Trapdoor Memory-Hard Functions

Benedikt Auerbach
benedikt.auerbach@ista.ac.at
ISTA

Christoph U. Günther
cguenthe@ista.ac.at
ISTA

Krzysztof Pietrzak
pietrzak@ista.ac.at
ISTA

February 23, 2024

Abstract

Memory-hard functions (MHF) are functions whose evaluation provably requires a lot of memory. While MHFs are an unkeyed primitive, it is natural to consider the notion of *trapdoor* MHFs (TMHFs). A TMHF is like an MHF, but when sampling the public parameters one also samples a trapdoor which allows evaluating the function much *cheaper*.

Biryukov and Perrin (Asiacrypt'17) were the first to consider TMHFs and put forth a candidate TMHF construction called Diodon that is based on the Scrypt MHF (Percival, BSDCan'09). To allow for a trapdoor, Scrypt's initial hash chain is replaced by a sequence of squares in a group of unknown order where the order of the group is the trapdoor. For a length n sequence of squares and a group of order N, Diodon's cumulative memory complexity (CMC) is $O(n^2 \log N)$ without the trapdoor and $O(n \log(n) \log(N)^2)$ with knowledge of it.

While Scrypt is proven to be optimally memory-hard in the random oracle model (Alwen et al., Eurocrypt'17), Diodon's memory-hardness has not been proven so far. In this work, we fill this gap by rigorously analyzing a specific instantiation of Diodon. We show that its CMC is lower bounded by $\Omega(\frac{n^2}{\log n} \log N)$ which almost matches the upper bound. Our proof is based Alwen et al.'s lower bound on Scrypt's CMC but requires non-trivial modifications due to the algebraic structure of Diodon. Most importantly, our analysis involves a more elaborate compression argument and a solvability criterion for certain systems of Diophantine equations.

Contents

1 Introduction

Moderately-hard functions have many applications, the most prominent one being password hashing. Early constructions of such functions aimed to be moderately hard in terms of computation. For example, PBKDF2 [Kal00] is essentially a regular hash function repeated sufficiently many times.

Unfortunately, computationally expensive functions are not *egalitarian*. Attackers can use specialized hardware (like FPGAs or ASICs) to evaluate some specific computationally expensive function several orders of magnitude more efficiently (in terms of energy and hardware cost) than general-purpose hardware. This creates an asymmetry between the cost for honest users and attackers. Specifically with password hashing in mind, Percival [Per09] introduced the notion of *memory-hard functions* (MHFs).

1.1 Memory-Hard Functions

Informally, a function is memory-hard if its evaluation cost on general-purpose hardware is dominated by the memory (rather than CPU) cost. Since memory costs are roughly the same for specialized- and general-purpose hardware, MHFs are more egalitarian than computationally-hard functions. The first MHF construction was Scrypt [Per09] followed by many others (e.g., [BDK16, BCS16, ABH17, BHK+19, BH22]). These constructions primarily differ in their notion of memory-hardness and side-channel resistance.

Memory-Hardness. A popular way to measure the memory-hardness of a function is the *cumulative memory complexity* (CMC) [AS15]. It basically sums the memory cost over all steps[1] of an evaluation. A secure MHF not only ensures that the CMC of the honest evaluation algorithm is high, but also that any other, adversarial evaluation algorithm has a CMC not much lower than the honest evaluation algorithm. While the honest evaluation algorithm uses little to no parallelism, the adversarial algorithm is allowed arbitrarily many parallel queries to the random oracle.

Unfortunately, the definition of CMC does not exclude time-memory trade-offs. In particular, an MHF may be evaluated in more steps where each step requires less memory [RD16]. A stronger notion is sustained space complexity [ABP18] which only counts steps where the memory usage is sufficiently high.

Finally, let us mention the related notion of bandwidth-hardness [RD17, BRZ18] capturing the number of cache-misses rather than the memory usage. This captures the energy cost of evaluating a function more accurately, whereas CMC is a better measure for the hardware cost.

Side-channel Resistance. MHFs come in two flavors, *data-dependent* and *data-independent MHFs* (dMHF and iMHF, respectively), which classify the side-channel resistance of MHFs. The memory-access patterns during the evaluation of an iMHF do not depend on the input but are fixed. In contrast, dMHFs allow the memory-access patterns to depend on the input. While dMHFs are easier to construct and can provably

[1] MHFs are typically constructed from hash functions. In security proofs, these are modelled as random oracles. So a "step" may be thought of as a query (or many independent queries in parallel) to the random oracle.

achieve higher evaluation cost [ABP17], their security can be compromised by side-channel attacks which leak memory access patterns. A notion aiming to combine the advantages of iMHFs and dMHFs using computational assumptions was proposed in [ABZ20].

1.2 Trapdoor MHFs

An MHF is an unkeyed primitive, but like for other unkeyed primitives, say, one-way permutations or collision-resistant hash functions, it is natural to consider a keyed version. Biryukov and Perrin [BP17] introduced *asymmetrically memory-hard functions* which are essentially MHFs admitting a trapdoor. So similarly to an MHF, evaluating the function is guaranteed to cost a lot of memory in general. However, in contrast to an MHF, when sampling the parameters one also generates a secret trapdoor. Knowledge of this trapdoor allows evaluating the function much cheaper using less memory. Therefore, we call such functions *trapdoor MHFs* (TMHFs) and note that the PURED framework [BLP23]—an effort to classify all types of resource-hard functions—also uses the word "trapdoor" to describe such functions.

In this work, we focus on a data-dependent TMHF that provably achieves high CMC. To precisely quantify the gap between the honest evaluation algorithms (with and without knowledge of the trapdoor) we use the notation $(c_{\text{hon}}, c_{\text{td}})$-TMHF. This means that the honest evaluation algorithms without and with knowledge of the trapdoor have a CMC of c_{hon} and c_{td}, respectively.

Applications. One potential application of TMHFs are proofs of CMC that benefit from efficient private verification. For example, consider an e-mail server that wants to combat junk mail [DN93] by requiring e-mail senders to solve a TMHF. Thanks to the trapdoor, the server can verify the sender's response with fewer resources than the sender. So the server can choose larger parameters that would otherwise be too costly, especially if the gap between c_{hon} and c_{td} is large.[2]

Another application is password hashing, albeit a less convincing one. In principle, the trapdoor allows a server to verify logins more cheaply compared to using an MHF. However, the trapdoor needs to be stored securely (e.g., in an HSM) and then it is conceivable to simply encrypt the passwords instead of hashing them. One downside of encryption is that a compromise of the key (e.g., the HSM is broken) reveals all passwords, a catastrophical failure. Since TMHFs would still require some bruteforcing in this case, they might still be preferable.

