ldots, y_{s,2^s}\}$
5 : $\quad$ **for** $i = 1, \ldots, 2^{s-1}$ **do**
6 : $\quad\quad (m, m') \leftarrow \mathsf{Claw}_{n/3}(h_{y_{s,2i-1}}, h_{y_{s,2i}})$
7 : $\quad\quad E \leftarrow E \cup \{(y_{s,2i-1}, m, h_{y_{s,2i-1}}(m)), (y_{s,2i}, m', h_{y_{s,2i-1}}(m))\}$
8 : $\quad\quad y_{s-1,i} \leftarrow h_{y_{s,2i-1}}(m)$ // for next iteration
9 : $\quad$ **endfor**
10 : **endfor**
11 : **return** (V, E)

Proposition 6. *Let* H *be a hash function with* β*-balanced compression function* $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$*. Algorithm* $\mathsf{Diamond}_{basic}(k)$ *outputs a diamond structure of height* $k \leq n$ *after* $\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}+k}\right)$ *expected evaluations of* h*.*

Once more, for constant β the bound simplifies to $\mathcal{O}\left(2^{\frac{n}{3}+k}\right)$. We note that the sampling of the 2^k values $y_{k,i}$'s in Line 2 must yield pairwise distinct strings. Since we later choose $k = n/9$ a collision among the random values happens with negligible probability only. Alternatively, we may either pick arbitrary distinct values, e.g., by incrementing a counter value, or sample each $y_{k,i}$ as often till we have a fresh value. In our implementation we use the latter. We remark that this sampling step in either case does not account for the number of hash evaluations.

Proof. The output graph is indeed a diamond structure of height k, because on the one hand the connecting rule is fulfilled since $h_{y_{s,2i-1}}(m) = h_{y_{s,2i}}(m')$ for any s, i and the edges are directed from higher to lower levels since the algorithm starts with level k. On the other hand the graph has an underlying binary tree structure with 2^k leaf nodes. The algorithm constructs, for a fixed value s, the next level $s-1$ by connecting the nodes of level s in a pairwise manner. Therefore the next level contains 2^{s-1} nodes. Within the last iteration, where $s = 1$, a level with $2^{1-1} = 1$ nodes is constructed, which is the root node of the resulting graph.

As for performance, we mainly need to consider the repetitions of Line 6, the executions of algorithm Claw. Individually each search on average needs $\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}}\right)$ evaluations of h according to Proposition 5. Thus, for building the entire structure,

$$\sum_{s=1}^{k} \sum_{i=1}^{2^{s-1}} \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}}\right) = \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}} \cdot \sum_{s=1}^{k} 2^{s-1}\right)$$
$$= \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}} \cdot (2^k - 1)\right) = \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}+k}\right)$$

expected evaluations are required in total. □

We next describe our algorithm Link finding the linking message m_{link}. The algorithm takes as input the 2^k leaves of the diamond structure and the prefix P, and finds a message block m_{link} that connects P to one of the leaves. This is done via Grover's algorithm by defining a suitable function F identifying such links:

$\mathsf{Link}(P, y_{k,1}, \ldots, y_{k,2^k})$:

1 : $p \leftarrow h^*(P)$
2 : $m_{\text{link}} \leftarrow \mathsf{Grover}(F, 1)$ // F as in Proposition 7
3 : let $y_{k,i}$ be a leaf with $h(m_{\text{link}}, p) = y_{k,i}$
4 : **return** $(m_{\text{link}}, y_{k,i})$

Proposition 7. *Let* H *be a hash function with* β*-balanced compression function* $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$*. Let* $P \in \{0,1\}^{\ell_P}$*,* $p \leftarrow h^*(P)$ *and* (V, E) *be a diamond structure of height* k *with (distinct) leaves* $(y_{k,1}, \ldots, y_{k,2^k})$*. Define the function* $F : \{0,1\}^B \to \{0,1\}$ *as*

$$F(m) = 1 \iff \exists i \in \{1, \ldots, 2^k\} : h(m, p) = y_{k,i}.$$

Then algorithm $\mathsf{Link}(k, y_{k,1}, \ldots, y_{k,2^k})$ *outputs a linking message block* m_{link} *and the connecting leaf node* $y_{k,i}$ *with* $h(m_{link}, p) = y_{k,i}$*, requiring* $\mathcal{O}\left(\beta^{-\frac{1}{2}} \cdot 2^{\frac{n-k}{2}} + \ell_P\right)$ *expected evaluations of* h *in total.*

Proof. The correctness follows directly by the construction of the function F. As for performance, Theorem 2 implies that the search for m_{link} needs $\mathcal{O}\left(\sqrt{N/t}\right)$ evaluations where

$$N = 2^B \quad \text{and} \quad t = |\bigcup_{i=1}^{2^k} h_p^{-1}(y_{k,i})| \geq \beta \cdot 2^{k+B-n},$$

since the function h_p is β-balanced, and the leaves are distinct in a diamond structure. The first step to compute p needs at most ℓ_P evaluations of h. □

At this point, $\mathsf{Diamond}_{\text{basic}}$ and Link can be composed to form our basic quantum attack $\mathcal{A}_{\text{basic}}$ (for parameter k), using a diamond structure of height k. We presume that $\mathsf{Path}((V, E), y)$ is the algorithm, which concatenates the message labels on the edges from leaf y to the root of the tree:

$\mathcal{A}^{k,\text{off}}_{\text{basic}}(1^\lambda, \ell_P)$:

1 : $(V, E) \leftarrow\!\!\$\ \mathsf{Diamond}_{\text{basic}}(k)$
2 : let $y_{0,1} \in V$ be the root of (V, E)
3 : $y_{\text{trgt}} \leftarrow h(\mathsf{pad}(B \cdot (\ell_P + k + 1)), y_{0,1})$
4 : **return** $((V, E), y_{\text{trgt}})$

$\mathcal{A}^{k,\text{onl}}_{\text{basic}}((V, E), P)$:

1 : leaves $y_{k,1}, \ldots, y_{k,2^k}$ of (V, E)
2 : $(m_{\text{link}}, y) \leftarrow \mathsf{Link}(P, y_{k,1}, \ldots, y_{k,2^k})$
3 : $m_{\text{path}} \leftarrow \mathsf{Path}((V, E), y)$
4 : $S \leftarrow m_{\text{link}} \| m_{\text{path}}$
5 : **return** S

Theorem 8 (Basic Quantum Attack with Diamond Structure)**.** *Let* H *be a hash function with* β*-balanced compression function* $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$*. Let* $k \in \mathbb{N}$*. Then adversary* $\mathcal{A}^k_{basic}$ *mounts a successful Nostradamus attack and needs*

$$\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n}{3}+k} + \beta^{-\frac{1}{2}} \cdot 2^{\frac{n-k}{2}} + \ell_P + 1\right)$$

expected evaluations of h*. In particular, for* $k = \frac{n}{9}$ *the adversary* $\mathcal{A}^{n/9}_{basic}$ *in total needs* $\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{4n}{9}} + \ell_P + 1\right)$ *expected evaluations of* h*.*

4.3 Attack with Enhanced Diamond Structure Generation

We next present an advanced attack, essentially reaching the lower bound of $\Omega(2^{3n/7})$ for herding attacks discussed in Section 5.2. The algorithm still applies the same general strategy as in the previous section, but uses an enhanced algorithm to create the diamond structure. The idea there was basically to connect two nodes in the tree via algorithm Claw for parameter $n/3$, resulting in $2^{n/3}$ compression function evaluations for each connection. We now discuss how we can speed up this process by re-using data across the various connection steps.