Related Primitives. Memory-hard puzzles [ABB22] are closely related, yet different. Solving a puzzle is memory-hard, but it is easy to sample a puzzle that will evaluate to a specific solution. While the first property is comparable to evaluating a TMHF without knowledge of the trapdoor, the second property can roughly be seen as the opposite of the trapdoor evaluation.

[2] In a similar vein, some proof of work blockchains use MHFs, but the parameters cannot be too large as otherwise the verification of blocks becomes too expensive. Sadly, TMHF are not suited for blockchains since it is unclear who would possess the trapdoor.

1.3 The Diodon TMHF

Diodon [BP17] is the first construction aiming to be TMHF. It is based on Scrypt [Per09], a well-known data-dependent MHF that has provably high memory-hardness [ACK+16, ACP+17]. Diodon as stated in [BP17] offers multiple parameters to fine-tune the security and performance of the function. In this paper, we stick to the natural choice of parameters that most closely resembles Scrypt[3] and also work in a slightly different algebraic setting. To avoid confusion, we call this specific instantiation TdScrypt.

TdScrypt essentially replaces the sequential hashing done in Scrypt's initial phase with sequential squaring in a group of unknown order. More precisely, TdScrypt is defined with respect to a group of unknown order where the trapdoor is the group order N. Given a parameter n which basically specifies the memory requirement of the evaluation and input group element $W =: W_0$, first define $W_i := W_{i-1}^2$ for $0 < i < n$. Then set $S_0 := \mathsf{h}(W_n, 0 \cdots 0)$, and, for $0 < i \leq n$, define $S_i := \mathsf{h}(W_j, S_{i-1})$ where $j := S_{i-1} \bmod n$ and h is a hash function. Finally, S_n is the output of TdScrypt on input W.

Evaluation Algorithms. The honest evaluation algorithm without knowledge of the trapdoor repeatedly squares W, and stores all intermediate values (i.e., all W_i) in memory. Then it computes all S_i in sequence, looking up W_j on demand. Adding up the memory consumed over all steps, we get a CMC of $\Theta(n^2 \log N)$ because the algorithm stores n group elements of size roughly $\log N$ bits while computing the values from S_0 to S_n within n steps.[4]

In comparison, the trapdoor evaluation algorithm first computes $W_n = W^{2^n}$ directly by first reducing $2^n \bmod N$. Then, it computes the S_i sequentially while computing the W_j on demand similarly to W_n. This has a cost of roughly $\Theta(n \log(N)^2 \log(n))$ and we defer the details to Section 3.2. In summary, TdScrypt is approximately an $(n^2 \log(N), n \log(n) \log(N)^2)$-TMHF.

1.4 Contributions and Technical Overview

Our main contribution is a rigorous proof of the following lower bound. It bounds the CMC required by any TdScrypt evaluation algorithm without knowledge of the trapdoor.

Theorem 1 (Informal)**.** *In the random oracle and generic group model, assuming that factoring is hard, any algorithm $\mathcal{A}$ evaluating* TdScrypt *has a cumulative memory complexity lower bounded by $\Omega(\frac{n^2}{\log n} \log N)$.*

Recall that the honest evaluation algorithm (without trapdoor) has a cumulative-memory complexity of $\Theta(n^2 \log N)$, so our lower bound is a factor $1/\log(n)$ loose. We do not know of any evaluation algorithm achieving $O(\frac{n^2}{\log n} \log N)$ and believe that the loss in tightness is an artifact of our proof.

Proof Strategy. We follow the proof of Alwen et al. [ACP+17] who proved that Scrypt has a CMC of $\Omega(n^2 \omega_{\mathsf{h}})$—which is tight—in the *random oracle model* (ROM).[5]

[3]Using notation from [BP17], $M := n$, $L := n$, and $\eta := 1$.

[4]We assume that a hash- and group operations take the same amount of time.

[5]Note that ω_{h} is the output length of the random oracle which corresponds to $\log N$ in our bounds.

Naturally, we also work in the ROM, but also need to consider Shoup's *generic-group model* (GGM) [Sho97]. Their proof first considers a *single-challenge time-memory trade-off* which is then generalized to get a *multi-challenge memory complexity lower bound*. We will elaborate how these two concepts are related to CMC in the following paragraphs.

We remark that our single-challenge time-memory trade-off proof is more involved and differs considerably from Alwen et al. [ACP+17] where this part of the proof was fairly simple. The generalization to the multi-challenge memory complexity lower bound—by far the most complicated part in [ACP+17]—is fortunately essentially identical. Thus, most of our proof focuses on the single-challenge case and we only sketch the multi-challenge argument while referring to [ACP+17] for details.

Single-challenge Time-Memory Trade-Off. Consider the following *single-challenge game*. Like when evaluating TdScrypt, the adversary $\mathcal{A}$ receives an input group element W but does not know the group order N. It is then given the *challenge* $j \xleftarrow{\$} \{0, \ldots, n-1\}$ and needs to output $W_j = W^{2^j}$ as quickly as possible. Before being challenged, $\mathcal{A}$ is allowed to perform precomputation and to store the resulting advice string. Clearly, if the advice string is large enough, $\mathcal{A}$ can store $W_0, \ldots, W_{n-1}$ and answer every challenge instantly. Inspired by this, we are interested in a time-memory trade-off: *How fast (on average) can $\mathcal{A}$ answer a challenge in relation to the size of the advice string?*

To this end, we first show that if $\mathcal{A}$ answers a large fraction of the challenges quickly, it must have a lot of group elements stored, else it could factor. Our approach draws inspiration from proofs showing that repeated squaring (i.e., on input W computing W^{2^j}) [RSW96] is sequential in generic models if factoring is hard [KLX20, Rot22, RS20]. On a high level, they first lower bound the number of queries required by algorithms oblivious to the group order. Using this lower bound, they show that if an adversarial algorithm is faster, its query behavior reveals a non-trivial factor of N. While the lower bound for sequential squaring trivially equals the prescribed number of iterations, figuring out a tight enough bound for the single-challenge game with respect to TdScrypt is substantially more complex. Without delving into the details, we prove a bound by analyzing the solvability of certain systems of Diophantine equations.[6] We use powerful mathematical tools such as a lemma due to van der Waerden [Laz96] and a generalization of the famous distinct subset sums problem due to Erdős [Guy94, C8].

Second, again assuming hardness of factoring, we show that storing k group elements roughly requires a memory of $k \log(N)$ as otherwise we could encode a random injection (the GGM's labeling function) more efficiently than information-theoretically possible. Our proof is inspired by Corrigan-Gibbs and Kogan [CK18] who analyze how helpful preprocessing is for computing discrete logarithms by using an incompressibility argument [DTT10]. In contrast to their work, our argument is more involved. On the one hand, the single-challenge game is more complex than the discrete logarithm problem. On the other, the group order is unknown which complicates bookkeeping in the encoding routine.