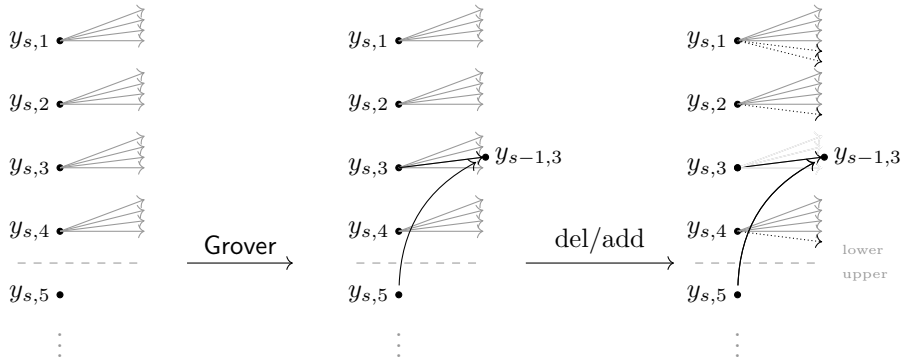

Fig. 5: Iteration of algorithm $\mathsf{Diamond}_{\text{enhanced}}$. Left: At the beginning of each iteration the nodes in the lower half have 2^ℓ potential successors, roughly distributed equally over all remaining nodes (here 16 successors, assigned evenly to the 4 nodes $y_{s,1}, \ldots, y_{s,4}$). The next node from the upper half (here $y_{s,5}$) shall be connected to a node in the lower part. Middle: Grover's algorithm finds a match to one of the 2^ℓ successors (called $y_{s-1,3}$ here) of a node from the lower part (here $y_{s,3}$) such that we can connect the two nodes. Right: We remove all potential successor pointers from the connected node from the lower half and add, step by step, the same number of new pointers to the remaining nodes (here displayed as dotted arrows).

Consider a level s of the tree for which we try to connect the 2^s nodes $y_{s,1}, \ldots, y_{s,2^s}$ in a pairwise manner. We split the 2^s nodes into a lower and an upper half of 2^{s-1} nodes each. For the lower half we will compute a list Y of 2^ℓ hash evaluations $h(m_j, y_{s,i})$, equally spread out over the 2^{s-1} values. Here ℓ will be an appropriate parameter to be determined later. Then we use Grover's

search to connect the first value $y_{s,2^{s-1}+1}$ of the upper half to some of these 2^ℓ values via some message block m'. Once we have found such a connection we are going to remove the partner node from the lower half and all of its $2^\ell/2^{s-1}$ entries in Y. We add this amount of new values, again equally spread out over the remaining $2^{s-1}-1$ values paired up, to fill the list Y up to 2^ℓ elements again. This idea is displayed in Figure 5. Then we are going to connect the second node from the upper half to an entry in the updated list Y, as before. We continue till we have all 2^s values and then proceed with the next level $s-1$ till we have eventually built the entire tree.

We present the enhanced algorithm for creating the diamond structure next. The algorithm takes as input the parameter k (for the tree height) and outputs the tree (V, E). It internally uses the parameter ℓ (for the list size) in dependence of the tree level s it currently considers. The algorithm uses Grover's algorithm as a subroutine for a function $F_{y,Y}(m)$ with parameters y and Y, which checks if $h_y(m)$ is in the list Y. Put differently, Grover's algorithm returns such a matching message block m.

$\mathsf{Diamond}_{\text{enhanced}}(k)$:

1 : $(V,E) \leftarrow \emptyset$
2 : $y_{k,1},\ y_{k,2},\ \ldots, y_{k,2^k} \leftarrow\!\!\$\ \{0,1\}^n$ pairwise different // leaf nodes
3 : **for** $s = k, \ldots, 1$ **do** // for constructing level $s-1$
4 : $\quad \ell \leftarrow \lceil (n + 2s - 2\log_2\ s)/3 \rceil$
5 : $\quad V \leftarrow V \cup \{y_{s,1}, \ldots, y_{s,2^s}\}$
6 : $\quad L \leftarrow \{1, \ldots, 2^{s-1}\}$ // list of unprocessed nodes in lower half
7 : $\quad Y \leftarrow \emptyset$ // storing precomputed values
8 : $\quad \gamma \leftarrow 1$ // state counter where to add next elements to Y
9 : $\quad$ **foreach** $y \in \{y_{s,2^{s-1}+1}, \ldots, y_{s,2^s}\}$ **do** // process each node in upper half
10 : $\quad\quad$ **while** $|Y| < 2^\ell$ **do** // Add elements to list of precomputed values?
11 : $\quad\quad\quad$ **while** $\gamma \notin L$ **do** $\gamma \leftarrow (\gamma \bmod 2^{s-1}) + 1$ // choose (circularly) next node
12 : $\quad\quad\quad m \leftarrow\!\!\$\ \{0,1\}^B$ such that $(*, *, h(m, y_{s,\gamma})) \notin Y$ // unique in Y
13 : $\quad\quad\quad Y \leftarrow Y \cup \{(m,\ \gamma,\ h(m, y_{s,\gamma})\}$
14 : $\quad\quad\quad \gamma \leftarrow \gamma + 1$
15 : $\quad\quad$ **endwhile** $\quad m \leftarrow \mathsf{Grover}(F_{y,Y}, 1)$ // F as in Theorem 9
16 : $\quad\quad$ search for $(m', i, y') \in Y$ with $y' = h_y(m)$ // unique due to Line 12
17 : $\quad\quad E \leftarrow E \cup \{(y_{s,i},\ m',\ y'), (y,\ m,\ y')\}$ // connect nodes
18 : $\quad\quad y_{s-1,i} \leftarrow y'$ // for next iteration, noting that $i \le 2^{s-1}$
19 : $\quad\quad$ delete any $(*, i, *)$ from Y // only values for other nodes remain in Y
20 : $\quad\quad$ delete i from L // $y_{s,i}$ has been processed
21 : $\quad$ **endforeach**
22 : **endfor**
23 : **return** (V, E)

Note that we need to find a message m in Line 12 for which the hash value has not been assigned yet. Recall that ℓ is in the order of $(n+2s)/3 \leq (n+2k)/3$. We will later set $k = n/7$ such that $\ell \leq 3n/7$. Hence, assuming that the hash value of m is uniformly distributed, the probability of having collisions is at most $2^{2\ell} \cdot 2^{-n} \leq 2^{-n/7}$ and thus negligible. Below, for general β-balanced compression function, we follow once more the idea to sample-till-success to find m, as also done in the algorithm Claw.

Theorem 9. *Let* H *be a hash function with* β*-balanced compression function* $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$*. For a list* Y *and a fixed value* y *let the function* $F : \{0,1\}^n \to \{0,1\}$ *be defined as*

$$F(m) = 1 \quad :\Longleftrightarrow \quad \exists y' \in Y : h_y(m) = y'.$$

Then algorithm $\mathsf{Diamond}_{enhanced}(k)$ *outputs a diamond structure* (V, E) *of height* k *with*

$$\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+2k+\log_2(k)}{3}}\right)$$

expected evaluations of h *for* $k < n$*.*

Proof. Correctness follows by construction. Starting with the random leaves the algorithm connects one node from the lower part with one node from the upper part in each iteration, yielding a binary tree. The exact matching of values in the lower and upper half also implies that the search for the next γ-value always terminates (because L has at least one value if there is some y in the upper half left).

We next look at the performance of the algorithm. Each application of the Grover algorithm (in Line 15) needs $\mathcal{O}\left(\sqrt{N/t}\right)$ evaluations of h, where

$$N = |\{0,1\}^B| = 2^B, \qquad t \geq \beta \cdot 2^\ell \cdot 2^{B-n}$$

by construction, since the set Y contains exactly 2^ℓ pairwise different elements from $\{0,1\}^n$ in each iteration and the function h is β-balanced. This means that the algorithm requires $\beta^{-\frac{1}{2}} \cdot 2^{\frac{n-\ell}{2}}$ evaluations of the compression function on the average. Note that we run Grover on level s for 2^s times, yielding an overall number of $\beta^{-\frac{1}{2}} \cdot 2^{\frac{n-\ell+2s}{2}}$ expected evaluations of h for this level.

Next we consider the number of hash evaluations for filling up the list Y (**while**-loop in Line 10) when iterating through all values y in the upper half (**foreach**-loop in Line 9). For this we start by looking at the search for a message block with a fresh hash value in Line 12. As in the Claw algorithm we can sample a message block $m \leftarrow_\$ \{0,1\}^B$ till we found one whose hash value is not in Y. Since we have at most $2^\ell - 1$ many values in Y at this point, where $\ell \leq (n+2s)/3 \leq (n+2k)/3 < n$ for $k < n$, we can conclude as in the analysis of Proposition 5 that each search requires $\mathcal{O}\left(\beta^{-1}\right)$ attempts on the average.

For the overall analysis of the loops we first discuss a simplified version, neglecting rounding of fractions. Since we start with Y being empty for the first value y, we need 2^ℓ many values to fill the list in the first iteration. Then, at the

end of the iteration, we remove at most $2^\ell/2^{s-1}$ entries in Y for the identified element $y_{s,i}$ in the lower part. This implies that in the next step we need to make $2^\ell/2^{s-1}$ hash evaluations to fill the list Y again up to 2^ℓ elements. In the next step, however, we have only $2^{s-1}-1$ elements left, and thus remove at most $2^\ell/(2^{s-1}-1)$ elements for the subsequent iteration. This continues for $2^{s-1}-2$, $2^{s-1}-3$, ..., until all the 2^{s-1} iterations for values in the upper half are completed. For the final step we need to add $2^\ell/(2^{s-1}-(2^{s-1}-2)) = 2^\ell/2$ elements.

We can write the total number of elements to be sampled for the resurrection of the complete list Y thus as:

$$2^\ell + 2^\ell \cdot \sum_{j=0}^{2^{s-1}-2} \frac{1}{2^{s-1}-j} = 2^\ell \cdot \sum_{j=1}^{2^{s-1}} \frac{1}{j} = \mathcal{O}\left(2^\ell \cdot \ln 2^s\right) = \mathcal{O}\left(s \cdot 2^\ell\right),$$

using $\sum_{i=1}^{q} \frac{1}{j} \leq \ln q + c$ for the harmonic series. On average, we need $\mathcal{O}\left(\beta^{-1}\right)$ h-evaluations for each element.

Note that, so far, we have not taken into account that we may not be able to spread out all 2^ℓ elements in Y evenly on the remaining entries in L. By construction, however, the difference of assigned elements from Y can differ by at most 1. We can incorporate this into our analysis by using an extra factor 2 for the number of elements which need to be added to Y in each iteration, resulting in the same asymptotic bound. Similarly, since we round up ℓ in the algorithm, it can grow by an additive term 1 at most, which is also "swallowed" by a constant in the asymptotic notation.

In total, for level s of the tree we thus need

$$\mathcal{O}\left(\beta^{-1} \cdot s \cdot 2^\ell + \beta^{-\frac{1}{2}} \cdot 2^{\frac{n-\ell+2s}{2}}\right) = \mathcal{O}\left(\beta^{-1} \cdot 2^{\ell+\log_2 s} + \beta^{-\frac{1}{2}} \cdot 2^{\frac{n-\ell+2s}{2}}\right)$$

evaluations on the average. By our choice of ℓ (in dependence of s) the two terms become equal (except for the β factor), such that together with $s \leq k$ the bound simplifies to

$$\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+2s+\log_2 s}{3}}\right) = \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+2s+\log_2 k}{3}}\right).$$

Summing over all k stages we thus get a total number of

$$\begin{aligned}\sum_{s=1}^{k} \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+2s+\log_2 k}{3}}\right) &= \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+\log_2 k}{3}} \cdot \sum_{s=1}^{k} 2^{\frac{2s}{3}}\right) \\ &= \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+\log_2 k}{3}} \cdot 2^{\frac{2k}{3}}\right) \\ &= \mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+2k+\log_2 k}{3}}\right)\end{aligned}$$

compression function evaluations on the average. □

With this new algorithm our adversary is now a straightforward adaption of the basic one, with the enhanced diamond structure generation replacing the basic one:

$\mathcal{A}^{k,\text{off}}_{\text{enhanced}}(1^\lambda, \ell_P)$:

1 : $(V, E) \leftarrow\!\!\$\ \mathsf{Diamond}_{\text{enhanced}}(k)$
2 : let $y_{0,1} \in V$ be the root of (V, E)
3 : $y_{\text{trgt}} \leftarrow h(\mathsf{pad}(B \cdot (\ell_P + k + 1)), y_{0,1})$
4 : **return** $((V, E), y_{\text{trgt}})$

$\mathcal{A}^{k,\text{onl}}_{\text{enhanced}}((V, E), P)$:

1 : leaves $y_{k,1}, \ldots, y_{k,2^k}$ of (V, E)
2 : $(m_{\text{link}}, y) \leftarrow \mathsf{Link}(P, y_{k,1}, \ldots, y_{k,2^k})$
3 : $m_{\text{path}} \leftarrow \mathsf{Path}((V, E), y)$
4 : $S \leftarrow m_{\text{link}} \| m_{\text{path}}$
5 : **return** S

Theorem 10. *Let* H *be a hash function with* β*-balanced compression function* $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$*. Let* $k \in \mathbb{N}$*. Then adversary* $\mathcal{A}^k_{enhanced}$ *mounts a successful Nostradamus attack and needs*

$$\mathcal{O}\left(\beta^{-1} \cdot 2^{\frac{n+2k+\log_2(k)}{3}} + \beta^{-\frac{1}{2}} \cdot 2^{\frac{n-k}{2}} + \ell_P + 1\right)$$

evaluations of h*. In particular, for* $k = \frac{n}{7}$ *we get a total number of*

$$\mathcal{O}\left(\beta^{-1} \cdot \sqrt[3]{n} \cdot 2^{3n/7} + \ell_P + 1\right)$$

evaluations of h *on the average.*

We note that, ignoring the factor $\sqrt[3]{k}$ and the term ℓ_P, and assuming β to be constant, this upper bound matches our lower bound for herding attacks shown in the next section. This means, in order to significantly improve the attack, a different strategy than building a diamond structure must be used. Even then, the general lower bound still applies.

5 Quantum Lower Bound for Nostradamus Attacks

In this section we show a lower bound on the number of hash queries for mounting a quantum Nostradamus attack, assuming that the compression function h behaves like a random function. Our result is based on a query lower bound for finding C-collisions for a random function f, i.e., distinct $x_1, \ldots, x_C$ such that $f(x_1) = f(x_2) = \cdots = f(x_C)$. Liu and Zhandry [LZ19] gave such a lower bound:

Theorem 11 ([LZ19])**.** *Given a random function* $f : X \to Y$ *any quantum algorithm needs to make at least* $\Omega\left(|Y|^{\frac{1}{2} \cdot \left(1 - \frac{1}{2^C - 1}\right)}\right)$ *quantum queries to oracle* f *to find a* C*-collision with constant probability.*

For example, for a threefold collision $C = 3$ one needs at least $|Y|^{3/7}$ many queries. Liu and Zhandry [LZ19] also give a matching upper bound (but which is irrelevant in our setting here).

5.1 General Lower Bound for Nostradamus Attacks

Recall that the general structure of an adversary $\mathcal{A}$ in the Nostradamus attack consists of two stages, on offline stage in which the adversary outputs the target hash value y_{trgt} and some state information st, and then the online-stage adversary receives the prefix P and the state, and outputs the suffix S to match the target hash value. The idea for the lower bound is now to repeatedly run the second-stage adversary to create multiple collisions for the same target hash value y_{trgt}, but for varying prefixes P_i, hoping to collect a C-collision. The total number of queries we make is $q_{\text{off}} + R \cdot q_{\text{onl}}$, where q_{off} is the number of hash evaluations of $\mathcal{A}$ in the offline phase, R is the number of repetitions, and q_{onl} the number of queries of $\mathcal{A}$ in the online phase. If we are able to show that we get a C-collision with this approach, then it follows that this total number of queries must exceed the lower bound in [LZ19], also indicating a lower bound for $\mathcal{A}$'s queries in relation to R.