Combining the two results above, we get that if an adversary answers challenges quickly on average, then the advice string must be large—assuming factoring is hard.

[6]This means that the systems have integer coefficients. Intuitively, if an algorithm is oblivious to the group order, its query behavior might as well be analyzed over $\mathbb{Z}$ instead of $\mathbb{Z}_N$.

Multi-challenge Time-memory Trade-Off. When an algorithm evaluates TdScrypt and outputs S_n, it almost surely must have computed $S_1, \ldots, S_n$ in sequence because h is a modeled as a random oracle. Computing S_i given S_{i-1} requires W_j by definition where is $j := S_{i-1} \bmod n$ (almost) uniformly random because, again, h is a random oracle. It follows that evaluating TdScrypt requires playing n single-challenge games in sequence. So, a lower bound on the memory complexity of solving multiple challenges implies a lower bound on the CMC of TdScrypt.

1.5 Open Problems

First, TdScrypt has a CMC of $O(n^2 \log N)$ whereas the lower bound is only $\Omega(\frac{n^2}{\log n} \log N)$. Ideally, this bound should be tight. Looking ahead, one possible way of achieving this is strengthening Lemma 2, a solvability criterion for certain systems of Diophantine equations. Essentially, one would need to show $\operatorname{rank}(A) \geq \ell/(ct)$ for some constant c.

Second, TdScrypt's drop in CMC when using the trapdoor is only due to the much lower memory requirement of the trapdoor evaluation. The computation actually increases from n to $n \log(N)$, as in the 2nd phase of the evaluation (where we compute the S_i's) the normal evaluation just makes n group operations (modular multiplications), while the trapdoor evaluation needs to do n exponentiations. An open problem is constructing a TMHF where the trapdoor evaluation not only improves CMC, but strictly improves on memory usage and computation individually (or at least improves on one of them without decreasing the other).

Finally, coming up with a TMHF that fulfills different notions of memory hardness would be interesting. For example, a data-independent TMHF or a TMHF that ensures high sustained space complexity.

2 Preliminaries

2.1 Notation

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}_N$, $\mathbb{Q}$, and $\mathbb{R}$ are the sets of natural numbers including 0, integers, integers modulo N, rational, and real numbers, respectively. For these sets, the superscript $^+$ denotes the strictly positive subset (e.g., $\mathbb{N}^+ = \{1, 2, \ldots\}$). $[a, b]$ denotes the set $\{a, \ldots, b\}$, $[a, b) = \{a, \ldots, b-1\}$, and $[n]$ is a shorthand for $[1, n]$. Furthermore, $\operatorname{Inj}(A, B)$ is the set of all injections from set A to set B and $x \overset{\$}{\leftarrow} X$ samples an element from the set X uniformly at random. Vectors are written in boldface (e.g., $\boldsymbol{x}$) and matrices in upper case. $|x|$ denotes the absolute value or the length of x depending on whether x is a number or a list, vector, etc., and $\|x\|$ is the number of bits required to encode x. Algorithms are usually typeset sans-serif (e.g., Alg), $x := y$ or $x := \mathsf{Alg}(\ldots)$ denote assignment or the output of a deterministic algorithm, and $a \leftarrow \mathsf{Alg}(\ldots)$ the output of a probabilistic one.

λ always denotes the security parameter and $\operatorname{negl}(\lambda)$ (resp. $\operatorname{poly}(\lambda)$) are the set of all functions that are negligible (resp. polynomial) in λ. Furthermore, we use standard Big O notation such as O, Ω, Θ, and ω. When working with groups, group elements are written upper case, and their exponents with respect to the group generator lower case. Lastly, log denotes the binary logarithm.

2.2 Algebraic Setting

To allow for a trapdoor, we require some algebraic structure: the group of quadratic residues with respect to RSA moduli. As a consequence, the memory-hardness of TdScrypt is based on factoring assumptions. In the following, we define modulus generation, related factoring assumptions, and the group of quadratic residues.

Definition 1 (Safe-prime Generator)**.** Let GenSP be an algorithm that samples two distinct safe primes of the same bit length uniformly at random. More precisely, define

$$(p', q') \leftarrow \mathsf{GenSP}(\lambda)$$

and let $p := (p' - 1)/2$ and $q := (q' - 1)/2$. It holds that p', p, q' and q are prime, $p \neq q$, and $\|p\| = \|q\| = k(\lambda)$ with $k(\lambda) \in \mathrm{poly}(\lambda)$.

In terms of notation, let $N := pq$ and $N' := p'q'$ in the rest of the paper.[7]. Next, we give two hardness assumptions stating that factoring N' as well as N is hard.

Definition 2 (Factoring N')**.** The game $\mathsf{Fac}'_{\mathsf{GenSP},\mathcal{A}}(\lambda)$ is defined with respect to GenSP and a *probabilistic polynomial-time* (PPT) adversary $\mathcal{A}$. On input of a security parameter λ it runs $(p', q') \leftarrow \mathsf{GenSP}(\lambda)$, sets $N' := p'q'$, and invokes the adversary, yielding $(p^*, q^*) \leftarrow \mathcal{A}(N')$. The game returns 1 if $\{p^*, q^*\} = \{p', q'\}$ and 0 otherwise.

We say that *factoring N' is hard* if, for every PPT algorithm $\mathcal{A}$, the advantage is negligible, i.e.,

$$\mathrm{Adv}^{\mathsf{Fac}'}_{\mathsf{GenSP},\mathcal{A}}(\lambda) := \Pr\Big[1 \leftarrow \mathsf{Fac}'_{\mathsf{GenSP},\mathcal{A}}(\lambda)\Big] \in \mathrm{negl}(\lambda)$$

where the probability is taken over the randomness of GenSP and $\mathcal{A}$.

Definition 3 (Factoring N)**.** The game $\mathsf{Fac}_{\mathsf{GenSP},\mathcal{A}}(\lambda)$ is defined similarly to $\mathsf{Fac}'_{\mathsf{GenSP},\mathcal{A}}(\lambda)$ but with N' replaced by N. So the game invokes $(p^*, q^*) \leftarrow \mathcal{A}(N)$ and returns 1 if and only if $\{p^*, q^*\} = \{p, q\}$. We say that *factoring N is hard* if, for every PPT algorithm $\mathcal{A}$,

$$\mathrm{Adv}^{\mathsf{Fac}}_{\mathsf{GenSP},\mathcal{A}}(\lambda) := \Pr[1 \leftarrow \mathsf{Fac}_{\mathsf{GenSP},\mathcal{A}}(\lambda)] \in \mathrm{negl}(\lambda)$$

where the probability is taken over the randomness of GenSP and $\mathcal{A}$.