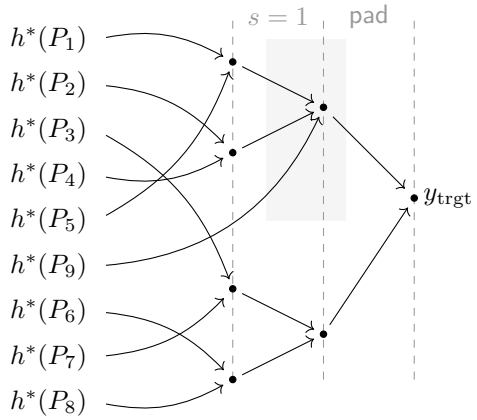

Fig. 6: Finding a C-collision by repeating online phase with different values $P_1, \ldots, P_R$. In the example $C = 3$ and $s = 1$. When running $\mathcal{A}^{\text{onl}}$ with inputs $P_1, \ldots, P_{R-1}$ in this order, we may fill up a $(C-1)$-ary tree with root y_{trgt} from the answers. After $R = (C-1)^{s+2} + 1 = 9$ repetitions we lastly get a C-collision somewhere in the tree (gray box).

Let us explain the process for the concrete example $C = 3$ in Figure 6. Assume that we have already obtained the target hash value y_{trgt} from the offline adversary $\mathcal{A}^{\text{off}}$. Now we run the online part $\mathcal{A}^{\text{onl}}$ several times, presenting the adversary different prefixes $P_1, P_2, \ldots$ in the repetitions. Each time the adversary will give us a suffix path to reach y_{trgt}. Some of these paths may collide before reaching y_{trgt}, e.g., all the suffixes for $P_1, \ldots, P_8$ in Figure 6 have length $s = 1$ and collide earlier or latest in y_{trgt}. The worst case for us occurs if these paths do not yield a C-collision yet. This can only happen if they form a full $(C-1)$-ary tree with root y_{trgt}, with all $(C-1)^{s+2}$ intermediate values $h^*(P_1), h^*(P_2), \ldots, h^*(P_{R-1})$ already forming $(C-1)$-collisions in one of the $(C-1)^{s+1}$ leaves, and the adversary connecting these leaves optimally with the

suffix block and the padding to yield y_{trgt}. But then, if we make the R-th repetition for P_R, where $R = (C-1)^{s+2}+1$, this link must connect to a node which now forms a C-collision. In the example in Figure 6 this happens for $R = 9$.

We next state the above idea formally in the following theorem. Recall that we assume that the adversary succeeds with probability 1 in the Nostradamus attack for any prefix P. We discuss afterwards relaxations for random prefixes and algorithms with lower success probabilities (over the choice of the compression function).

Theorem 12. *Let H be a hash function with compression function $h : \{0,1\}^B \times \{0,1\}^n \to \{0,1\}^n$, and assume that h is a random function. Let $\mathcal{A}$ be a quantum Nostradamus attacker, making at most q_{off} quantum queries to oracle h in the offline phase, and at most q_{onl} quantum queries to oracle h in the online phase. Assume further that $\mathcal{A}$ outputs a suffix S of at most s blocks. Then for any integer $C \geq 3$ such that $(C-1)^{s+2} < 2^B$ we have*

$$q_{off} + ((C-1)^{s+2}+1) \cdot q_{onl} = \Omega\left(2^{\left(1-\frac{1}{2^C-1}\right)n/2}\right).$$

Let us interpret this bound for some concrete cases. For sake of simplicity let us assume that the number of offline and online queries are roughly equal (as is the case in our herding attack). We can thus ignore q_{off} for now, and we also simplify the bound further to:

$$(C-1)^s \cdot q_{\text{onl}} = \Omega\left(2^{\left(1-\frac{1}{2^C-1}\right)n/2}\right).$$

If we fix $C = 3$, for example, then this bound becomes $q_{\text{onl}} = \Omega\left(2^{3n/7-s}\right)$.

It seems as if we can push the lower bound on the right hand side as close to $2^{n/2}$ as desired, by increasing C, in contradiction to our basic attack with $2^{4n/9}$ evaluations and the advanced one with $2^{3n/7}$ evaluations. However, recall that the basic herding attack chooses a tree of height roughly $s = k = n/9$ such that the pre-factor $(C-1)^s$ is at least $2^{n/9}$, yielding an overall product of $2^{5n/9}$ on the left hand side. The same holds for the enhanced attack, where $s = k = n/7$ such that the factor becomes $2^{n/7}$ and hence lifts the value from $2^{3n/7}$ to $2^{4n/7}$.

A remarkable conclusion from the theorem's bound is that, choosing short suffixes for attacks, e.g., of constant block size s, one cannot go significantly below the bound $2^{n/2}$. This holds unless the attack exploits knowledge of the compression function h and does not treat it as a black-box random function. But this square-root bound is already achieved with our simple quantum attack—where indeed the suffix consists of the single linking message block m_{link}. In other words, sophisticated attacks need to rely on long suffixes.

Proof of Theorem 12. Consider an arbitrary Nostradamus adversary $\mathcal{A}$ with the above restrictions. We construct a c-collision finder $\mathcal{C}$ against h as follows. Algorithm $\mathcal{C}$ runs $\mathcal{A}^{\text{off}}$ to derive (st, y_{trgt}). Then it runs $\mathcal{A}^{\text{onl}}$ for $R = (C-1)^{s+2}+1$ times for the same state but different values $P_1, \ldots, P_R \in \{0,1\}^B$. Note that the

requirement on C guarantees that there are so many distinct input blocks. These executions create R suffixes $S_1, \ldots, S_R$ such that for all i the hash evaluations yield the target value, $\mathsf{H}(P_i \| S_i) = y_{\text{trgt}}$. Recall that the final block may contain the padding information such that the suffix part actually consists of up to $s+1$ blocks. Below we simply consider this to be part of the suffix S_i and think of the hash function using no padding.

Assume now that, in the last iteration of h of the R hash evaluations, we have C distinct inputs (m_j, y_j) with $h(m_j, y_j) = y_{\text{trgt}}$. Then we have found the C-collision for h and can stop here. If not, there are at most $C-1$ distinct pairs (m_j, y_j) resulting in y_{trgt}. But then there must exist one value y_j among those pairs which $\lceil R/(C-1) \rceil$ of the total number of hash evaluations reach in the second-to-last iteration. Note that $\lceil R/(C-1) \rceil = (C-1)^{s+1} + 1$ such that we can recursively apply the argument, losing a factor $(C-1)$ in the remaining set of collision with each unsuccessful iteration. After at most $s+1$ iterations we have then either found our C-collision, or have a value y which at least $(C-1)+1 = C$ distinct input blocks $P_{j_1}, P_{j_2}, \ldots$ reach, i.e., $h(IV, P_{j_q}) = y$ for $q = 1, 2, \ldots, C$. Fortunately, this would then constitute our sought-after C-collision. □

We finally discuss how to attenuate the assumption about the adversary always winning in the Nostradamus attack. As remarked earlier, if the adversary's success probability was only over its internal randomness, then we could easily account for this by repeating the adversary. However, now the probability space also comprises the choice of the random compression function (which is not chosen freshly if we would rerun the attack). Hence, assume that the adversary $\mathcal{A}$ has a success probability of ϵ (over the choice of the random function h and its internal coin tosses, like measurements of quantum states). One may think of ϵ to be some small constant, although the approach below also works with other values for ϵ.

We first use the splitting lemma [PS00] to conclude that our once sampled offline-stage output (st, y_{trgt}) is often good enough to make the online stage succeed with probability $\epsilon/2$. The probability of obtaining such a good offline-stage output is at least $\epsilon/2$ itself. Condition on this being the case. Next recall that the target value y_{trgt} is from now on fixed and that $\mathcal{A}$ succeeds with probability $\epsilon/2$ if we give it a prefix P. Hence, to collect $R = (C-1)^{s+2} + 1$ such successful samples as in the proof, we need on average $2R/\epsilon$ attempts now. In other words, we have an expected number of $q_{\text{off}} + \frac{2R}{\epsilon}((C-1)^{s+2} + 1) \cdot q_{\text{onl}}$ queries in order to succeed with probability $\epsilon/2$ (with which the offline-phase output is good).

We can even go one step further and take the prefix to be a random element from $\{0,1\}^B$ (as it was assumed in the original work by Kelsey and Kohno [KK06]). Let us assume it is uniform in $\{0,1\}^B$. Since we expect no collisions to occur before reaching $2^{B/2}$ samples—and B is usually significantly larger than $\log R$—we can assume that all sampled values P_i are distinct. This is sufficient for the argument in the proof.

5.2 Improved Lower Bound for Herding Attacks

In this section we discuss that we get a better lower bound for quantum herding attacks. For this we assume that $\mathcal{A}^{\text{off}}$ creates a diamond tree structure of height k and that $\mathcal{A}^{\text{onl}}$ then tries to find a linking message part m_{link} to connect the prefix P to the tree. We assume here for simplicity of presentation that P is aligned to block length and that m_{link} is a single block. As before, we show how to build a 3-collision finder $\mathcal{C}$ for $C = 3$ from such a herding adversary $\mathcal{A}$. Our approach, however, is more direct and omits the large number of repetitions, thus yielding a better bound.

Our collision finder $\mathcal{C}$ will run the two phases of the herding adversary, $\mathcal{A}^{\text{off}}$ and $\mathcal{A}^{\text{onl}}$, but for different height parameters. For $\mathcal{A}^{\text{off}}$ it will pretend that the height of the tree should be $k + 1$, but then prune the last layer of the tree and give $\mathcal{A}^{\text{onl}}$ the pruned tree of height k. The advantage of this approach is that we already have a 2-collision on the k-th level via the tree for height $k + 1$ from $\mathcal{A}^{\text{off}}$. Together with the new link to level k which we receive from $\mathcal{A}^{\text{onl}}$, we immediately have a 3-collision (see Figure 7).

More formally, $\mathcal{C}$ first lets $\mathcal{A}^{\text{off}}$ create a diamond structure, but uses the parameter $k + 1$. By this we get a tree of height $k + 1$ from $\mathcal{A}^{\text{off}}$. Then we cut level $k+1$ to get a tree of height k as required. Note that here the node values $y_{k,i}$ are, strictly speaking, not picked randomly. But they are the result of applying the random function h to a uniformly chosen input $(y_{k+1,i}, m_{k+1,i})$, such that we assume for simplicity that the herding attacks also works for such generated values.

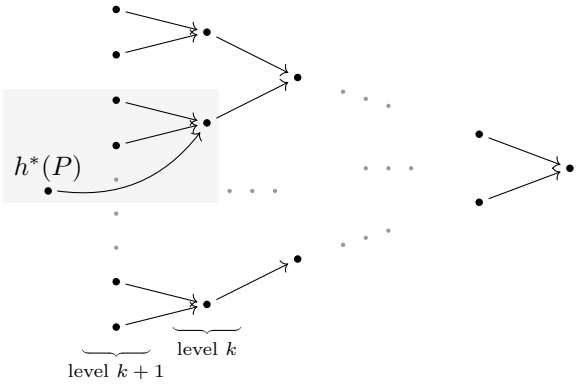

Fig. 7: Finding a 3-collision (gray box) using a diamond structure for height k for $\mathcal{A}$, generated by truncating a $k + 1$-height structure.

Algorithm $\mathcal{C}$ next runs $\mathcal{A}^{\text{onl}}$ for the pruned tree of height k, and an arbitrary prefix P. This adversary eventually outputs a message part m_{link} that connects the iterated prefix value $h^*(P)$ to a tree node $y_{k,i}$ *at level* k (see Figure 7). Algorithm $\mathcal{C}$ outputs the two children $y_{k+1,2i-1}$ and $y_{k+1,2i}$ with their corresponding labels $m_{k+1,2i-1}$ and $m_{k+1,2i}$ pointing to $y_{k,i}$, together with $(h^*(P), m_{\text{link}})$ as the 3-collision for h.

For the analysis of the success probability of $\mathcal{C}$ we would have to make another assumption about the number of preimages under h. But since on average each image has 2^B preimages, we simply neglect the probability of $(h^*(P), m_{\text{link}})$ being equal to $(y_{k+1,2i-1}, m_{k+1,2i-1})$ or $(y_{k+1,2i}, m_{k+1,2i})$. In this case $\mathcal{C}$ succeeds whenever $\mathcal{A}$ wins. For the run time of $\mathcal{C}$ note that creating the diamond structure for $k+1$ instead of k is, asymptotically, equally expensive. Hence, except for constants, our algorithm $\mathcal{C}$ obtains a 3-collision with the same number of oracle queries to h as $\mathcal{A}$. It follows that the total number of quantum queries of $\mathcal{C}$—and thus of $\mathcal{A}$—to random oracle h is at least $\Omega(2^{3n/7})$.

6 Implementation Results

In this section, we empirically evaluate the algorithms from Sections 4.1, 4.2, and 4.3. For this purpose we have implemented the quantum algorithms in IBM's Qiskit.[4] The open-source software development kit Qiskit makes it possible to design quantum circuits in Python and to simulate their execution on a classical computer. Potentially, these algorithms can also be later run on quantum computing devices.

6.1 Algorithms

The local simulation of the quantum algorithm severely restricts the number of simultaneously accessible qubits. Hence, instead of using output-truncated versions of SHA2 or SHA3 with large internal states, we use an iterative toy hash function with adjustable block size B and output n as the attack target for the algorithms.

We build this toy hash function in a similar way as the open-source project Qibo [ECBP+22], which is used to evaluate generic quantum attacks by Ramos-Calderer et al. [RBL+21] and utilizes a ChaCha-permutation [Ber08] as sponge function. In particular, we use this permutation $f : \{0,1\}^{B+n} \to \{0,1\}^{B+n}$ as the basis for our hash function and truncate it (to the last n bits) to derive our compression function

$$h(m, y) := \mathsf{trunc}(f(m \| y))$$

for the Merkle-Damgård construction.

The quantum circuit corresponding to the permutation f is shown in (the right hand side of) Figure 8. We use parameters $B = 8$ and $n = 8$ for the compression function throughout the evaluation. We note that empirically evaluations show that the compression function h_y is not surjective for all y, such that it not β-balanced for any $\beta > 0$. For measuring the run times we ignore such failures. However, as we discuss in more detail later, our advanced attacks still work well, showing that the theoretical results are on the conservative side.

[4] `https://qiskit.org/`

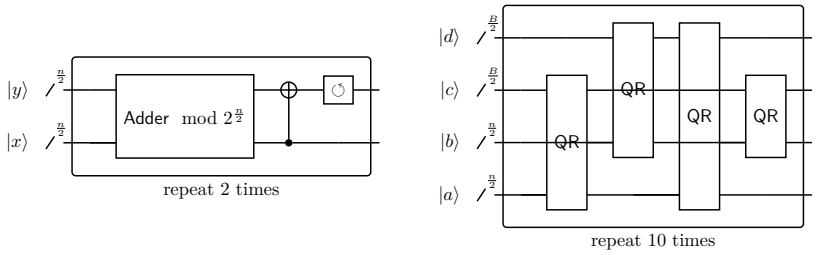

Fig. 8: The description of the quantum circuit for the permutation that is based on the ChaCha permutation. The left part presents the description of the sub algorithm Quarter Round, which is denoted as QR. A quarter round consists of an adder, CNOT-gates, and qubit shuffling. The latter is denoted by the ↻ symbol. The right part describes the permutation f. In the 10 repetitions of the core function, four QR executions are applied to the input registers. We note that a line which is drawn through a gate is not an input to that specific gate but rather routed through.

Recall that our algorithms apply Grover's algorithm for different functions F for finding claws in a list of values. To implement these functions F from Proposition 5 and Theorem 9 in a quantum circuit, we hardcode the list into the circuit, using one n-bit Toffoli gate for each of the 2^ℓ distinct bit strings $h_y(m_i)$ in the list resp. the 2^k distinct leaves $y_{k,i}$. More precisely, given a bit string that either represents a hash evaluation $h_y(m_i)$ or a leaf $y_{k,i}$, an n-bit Toffoli gate is constructed such that the corresponding classical logical operator exclusively maps this bit string to 1. These Toffoli gates are chained together to form the existential quantification in the proposition. The resulting quantum circuit is illustrated in Figure 9.