Let us briefly analyze both assumptions. It is conjectured that the density of safe primes p' in the interval $[2^{k-1}, 2^k]$ is of order $1/k^2$ (e.g., [vzGS13]). Note that the map $p' \mapsto (p' - 1)/2$ is injective, so this statement implies the same regarding the density of $(p' - 1)/2 = p$ in $[2^{k-2}, 2^{k-1}]$. Thus, assuming the conjecture holds, the standard factoring assumption (i.e., factoring the product of two uniformly sampled arbitrary k-bit primes is hard) implies that factoring N' as well as N is hard. For simplicity, we say that *factoring is hard* with respect to GenSP if factoring N as well as factoring N' are hard with respect to GenSP.

Equipped with these definitions, we finally define the group of quadratic residues modulo N'.

[7]Note that usually N and N' are defined the other way around. However, most parts of the paper are only concerned with N, so we chose this notation to avoid clutter.

Definition 4 (Group of Quadratic Residues)**.** For $(p', q') \leftarrow \mathsf{GenSP}(\lambda)$, let $p := (p'-1)/2$, $q := (q'-1)/2$, $N' := p'q'$, and $N := pq$. The group of quadratic residues modulo N' is a subgroup of $\mathbb{Z}_{N'}^*$ defined as $\mathbb{QR}_{N'} := \{X^2 \mid X \in \mathbb{Z}_{N'}^*\}$.

Since p' and q' are safe primes (and thereby Blum integers), $\mathbb{QR}_{N'}$ is a cyclic group of order $|\mathbb{QR}_{N'}| = N$ and a uniformly sampled element $X \leftarrow \mathbb{QR}_{N'}$ is a generator with overwhelming probability $\varphi(N)/N = 1 - \frac{q+p-1}{N} \in (1 - \mathrm{negl}(\lambda))$ where φ denotes Euler's totient function.

2.3 Generic Group Model

In the proof of our main result we model the cyclic group $\mathbb{QR}_{N'} = \langle g \rangle$ of order $N := pq$ as a generic group in the style of Shoup's generic group model (GGM) [Sho97]. In the GGM algorithms do not get direct access to the group (that we may identify with the group $(\mathbb{Z}_N, +)$ by using the isomorphism $\mathbb{Z}_N \to \mathbb{QR}_{N'}$; $z \mapsto g^z$). Instead, access to group elements is provided via abstract labels $\sigma \in \mathcal{L}$ where $\mathcal{L} := \{0,1\}^{\omega_{\mathcal{L}}}$ (with $\omega_{\mathcal{L}} \geq \lceil \log(N) \rceil$) and group operations are performed using an oracle $\mathcal{G}$ that allows for multiplication, division, and inversion.

More precisely, let $\mathrm{Inj}(\mathbb{Z}_N, \mathcal{L})$ be the set of all injections from $\mathbb{Z}_N$ to $\mathcal{L}$ and let $\Sigma \overset{\$}{\leftarrow} \mathrm{Inj}(\mathbb{Z}_N, \mathcal{L})$ be one such injection chosen uniformly at random. Σ is called the *labeling function*, and it defines the oracle $\mathcal{G}\colon \{+, -, \mathsf{inv}\} \times \mathcal{L} \times \mathcal{L} \to \mathcal{L} \cup \{\perp\}$. $\mathcal{G}$ answers a query $(\circ, \sigma_1, \sigma_2)$ as follows. If σ_1 and σ_2 are not in the image of Σ, it returns $\perp$. Otherwise, if $\circ \in \{+, -\}$, it returns $\Sigma(\Sigma^{-1}(\sigma_1) \circ \Sigma^{-1}(\sigma_2) \bmod N)$, and if $\circ = \mathsf{inv}$, it returns $\Sigma(-\Sigma^{-1}(\sigma_1) \bmod N)$.

Abusing notation, we will almost always write Σ instead of $\mathcal{G}$ when talking about a concrete instantiation of the generic group oracle.

2.4 Machine Model and Complexity Measure

Parallel Oracle Model. To define a memory complexity measure, we first require a machine model. Prior work uses the *parallel random oracle model* [AS15, ACP+17] which allows algorithms to perform an unlimited number of random oracle queries in parallel. Similarly, we define the *parallel oracle model* which additionally allows for generic group queries. That is, a *polynomial-time* (PT) algorithm $\mathcal{A}$ has access to two oracles: $\mathcal{G}$, a group-operation oracle, and h, a random oracle. We will sometimes explicitly state the oracles in superscript, i.e., $\mathcal{A}^{\mathcal{G},\mathsf{h}}$, and omit them when they are clear from context.

The random oracle $\mathsf{h}\colon \{0,1\}^* \to \{0,1\}^{\omega_{\mathsf{h}}}$ maps inputs to bit strings of length ω_{h} and we require that $\omega_{\mathsf{h}} \in \Theta(\log N)$. To ensure that the set of random oracles is finite, we assume some sufficiently large, finite bound $*$ on the inputs.

Algorithm $\mathcal{A}$'s execution proceeds in *rounds* starting with round 1. Within each round, $\mathcal{A}$ performs local computation and submits oracle queries at the end of it. Then, at the beginning of the next round, $\mathcal{A}$ receives the response to its queries. Formally, *states* capture $\mathcal{A}$'s progress throughout the rounds and its queries to both oracles. At the end of round i, $\mathcal{A}$ produces an *output state* $\overline{\mathsf{st}}_i := (\tau_i, \mathbf{qrs}_i^{\mathcal{G}}, \mathbf{qrs}_i^{\mathsf{h}})$ where τ is a bit string, and $\mathbf{qrs}^{\mathcal{G}}$ and $\mathbf{qrs}^{\mathsf{h}}$ are vectors containing queries to $\mathcal{G}$ and h, respectively. Consequently, in round $i+1$, it receives the *input state* $\mathsf{st}_i := (\tau_i, \mathbf{res}_i^{\mathcal{G}}, \mathbf{res}_i^{\mathsf{h}})$ where $\mathbf{res}_i^{\mathcal{G}}$ and $\mathbf{res}_i^{\mathsf{h}}$ are vectors containing the results of the queries $\mathbf{qrs}_i^{\mathcal{G}}$ and $\mathbf{qrs}_i^{\mathsf{h}}$.

We will only consider *deterministic* algorithms in the interest of keeping proofs readable and concise. This is essentially without loss of generality in our setting since we only care about algorithms that correctly evaluate TdScrypt with sufficiently high, i.e., noticeable, success probability. Such algorithms can be derandomized by trying out a few choices of the randomness, checking each randomness on a polynomial number of random inputs, and then fix the randomness on which the algorithm performed best. This will with overwhelming probability result in a deterministic algorithm whose success probability is close to the randomized one.

Complexity Measure. To evaluate the memory complexity of an algorithm, we use a notion called *cumulative memory complexity* (CMC) [AS15]. Essentially, CMC is the sum of an algorithm's memory consumption at every point in time, i.e., the area under the memory usage curve.

More formally, consider an algorithm $\mathcal{A}$ running in the parallel oracle model on some input x. Its execution results in a sequence of input states $\mathsf{st}_i = (\tau_i, \mathbf{res}_i^{\mathcal{G}}, \mathbf{res}_i^{\mathsf{h}})$. Then, its CMC is given by

$$\mathsf{cc}_{\mathsf{mem}}(\mathcal{A}^{\mathcal{G},\mathsf{h}}(x)) := \sum_i \|\mathsf{st}_i\|$$

where the bit length of the input state st_i is defined as $\|\mathsf{st}_i\| := \|\tau_i\| + |\mathbf{res}_i^{\mathcal{G}}|\omega_{\mathcal{L}} + |\mathbf{res}_i^{\mathsf{h}}|\omega_{\mathsf{h}}$. Without loss of generality, an algorithm submits at least one query per round, so $\|\mathsf{st}_i\| \geq \min\{\omega_{\mathcal{L}}, \omega_{\mathsf{h}}\} \in \Theta(\log N)$ by definition.

Preprocessing Algorithms. Instead of working with a single deterministic PT algorithm $\mathcal{A}$, we often view it as a pair of deterministic PT algorithms $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$. Intuitively, $\mathcal{A}_0$ performs preprocessing and outputs some advice which $\mathcal{A}_1$ receives as input. It follows that we do not charge $\mathcal{A}_0$ for any memory usage and that we do not count its number of rounds (but emphasize that, unlike in other works, $\mathcal{A}_0$'s computation is not completely unbounded as we cannot allow it to factor N with noticeable probability).

In the terms of the parallel oracle model, given public parameters pp and online input x, to evaluate $\mathcal{A}^{\mathcal{G},\mathsf{h}}(\mathsf{pp}, x)$, first execute $\mathcal{A}_0^{\mathcal{G},\mathsf{h}}(\mathsf{pp})$ resulting in an input state st_0. Then, starting in round 1, run $\mathcal{A}_1^{\mathcal{G},\mathsf{h}}(\mathsf{st}_0, x)$ with the online input x yielding the output of $\mathcal{A}$. The CMC of $\mathcal{A}$ is defined as $\mathsf{cc}_{\mathsf{mem}}(\mathcal{A}^{\mathcal{G},\mathsf{h}}(\mathsf{pp}, x)) := \mathsf{cc}_{\mathsf{mem}}(\mathcal{A}_1^{\mathcal{G},\mathsf{h}}(\mathsf{st}_0, x))$.

3 A Trapdoor Memory-Hard Function from Factoring

In this section we define the syntax for trapdoor memory-hard functions (Section 3.1) and then describe TdScrypt, the instantiation of Diodon [BP17] we will use throughout the paper (Section 3.2).

3.1 Trapdoor Memory-Hard Functions

A *trapdoor memory-hard function* (TMHF) is defined by a triple of polynomial-time algorithms (Setup, Eval, TDEval) as follows (with the security parameter λ left implicit).

- $\mathsf{Setup}() \to (\mathsf{pp}, \mathsf{td})$. The probabilistic setup algorithm samples public parameters pp and corresponding trapdoor td. The public parameters implicitly determine the domain $\mathrm{Dom}(\mathsf{pp})$ and range $\mathrm{Ran}(\mathsf{pp})$ of the TMHF.

- $\mathsf{Eval}(\mathsf{pp}, w) =: y$. The deterministic evaluation algorithm takes public parameters pp and $w \in \mathrm{Dom}(\mathsf{pp})$ as inputs and returns a $y \in \mathrm{Ran}(\mathsf{pp})$.

- $\mathsf{TDEval}(\mathsf{pp}, \mathsf{td}, w) =: y$. The deterministic trapdoor evaluation algorithm takes public parameters pp, trapdoor td, and $w \in \mathrm{Dom}(\mathsf{pp})$ as inputs and returns $y \in \mathrm{Ran}(\mathsf{pp})$.

We require correctness, i.e., for all $(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{Setup}()$ and all $w \in \mathrm{Dom}(\mathsf{pp})$, it holds that

$$\mathsf{Eval}(\mathsf{pp}, w) = \mathsf{TDEval}(\mathsf{pp}, \mathsf{td}, w).$$

To quantify the quality of a TMHF we have to analyze the cumulative memory required to evaluate it with and without access to the trapdoor. Accordingly, if the required CMC is given by functions c_{hon} and c_{td}, i.e.,

$$c_{\text{hon}}(\mathsf{pp}) = \mathsf{cc}_{\mathsf{mem}}(\mathsf{Eval}(\mathsf{pp}, w)) \text{ and } c_{\text{td}}(\mathsf{pp}) = \mathsf{cc}_{\mathsf{mem}}(\mathsf{TDEval}(\mathsf{pp}, \mathsf{td}, w))$$

the TMHF is referred to as a $(c_{\text{hon}}, c_{\text{td}})$-TMHF.

Naturally, trapdoor evaluations must have a lower CMC than the standard evaluation algorithm. That is, for all $(\mathsf{pp}, \mathsf{td}) \leftarrow \mathsf{Setup}$, there exists some $0 < \Delta(\mathsf{pp}) < 1$ (ideally $\Delta(\mathsf{pp}) \in o(1)$), such that, for all inputs $w \in \mathrm{Dom}(\mathsf{pp})$, we have

$$c_{\text{td}}(\mathsf{pp}) < \Delta(\mathsf{pp}) \cdot c_{\text{hon}}(\mathsf{pp}).$$

Moreover, we want our function to be a good MHF when ignoring the trapdoor. This means the CMC of $\mathsf{Eval}(\mathsf{pp}, w)$ should be high by construction, but no adversarial evaluation algorithm should exist that can evaluate the function with much lower CMC. This must hold even when the other adversarial algorithm is allowed to make many parallel queries to the oracles and when it is given some advice that was computed (by any polynomial time preprocessing) dependent on pp (but of course not the input w or the trapdoor).

3.2 Description of TdScrypt

We will analyze TdScrypt which can be viewed as a concrete instantiation of Diodon [BP17]. Specifically, using notation from [BP17], we set $M := n$, $L := n$, and $\eta = 1$. Furthermore, TdScrypt is defined over the group of quadratic residues $\mathbb{QR}_{N'}$ instead of $\mathbb{Z}_{N'}$ due to technicalities.