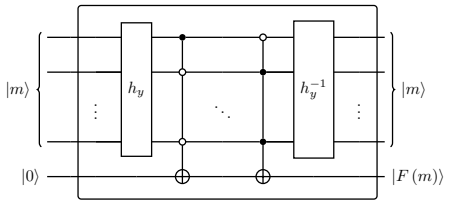

Fig. 9: Description of the quantum circuits that implements the functions F from from Proposition 5 and Theorem 9. The string y in $h_y(\cdot) = h(\cdot, y)$ corresponds either to y' or to p, depending on the context. Each n-bit Toffoli gate represents one (unique) string to which the input is compared to, such that the exclusive-or on the final qubit is only set at most once. Applying h_y^{-1} at the end restores the original content of $|m\rangle$.

We note that implementing Grover's algorithm for our functions F, also allowing to recover the matching message m, as well as the other steps of the algorithms are straightforward to implement. We thus omit their description here for sake of brevity.

6.2 Experiments

The evaluation was carried out with commodity hardware, namely, a 2.6 GHz 6-core Intel Core i7 processor, an AMD Radeon Pro 5300M 4 GB graphics card, and a 16 GB 2667 MHz DDR4 RAM. The evaluation process consists of running all three algorithms (simple, basic, and enhanced). We measure the offline and online run times, as well as the number of sampling operations and function calls. The latter includes the number of calls to the quantum compression function and the number of quantum calls to the functions from Proposition 5 and Theorem 9. The results of the evaluation are shown in Figure 10.

Simple (Section 4.1)

k	Offline [s]	Online [s]	F_{claw}	F_{link}	Samples
—	0	147.93	—	—	—

Basic (Section 4.2)

k	Offl. [s]	Onl. [s]	F_{claw}	F_{link}	Sampl.
1	89.55	232.47	3	8	10
2	260.73	142.52	9	5	28
3	626.13	90.95	21	3	68
4	1363.76	62.13	45	2	138
5	2824.66	34.17	93	1	290

Enhanced (Section 4.3)

k	Offl. [s]	Onl. [s]	F_{claw}	F_{link}	Sampl.
1	62.73	235.34	2	8	18
2	190.2	145.49	6	5	50
3	429.95	90.27	14	3	84
4	927.26	63.83	30	2	138
5	1459.14	31.69	46	1	285

Fig. 10: Evaluation results for the algorithms from Sections 4.1, 4.2, and 4.3 and our compression function $h(m, y)$. The parameter k denotes the height of the tree of the diamond. *Online* and *Offline* denote the online and offline run time, respectively, in seconds. F_{claw} and F_{link} denote the quantum function calls to the functions F from Proposition 5 and 7, respectively. *Samples* denotes the number of sampling operations for messages and leaves for building the diamond structure during an attack. Note that the simple attack fails in some cases for our specific hash function in which case we do not measure its run time.

Note that for $k = 1$, i.e., trees consisting only of one level, the sum of calls (i.e., $F_{claw} + F_{link}$) to the functions F from Proposition 5 and 7 is minimal. This is true for the attack with the basic and the enhanced diamond structure, confirming the optimal choice of $k = \lceil n/9 \rceil$ for $n = 8$ according to Theorems 8 and 9. Furthermore, for $k = 1$ the sum of function calls for the basic and enhanced online attacks (i.e., F_{link}) already is less or equal to the 16 quantum function calls in the simple quantum attack based on Grover's algorithm. Nevertheless, the run time of the attacks with a diamond structure is larger compared to the run time of the simple attack. We expect this to be related to the fact that the quantum circuit of Grover's algorithm for the functions of Proposition 5 and 7 is much more complex than when only applied to the compression function in the simple attack. More precisely, with our design of the circuit in Figure 9, Qiskit simulates the compression function twice and an exponential number of n-bit Toffoli gates, compared to only one simulated compression function call in the simple quantum attack, such that the advantage of using a diamond structure does not pay off for the small value of n yet.

Initially, the enhanced diamond construction requires more samples than the basic diamond construction, while it also exhibits a smaller sum of calls (i.e., $F_{claw} + F_{link}$) to the functions F. For the construction of larger diamonds, such as $k = 4$, the sample count of the enhanced attack is surpassed by the basic diamond construction. This is due to the fact that the enhanced attacker reuses data across the connection steps of a single layer in the diamond, while the basic attacker resamples all data for each quantum collision finding. Thus, our experiments confirm that the advanced attacker requires less sampled data to construct a large diamond.

Instead of using the optimal run time, an attacker may also deploy a trade-off and, for a slight decrease in online run time, accept an increase in the offline run time. In this case, the attacker creates a larger diamond for $k > 1$ and thereby achieves an improvement for the online run time. Our experiments confirm this expected behavior, with F_{link} dropping and F_{claw} increasing.

We previously noted that $h(m, y)$ is not even surjective and not β-balanced. Since the simple attack from Section 4.1 picks a random value $y \leftarrow_{\$} \{0,1\}^n$ this results in the attack failing for values y without preimage. In this case, Grover's algorithm cannot find a corresponding preimage. However, the more advanced attacks from Sections 4.2 and 4.3 always succeeded. The reason is that the trees are built in forward direction, such that each value y has at least two preimages.

We note that all the observations are specific to the parameters $n = B = 8$. According to our theoretical results the asymptotic complexity of the simple quantum attack increases faster in n than it does for the diamond attacks. As a consequence, for higher values of n a larger difference in online run time should occur even for smaller k. Unfortunately, the simulation of higher n in Qiskit is expensive and, in some cases, technically infeasible due to the large sizes of quantum circuits. Thus, the empirical demonstration remains open.

7 Conclusion

Our results show that fundamental quantum algorithms for finding collisions can be used to speed up the classical Nostradamus attack. Our algorithms have been designed and analyzed in an "idealized" quantum model, but our implementation of the toy example indicates that they can be run on in principle on a quantum computer. Turning these attacks into real quantum programs may still entail a lot of engineering aspects which can significantly influence the run time, e.g., [AMG+16,Ber09,BJ22]. It remains an interesting open question how fast our attacks can be made on real quantum computers.

As mentioned in the related work section, other works like [BB17,CNS17] have aimed to give time-memory trade-offs, especially in order to reduce quantum memory, and to parallelize the search for collisions and preimages. We have not investigates such trade-offs for Nostradamus attacks, especially in light of the large quantum memory requirements, inherited from the BHT collision search algorithm [BHT98]. Nonetheless, we expect similar techniques as in [BB17,CNS17] to be applicable here as well.

Acknowledgments

We thank the anonymous reviewers for valuable comments.

This research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

ABDK09. E. Andreeva, C. Bouillaguet, O. Dunkelman, and J. Kelsey. Herding, second preimage and trojan message attacks beyond merkle-damgård. In *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009, Calgary, Alberta, Canada, August 13-14, 2009, Revised Selected Papers*, pages 393–414, 2009.

AM11. E. Andreeva and B. Mennink. Provable chosen-target-forced-midfix preimage resistance. In *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, pages 37–54, 2011.

AMG+16. M. Amy, O. D. Matteo, V. Gheorghiu, M. Mosca, A. Parent, and J. M. Schanck. Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3. In *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, pages 317–337, 2016.

BB17. G. Banegas and D. J. Bernstein. Low-communication parallel quantum multi-target preimage search. In *Selected Areas in Cryptography - SAC 2017 - 24th International Conference, Ottawa, ON, Canada, August 16-18, 2017, Revised Selected Papers*, pages 325–335, 2017.

BBHT98. M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(4-5):493–505, 1998.

BDF+11. D. Boneh, Ö. Dagdelen, M. Fischlin, A. Lehmann, C. Schaffner, and M. Zhandry. Random oracles in a quantum world. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, pages 41–69, 2011.

BDPV07. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Sponge functions. Ecrypt Hash Workshop, 2007.

Ber08. D. Bernstein. ChaCha, a variant of Salsa20, 2008. `https://cr.yp.to/chacha/chacha-20080128.pdf`.

Ber09. D. J. Bernstein. Cost analysis of hash collisions : will quantum computers make SHARCS obsolete? In *SHARCS'09 Workshop Record (Proceedings 4th Workshop on Special-purpose Hardware for Attacking Cryptograhic Systems, Lausanne, Switserland, September 9-10, 2009)*, pages 105–116, 2009.

BHT98. G. Brassard, P. Høyer, and A. Tapp. Quantum cryptanalysis of hash and claw-free functions. In *LATIN '98: Theoretical Informatics, Third Latin American Symposium, Campinas, Brazil, April, 20-24, 1998, Proceedings*, pages 163–169, 1998.

BJ22. X. Bonnetain and S. Jaques. Quantum period finding against symmetric primitives in practice. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):1–27, 2022.

BK04. M. Bellare and T. Kohno. Hash function balance and its impact on birthday attacks. In *Advances in Cryptology – EUROCRYPT 2004*, pages 401–418, 2004.

BSU12. S. R. Blackburn, D. R. Stinson, and J. Upadhyay. On the complexity of the herding attack and some related attacks on hash functions. *Des. Codes Cryptogr.*, 64(1-2):171–193, 2012.

CNS17. A. Chailloux, M. Naya-Plasencia, and A. Schrottenloher. An efficient quantum collision search algorithm and implications on symmetric cryptography. In *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*, pages 211–240, 2017.

Dam90. I. Damgård. A design principle for hash functions. In *Advances in Cryptology – CRYPTO'89*, pages 416–427, 1990.

Dan15. Q. Dang. Secure hash standard. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2015.

Dea99. R. D. Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Computer Science Department, Princeton University, 1999.

DSS+20. X. Dong, S. Sun, D. Shi, F. Gao, X. Wang, and L. Hu. Quantum collision attacks on AES-like hashing with low quantum random access memories. In *Advances in Cryptology – ASIACRYPT 2020, Part II*, pages 727–757, 2020.

Dwo15. M. Dworkin. SHA-3 standard: Permutation-based hash and extendable-output functions. Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, 2015.

DZS+21. X. Dong, Z. Zhang, S. Sun, C. Wei, X. Wang, and L. Hu. Automatic classical and quantum rebound attacks on aes-like hashing by exploiting related-key differentials. In *Advances in Cryptology - ASIACRYPT 2021 - 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6-10, 2021, Proceedings, Part I*, pages 241–271, 2021.

ECBP+22. S. Efthymiou, S. Carrazza, C. Bravo-Prieto, AdrianPerezSalinas, S. Ramos, D. García-Martín, M. Lazzarin, N. Zattarin, A. Pasquale, Paul, and J. Serrano. Qibo: An open-source full stack api for quantum simulation and quantum hardware control, 2022.

FGLN+20. A. Flórez-Gutiérrez, G. Leurent, M. Naya-Plasencia, L. Perrin, A. Schrottenloher, and F. Sibleyras. New results on gimli: Full-permutation distinguishers and improved collisions. In *Advances in Cryptology – ASIACRYPT 2020, Part I*, pages 33–63, 2020.

Gro96. L. K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 212–219, 1996.

HS20. A. Hosoyamada and Y. Sasaki. Finding hash collisions with quantum computers by using differential trails with smaller probability than birthday bound. In *Advances in Cryptology – EUROCRYPT 2020, Part II*, pages 249–279, 2020.

HS21. A. Hosoyamada and Y. Sasaki. Quantum collision attacks on reduced SHA-256 and SHA-512. In *Advances in Cryptology – CRYPTO 2021, Part I*, pages 616–646, 2021.

KK06. J. Kelsey and T. Kohno. Herding hash functions and the nostradamus attack. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, pages 183–200, 2006.

KK13. T. Kortelainen and J. Kortelainen. On diamond structures and trojan message attacks. In *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II*, pages 524–539, 2013.

KS05. J. Kelsey and B. Schneier. Second preimages on n-bit hash functions for much less than 2^n work. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 474–490, 2005.

LZ19. Q. Liu and M. Zhandry. On finding quantum multi-collisions. In *Advances in Cryptology – EUROCRYPT 2019*, pages 189–218, 2019.

Mer90. R. C. Merkle. A certified digital signature. In *Advances in Cryptology – CRYPTO'89*, pages 218–238, 1990.

NDJY21. B. Ni, X. Dong, K. Jia, and Q. You. (quantum) collision attacks on reduced simpira v2. *IACR Trans. Symmetric Cryptol.*, 2021(2):222–248, 2021.

PS00. D. Pointcheval and J. Stern. Security arguments for digital signatures and blind signatures. *Journal of Cryptology*, 13(3):361–396, 2000.

RBL+21. S. Ramos-Calderer, E. Bellini, J. I. Latorre, M. Manzano, and V. Mateu. Quantum search for scaled hash function preimages. *Quantum Inf. Process.*, 20(5):180, 2021.

WDH17. A. Weizman, O. Dunkelman, and S. Haber. Efficient construction of diamond structures. In *Progress in Cryptology - INDOCRYPT 2017 - 18th International Conference on Cryptology in India, Chennai, India, December 10-13, 2017, Proceedings*, pages 166–185, 2017.

WLG+22. R. Wang, X. Li, J. Gao, H. Li, and B. Wang. Quantum rotational cryptanalysis for preimage recovery of round-reduced keccak. *IACR Cryptol. ePrint Arch.*, page 13, 2022.

A Quantum Attacks on Sponge-based Hash Functions

We briefly discuss how to transfer our quantum attacks to sponge-based hash functions. As we will see, the best attack option depends now on the relationship of the two parameters for output size and for the size of the inner state. We therefore discuss the various options in light of the actual SHA3 and SHAKE candidates [Dwo15].

A.1 Sponge-based Hash Functions

Sponges split the current state into a *rate* part of r bits and a *capacity* part of c bits. Starting with the all zero vector $IV = 0^{r+c}$ one mixes the next r bits m_i of the message to the intermediate state via exclusive-or, and then applies a round permutation f to the entire state: $y_i = f(y_{i-1} \oplus m_i \| 0^c)$. In analogy to the case before, we can thus write $y_i = h(m_i, y_{i-1})$ with $y_0 = IV = 0^{r+c}$, and for a message $m_1 \| \ldots \| m_\ell$ consisting of full r-bit blocks m_i we write $h^*(m_1 \| \ldots \| m_\ell)$ for $y_\ell \in \{0,1\}^{r+c}$.

Algorithm	rate r	capacity c	output size n
SHA3-224	1152	448	224
SHA3-256	1088	512	256
SHA3-384	832	768	384
SHA3-512	576	1024	512
SHAKE128	1344	256	variable
SHAKE256	1088	512	variable

Fig. 11: Parameters for the SHA3 and SHAKE families.

In the squeezing phase one extracts the output bits. For SHA3 with fixed output length one may produce $n = 224, 256, 384$ or 576 output bits, depending on the parameter of the function family [Dwo15]. We note that the choice of n also determines the rate and capacity, see Figure 11. The padding for SHA3 is given by the message suffix 01 followed by 10^j1, where j is such that the padded message is aligned to an integer multiple of $r + c$. The padding thus fits into at most one extra block, but the message length does not need to be available beforehand. The output of all SHA3 algorithms in simply the truncation to $n < r$ bits of the final result, without requiring further iterations of f.

The extendable output function SHAKE, also standardized by NIST [Dwo15], has a variable output length (which, in our setting, means that the adversary determines the output size when picking y_{trgt}). It comes in two variants, one with capacity 256 one with 512 (see again Figure 11). The message padding consists of a message suffix 1111 followed by 10^j1. In the output (or squeezing) phase one iterates the intermediate value further for permutation f, extracting the rate part of r bits in each iteration till sufficiently many bits have been collected.