Construction. TdScrypt is defined with respect to integer $n \in \mathbb{N}$ corresponding to the number of iterated steps (i.e., repeated squarings). It relies on computations in the group $\mathbb{QR}_{N'}$ of quadratic residues modulo N' as defined in Definition 4. Accordingly, its public parameters pp consist of an RSA modulus $N' := p'q'$, where p' and q' are safe primes generated using safe prime generator GenSP (see Definition 1), a hash function $\mathsf{h}\colon \{0,1\}^* \to \{0,1\}^{\omega_{\mathsf{h}}}$, and n. The corresponding trapdoor is $\mathsf{td} := N = (p'-1)(q'-1)/4 = |\mathbb{QR}_{N'}|$. TdScrypt's formal description is in Figure 1.

```
Setup()
00 $(p', q') \leftarrow \mathsf{GenSP}()$
01 $N' := p'q'$
02 $N := (p'-1)(q'-1)/4$
03 pick $\mathsf{h}$
04 $\mathsf{pp} := (N', \mathsf{h}, n)$
05 $\mathsf{td} := N$
06 Return $(\mathsf{pp}, \mathsf{td})$

Eval(pp, W)
07 $W_0 := W$
08 For $i := 1, \ldots, n-1$:
09     $W_i := W_{i-1}^2$
10     Store $W_i$
11 $S_0 := \mathsf{h}(W_n, 0^{\omega_\mathsf{h}})$
12 For $i := 1, \ldots, n$:
13     $j_i := S_{i-1} \bmod n$
14     $S_i := \mathsf{h}(W_{j_i}, S_{i-1})$
15 Return $S_n$

TDEval(pp, td, W)
16 $W_0 := W$
17 $m := 2^n \bmod N$
18 $W_n := W_0^m$
19 $S_0 := \mathsf{h}(W_n, 0^{\omega_\mathsf{h}})$
20 For $i := 1, \ldots, n$:
21     $j_i := S_{i-1} \bmod n$
22     $m_i := 2^{j_i} \bmod N$
23     $W_{j_i} := W_0^{m_i}$
24     $S_i := \mathsf{h}(W_{j_i}, S_{i-1})$
25 Return $S_n$
```

Figure 1: Trapdoor memory-hard function $\text{TdScrypt}_n^{\mathsf{h}}$ defined with respect to number of steps $n \in \mathbb{N}$. The TMHF uses a hash function $\mathsf{h}\colon \{0,1\}^* \to \{0,1\}^{\omega_\mathsf{h}}$, domain $\mathrm{Dom}(\mathsf{pp}) = \mathbb{QR}_{N'}$ and range $\{0,1\}^{\omega_\mathsf{h}}$. The exponentiations W_i^2, W_0^m, and $W_0^{m_i}$ are computed in $\mathbb{QR}_{N'}$.

In the following, we will assume that the time of evaluating h approximately matches the time of evaluating a group operation, i.e., computing a multiplication in modulo N'. This could for example be implemented by setting $\mathsf{h}(s) := \mathsf{h}''(\mathsf{h}'(s) \cdot \mathsf{h}'(s) \bmod N')$, where $\mathsf{h}'\colon \{0,1\}^* \to \mathbb{QR}_{N'}$ and $\mathsf{h}''\colon \mathbb{QR}_{N'} \to \{0,1\}^{\omega_\mathsf{h}}$ are cryptographic hash functions.

To evaluate the TMHF on input $W \in \mathrm{Dom}(\mathsf{pp}) := \mathbb{QR}_{N'}$ algorithm Eval sets $W_0 := W$ and computes and stores the group elements $W_i := W_{i-1}^2 = W^{2^i}$ for $i \in [n-1]$. Then it sets $S_0 := \mathsf{h}(W_n, 0^{\omega_\mathsf{h}})$, and, for $i := [n]$, computes $S_i := \mathsf{h}(W_{j_i}, S_{i-1})$ where the index $j_i := S_{i-1} \bmod n$. The output of Eval is $S_n \in \mathrm{Ran}(\mathsf{pp}) := \{0,1\}^{\omega_\mathsf{h}}$.

TDEval computes the values S_i without storing the group elements W_{j_i}, but instead recomputes them efficiently using its knowledge of $N = |\mathbb{QR}_{N'}|$. It sets $W_0 := W$, computes $W_n := W_0^m$, and $S_0 := \mathsf{h}(W_n, 0^{\omega_\mathsf{h}})$, where $m := 2^n \bmod N$. Then for $i \in [n]$ it computes, in order, $j_i := S_{i-1} \bmod n$, $m_i := 2^{j_i} \bmod N$, $W_i = W_0^{m_i}$, and $S_i := \mathsf{h}(W_{j_i}, S_{i-1})$. Its output is S_n.

Cumulative Memory Complexity. Recall that the CMC is the memory usage summed over all steps. In the following analysis of TdScrypt we will give a brief estimate of Eval and TDEval's CMC where we define a "step" to constitute the time taken by one group operation. For details on this choice and an in-depth CMC analysis we refer to Appendix A.

For Eval, note that its CMC is dominated by the second loop (Lines 12–14). During the loop, Eval keeps n group elements $W_0, \ldots, W_n$ in memory, amounting to $\Theta(n \log(N'))$ bits. The loop requires $\Theta(n)$ group operations in total as every evaluation of h takes the time of one group operation by definition. We thus obtain

$$\mathsf{cc}_{\mathsf{mem}}(\mathsf{Eval}(\mathsf{pp}, x)) \in \Theta(n \log(N') \cdot n) = \Theta(n^2 \log(N')).$$

TDEval's CMC is also dominated by the loop (Lines 20–24). In contrast to Eval, the memory usage during the loop is low because it does not store n group elements. However, every loop iteration requires more steps since TDEval reduces n-bit integers of the form 2^j

with $j \in [0, n)$ modulo N (Line 22) and performs exponentiations in $\mathbb{QR}_{N'}$ (Line 23). In general, the former operations requires $\Theta(n \log(N'))$ group operations which can be reduced to $\log(n)$ group operations by using a lookup table of size $\Theta(\log(n) \log(N'))$. The exponentation requires $\Theta(\log N')$ group operations using square-and-multiply. So during the loop, TDEval stores the lookup table using $\Theta(\log(n) \log(N'))$ bits, and every iteration takes $\Theta(\log(n) + \log(N'))$ group operations, resulting in

$$\begin{aligned}\mathsf{cc}_{\mathsf{mem}}(\mathsf{TDEval}(\mathsf{pp}, \mathsf{td}, x)) &\in \Theta(\log(n) \log(N') \cdot n \left(\log(N') + \log(n)\right)) \\ &= \Theta(n \log(N')^2 \log(n)).\end{aligned}$$

Summing up, TdScrypt is a $(n^2 \log(N'), n \log(n) \log(N')^2)$-TMHF.

4 Overview of the Lower Bound Proof

In the previous section, we established that Eval has a CMC of $\Theta(n^2 \log N)$ and TDEval one of $\Theta(n \log n \log N)$. These analyses only hold for the specific algorithms stated in Figure 1. However, an adversary (without knowledge of the trapdoor) need not follow Eval. The rest of this paper is devoted to showing that no evaluation algorithm is meaningfully faster than Eval.