A.2 Attacks on SHA3

For SHA3 the simple quantum attack in Section 4.1, where one picks a random output and then runs Grover's algorithm to map the state after processing the prefix P to this output, outperforms the advanced attacks. This is true as long as $n < c$ which is the case for all parameter choices of the SHA3 family. Let us explain the modifications of the simple attack we need to make. In the offline phase we simply pick a random value y_{trgt} as the target n-bit output. When receiving a prefix P in the online phase, we iterate the sponge-construction to get the full state p for this prefix, and then run Grover to find a message block S mapping the truncated output of the final evaluation of f to y_{trgt}. Here S is only $r-4$ bits, because we need to put the SHA3 message suffix 01 and the minimal padding 11 at the end:

$\underline{\mathcal{A}^{\text{off}}_{\text{simple}}(1^\lambda)\text{: } /\!/ \text{ offline}}$

1 : $y_{\text{trgt}} \leftarrow\!\!\$\ \{0,1\}^n$

2 : **return** y_{trgt}

$\underline{\mathcal{A}^{\text{onl}}_{\text{simple}}(y,P)\text{: } /\!/ \text{ online}}$

1 : $p \leftarrow h^*(P)$

2 : $S \leftarrow \mathsf{Grover}(h_p, y_{\text{trgt}}) /\!/ h_p(\cdot) = h(\cdot \| 0111, p)$

3 : **return** S

Assuming that the truncation of permutation f to the first n bits is β-balanced, the function h_p with the fixed four bits is at least $\beta/16$-balanced. Hence, Grover's algorithm finds a suitable input S with an expected number $\mathcal{O}(\sqrt{16 \cdot 2^{r-4}/\beta 2^{r-n}}) = \mathcal{O}(\beta^{-1/2} \cdot 2^{n/2})$ of evaluations of h and thus of f. This still improves over the classical Nostradamus bound.

In principle, one could also use our advanced herding strategies for SHA3. However, this does not seem to enhance the bound compared to the simple attack, as we explain next. Since we only illustrate why the herding attack does not give any advantage, we assume for simplicity that all the considered hash functions are well balanced and that β is constant. The central algorithm for both diamond constructions is the Claw algorithm (which is implicit in the enhanced herding attack with an incrementally constructed list). We can apply the same strategy here for the hash function $h(m,y) = f(y \oplus m \| 0^c)$, mapping inputs $m \in \{0,1\}^r$ and $y \in \{0,1\}^{r+c}$ to $r+c$ bits output. An optimization is to only consider collisions on the capacity part of c bits in the outputs. We can later "patch" diverging rate parts by picking the message blocks for the next iteration accordingly. Specifically, if we have a "capacity collision" $h(m,y)$,$h(m',y')$ then we can take the exclusive-or Δ of the rate parts, and add Δ to one of the two next message blocks. See Figure 12.[5]

If we now apply algorithm Claw as before, looking at the c-bit truncated output of h, then we get a bound $\mathcal{O}\left(2^\ell + 2^{\frac{c-\ell}{2}}\right)$ of expected evaluations of f, with

[5] We note that if the capacity c exceeds the rate r, as for SHA3-512, then not all capacity parts $z \in \{0,1\}^c$ actually have a preimage under $h_y(\cdot)$ and we may not be able to use Grover's algorithm. In this case one would have to look at the full intermediate state. We assume to the advantage of the attacker here, that $r \leq c$, as for SHA3-224, SHA3-256, and SHA3-384, and that the adversary can always use the optimization via "capacity collisions".

c substituting n. This is again optimized by setting $\ell = c/3$ such that we get the overall bound of $\mathcal{O}\left(2^{c/3}\right)$ for this step. Plugging this into the basic and the enhanced herding algorithms, we obtain the bounds $\mathcal{O}\left(2^{4c/9}\right)$ resp. $\mathcal{O}\left(\sqrt{c} \cdot 2^{3c/7}\right)$. But note that for SHA3 we have $c = 2n$, such that the exponents in both cases would be close to n. This, however, is worse than the exponent $n/2$ for the simple attack.

A.3 Attacks on SHAKE

For the extendable output function SHAKE the capacity is either 256 or 512. Here the best attack strategy depends on the relationship of the actual output length n and the capacity $c \in \{256, 512\}$. Let us first consider the simple attack, either searching for a linking message block for the output or for the capacity part of an intermediate value. We show that we obtain a successful attacker with $\mathcal{O}(2^{\min\{c,n\}/2})$ evaluations of the permutation f (assuming balanced functions for constant β). This matches the classical bound for collision resistance for SHAKE.

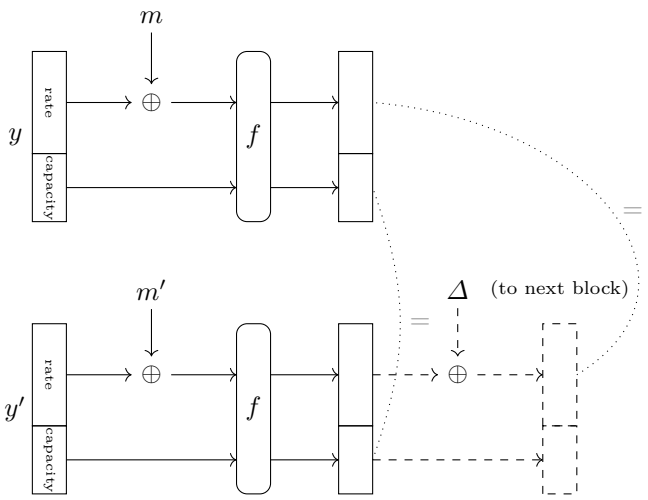

Fig. 12: Patching collisions on capacity part for sponge-based hash functions. If we have two values colliding on the capacity part (lower rectangle) then we can add the difference Δ to the rate part to achieve equality on the entire state. Formally, we add this value Δ to the next message block later, and continue with the inner state of the upper branch.

For $n \leq c$ the best option for the simple attack is to adopt the simple attack against SHA3 as above, sampling $y_{\text{trgt}} \leftarrow\!\!\$\, \{0,1\}^n$ randomly, and then searching for a message block (with padding) generating the same output sequence. Note that this may require to iterate f, viewing $h_{y_{\text{trgt}}}(\cdot \| 111111)$ as the squeezing step. This yields a bound $\mathcal{O}(2^{n/2})$ if we assume that the squeezing step is well balanced.

If $c < n$, on the other hand, then we rather search for a matching preimage for the capacity part. That is, pick $y \leftarrow_{\$} \{0,1\}^{r+c}$ and iterate once with an empty message block, and once for the padding step in an extra block by computing $y' = f(f(y) \oplus 111110^j 1 \| 0^k)$ for $j = 1338$, $k = 256$ (SHAKE128) or $j = 1082$, $k = 512$ (SHAKE256). Compute the squeezing output y_{trgt} from y'. Then, given a prefix P and the iteration value $p = h^*(P)$, use Grover's algorithm to search for a message block m_{link} such that $h_p(m_{\text{link}})$ coincides on the capacity part with y. Let $\Delta \in \{0,1\}^r$ be the difference in the rate part between y and $h_p(m_{\text{link}})$. Set $S \leftarrow m_{\text{link}} \| \Delta$. With these choices we then have

$$h^*(P \| S) = f(\ f(p \oplus m_{\text{link}} \| 0^c)\ \oplus \Delta \| 0^c) = f(y).$$

Hence, we generate the same output. Assuming that the truncation to the capacity part is well balanced, we find the message block m_{link} in $\mathcal{O}(2^{c/2})$ evaluations.

We can analogously transfer the herding attacks. Here, once more, the capacity takes the role of the output size n, such that we obtain the approximate bounds $\mathcal{O}(2^{4c/9})$ and $\mathcal{O}(2^{3c/7})$ for the basic and enhanced diamond structure (assuming balanced outputs). The latter bound only gives an advantage over the simple attack if c is at most $7n/6$. Neglecting constants, this means for SHAKE128, for example, that one needs to produce at least 220 bits output.