Theorem 1 (Restated)**.** *Let $n \in \mathrm{poly}(\lambda)$ with $n \geq 8$ and let $\mathcal{A}$ be a deterministic parallel oracle machine that evaluates* TdScrypt *correctly with probability $\chi(\lambda)$ over the choice of the parameters and input W. Then, assuming that factoring is hard, in the GGM and ROM with probability at least $\chi(\lambda) - \epsilon(\lambda)$,*

$$\mathsf{cc}_{\mathsf{mem}}(\mathcal{A}^{\mathsf{pp}}(W)) \in \Omega\left(\frac{n^2}{\log n} \log N\right)$$

where pp are the public parameters, $\epsilon(\lambda) \in \mathrm{negl}(\lambda)$, and the probability is taken over the choice of parameters.

So we show a lower bound on the CMC of evaluating TdScrypt that almost matches Eval and is only a factor of $1/\log n$ loose. Note that $\log n \in O(\log \lambda)$ by definition of n, so the loss in tightness is small asymptotically.

To prove our lower bound, we assume that factoring is hard and work in generic models. This is in line with lower bounds on the memory complexity of (non-trapdoor) memory-hard functions which are situated in the ROM. The rest of the paper is divided into two sections similar to Alwen et al. [ACP+17]

Single-challenge Time-Memory Trade-Off (Section 5). Before considering the whole TdScrypt execution, we focus on the hardness of computing $W_j := W^{2^j}$ for $j \xleftarrow{\$} [n]$ given M bits of precomputed advice. This is closely related to the second phase of TdScrypt which, for all $i \in [n]$, requires computing $S_i := \mathsf{h}(W^{2^{j_i}}, S_{i-1})$ given challenge $j_i := S_{i-1} \bmod n$. Note that j_i is chosen (almost) uniformly from $[0, n)$ when modeling h as a random oracle. Here, the precomputed advice can be thought of as the state of the algorithm before learning challenge j_i. Our goal is to show that if the advice is short, computing W_j takes a long time on average. Stated differently, if $\mathcal{A}$ computes W_j quickly on average, the advice must be large.

Multi-challenge Memory Complexity Lower-Bound (Section 6). We abstract the whole evaluation of TDSCRYPT as a multi-challenge game to get the lower bound on $\mathsf{cc}_{\mathsf{mem}}(\mathcal{A})$. To this end, the single-challenge trade-off is applied to every challenge, i.e., to the moment in time before it is known *and every point in time before the preceding challenge has been issued.* This idea closely follows the corresponding proof in Alwen et al. [ACP+17], and we refer interested readers to the paper for details.

5 Single-challenge Time-Memory Trade-Off

We consider a pair of deterministic parallel oracle machines $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ where $\mathcal{A}_0$ receives the input W and performs preprocessing, resulting in an advice string. In the context of TDSCRYPT, this is best thought of as the computation performed by the (adversarial) evaluation algorithm up to learning challenge j. $\mathcal{A}_1$, on input of challenge $j \overset{\$}{\leftarrow} [0, n)$, uses the advice string to query W^{2^j} to h in as few rounds as possible. For now, we explicitly pass the challenge j as an input. Our goal is to relate the advice string size to the number of rounds required by $\mathcal{A}_1$ on average across all possible choices of j.

To formalize the preceding description, we first specify the set of parameters which determine an execution of TDSCRYPT and then the game sketched above.

Definition 5 (Parameters)**.** For security parameter $\lambda \in \mathbb{N}$, let $\mathsf{params}(\lambda)$ be the set of all possible parameters. It contains all quadruples $(N, \Sigma, \mathsf{h}, w)$ where $(p', q') \leftarrow \mathsf{GenSP}(\lambda)$ with $p := (p' - 1)/2$ and $q := (q' - 1)/2$ defines the group order $N := pq$, the labeling function Σ is an injection $\mathrm{Inj}(\mathbb{Z}_N, \mathcal{L})$, h is a random oracle, and the input w is non-zero, i.e., $w \in \mathbb{Z}_N \setminus \{0\}$. Here, w is the discrete logarithm of W, i.e., $\Sigma^{-1}(W) = w$ in the GGM. Furthermore, recall that we require $\omega_{\mathcal{L}}$, the length of labels in $\mathcal{L}$, and ω_{h}, the output length of h, to be larger than $\log N$ and of order $\log N$ respectively.

To avoid clutter, we sometimes leave λ implicit and write params instead of $\mathsf{params}(\lambda)$.

Definition 6 (Single-challenge Game)**.** Let $(N, \Sigma, \mathsf{h}, w) \in \mathsf{params}$ and $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ be deterministic parallel oracle machines . To the challenge $j \in [0, n)$ and state $\mathsf{st}_0 := \mathcal{A}_0^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$ we associate

$$\mathsf{TimeSC}_{\mathcal{A}_1}^{\Sigma,\mathsf{h}}(\mathsf{st}_0, j) = \begin{cases} \min\{i \colon \Sigma(w^{2^j}) \in \mathbf{qrs}_i^{\mathsf{h}}\}, & \text{if the minimum exists} \\ \infty, & \text{otherwise.} \end{cases}$$

So TimeSC is the earliest round $i \in \mathbb{N}^+$ in which $\mathcal{A}_1^{\mathcal{G},\mathsf{h}}(\mathsf{st}_0, j)$ queries $\Sigma(w^{2^j})$ to h or ∞ if it never does.

Next, we answer the following question: *How small can* TimeSC *be relative to the size of* st_0*?* That is, we are interested in a time-memory trade-off when playing TimeSC.

A preliminary observation is that if $\|\mathsf{st}_0\| \geq n\omega_{\mathcal{L}}$, then there exists a simple strategy $\mathcal{SS}$ achieving $\mathsf{TimeSC}_{\mathcal{SS}} = 1$. Indeed, $\mathcal{SS}$ simply stores the answers to every possible challenge in st_0. Generalizing to any $\|\mathsf{st}_0\| = M$, we get the following strategy $\mathcal{SS}$: $\mathcal{SS}_0$ encodes $\rho := \lfloor M/\omega_{\mathcal{L}} \rfloor$ group elements in st_0 where the elements are of the form W^{2^i} with the i equidistantly spaced across $0, \ldots, n-1$. $\mathcal{SS}_1$, on input j, picks the maximum $i \leq j$ such that W^{2^i} is stored in st_0 and, if necessary, repeatedly squares it until reaching W^{2^j}. So $\mathsf{TimeSC}_{\mathcal{SS}} > n/(2\rho)$ for at least half of the challenges.

We want to show that if factoring is hard, no algorithm $\mathcal{A}$ can meaningfully beat $\mathcal{SS}$ for most choices of parameters. That is, there exists a partition of params into sets good and bad such that bad contains a negligible fraction of params and $\mathcal{A}$ can only beat the strategy when the chosen parameters are in bad. Note that we cannot give guarantees for all possible parameter choices since, e.g., sometimes N might be easy to factor. This leads us to Lemma 1 that characterizes the set bad.

Lemma 1 (Single-challenge Trade-Off)**.** *For every pair of deterministic parallel oracle machines $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, all $c \in \mathbb{N}$ with security parameter λ large enough, and all $n, M, Q \in \mathrm{poly}(\lambda)$ with $n \geq 8$ and subsets $\mathsf{bad} \subseteq \mathsf{params}(\lambda)$, if, for every $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$, $\mathcal{A}_1$ makes at most Q queries, $\mathsf{st}_0 := \mathcal{A}_0^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$ is of size $|\mathsf{st}_0| \leq M$ and*

$$\Pr_{j \xleftarrow{\$} [0,n)}\left[\mathsf{TimeSC}_{\mathcal{A}_1}^{\Sigma,\mathsf{h}}(\mathsf{st}_0, j) \leq \frac{n}{6\rho \log(n/2)}\right] \geq 1/2$$

where $\rho = (M + \log n + \log Q + c \log \lambda + 1)/(\log(N) - 3(\log Q - \log n) - 3)$, then $|\mathsf{bad}| < \lambda^{-c}|\,\mathsf{params}(\lambda)|$.

In other words, bad is the set of parameters where, for every choice of parameters contained therein, $\mathcal{A}_1$ answers more quickly than expected. It is also possible to define the single-challenge trade-off the other way around. So there exists a large set good, and the trade-off holds for any parameters in good. Corollary 1 captures this view and also simplifies Lemma 1 by not stating all constants explicitly.

Corollary 1. *For every pair of deterministic parallel oracle machines $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, there exists a negligible function $\epsilon(\lambda) \in \mathrm{negl}(\lambda)$ such that, for all $n, M, Q \in \mathrm{poly}(\lambda)$ with $n \geq 8$, there exists a subset $\mathsf{good} \subseteq \mathsf{params}(\lambda)$ such that, for every $(N, \Sigma, \mathsf{h}, w) \in \mathsf{good}$, $\mathcal{A}_1$ makes at most Q queries, $\mathsf{st}_0 := \mathcal{A}_0^{\Sigma,\mathsf{h}}(\Sigma(w))$ is of size $|\mathsf{st}_0| \leq M$,*

$$\Pr_{j \xleftarrow{\$} [0,n)}\left[\mathsf{TimeSC}_{\mathcal{A}_1}^{\Sigma,\mathsf{h}}(\mathsf{st}_0, j) > \frac{n}{6\rho \log(n/2)}\right] \geq 1/2$$

where $\rho \in \Theta((M + \log n + \log Q + \log \lambda)/(\log(N) - (\log Q - \log n)))$ and $|\mathsf{good}| \geq (1 - \epsilon(\lambda))|\mathsf{params}(\lambda)|$.

Let us now compare Corollary 1's guarantees to the simple strategy $\mathcal{SS}$. ρ is asymptotically close to $\lfloor M/\log N \rfloor \leq \lfloor M/\log \omega_{\mathcal{L}} \rfloor$ as in $\mathcal{SS}$. However, $n/(6\rho \log(n/2))$ is roughly a $1/\log(n)$ factor looser than $n/(2\rho)$. This seemingly is an artifact of our proof (looking ahead, Lemma 2), and we leave possible improvements to future work. The remainder of the section is devoted to the proof and structured as follows: First, we explain how to reason about $\mathcal{A}_1$'s query behavior algebraically and state some algebraic tools related to that. Second, we give the high-level structure of the proof and highlight three central claims. Finally, we prove each claim to complete the proof.

5.1 Reasoning About $\mathcal{A}_1$'s Queries Algebraically

Algebraic Representation of Labels. Algorithm $\mathcal{A}_1$ receives st_0 (the output of preprocessing algorithm $\mathcal{A}_0(\Sigma(1), \Sigma(w))$) and challenge j as input. It then uses the GGM

Table 1: Associating labels in the GGM to linear representations over $\mathbb{Z}_N$. Column "Query" lists the queries made by $\mathcal{A}_1$ and column "Response" the label returned as response to the query. Labels queried *out of the blue*, i.e., not previously returned to $\mathcal{A}_1$ as answer to a query, are collected in column "O.o.t.B. Inputs". The representations assigned to responses and o.o.t.b. input labels are given in column "Representation".

Round	Query	O.o.t.B. Inputs	Response	Representation
1	$\mathcal{G}(+, \sigma_1, \sigma_2)$			
		σ_1		x_1
		σ_2		x_2
			σ_3	$x_1 + x_2$
2	$\mathcal{G}(+, \sigma_3, \sigma_3)$			
			σ_4	$2x_1 + 2x_2$
3	$\mathcal{G}(-, \sigma_3, \sigma_5)$			
		σ_5		x_3
			σ_6	$x_1 + x_2 - x_3$
		$\vdots$		

oracle to generate the label $\Sigma(w2^j)$. As is typical in the GGM (e.g., [CK18, Sho97]), the reduction in our proof exploits $\mathcal{A}_1$'s query behavior. To this end, the reduction associates an algebraic representation to every label that $\mathcal{A}_1$ queries to or receives from the oracle. More precisely, the algebraic representations are linear terms over $\mathbb{Z}_N$ in several indeterminates $x_1, \ldots, x_m$. The reduction stores the mapping of algebraic representations to labels in an (initially empty) table T to ensure consistency between queries.

Whenever $\mathcal{A}_1$ makes a query $\mathcal{G}(+, \sigma_1, \sigma_2)$, the reduction does the following:[8] First, it checks whether T already contains the label σ_1. If not, it represents σ_1 by a new indeterminate x_i and stores this mapping in T. Then, σ_2 is processed analogously before the reduction computes the label $\sigma_3 = \mathcal{G}(+, \sigma_1, \sigma_2)$. If T does not contain a representation for σ_3, its representation is set to the sum of the ones of σ_1 and σ_2. Queries for the operations $\circ \in \{-, \mathsf{inv}\}$ are handled in the analogous way. Table 1 illustrates this explanation by example.

We stress that $\mathcal{A}_1$ only receives the (bit string) st_0 and the challenge j as input but not any labels. As a consequence, T is initially empty. Thus, all algebraic representations will not contain any constant terms. This is in contrast to similar approaches (e.g., [Sho97]) where constant terms arise as a consequence of the adversary explicitly receiving the group generator as input at the beginning of the game. Intuitively, in our scenario $\mathcal{A}_0$ must store some labels (e.g., $\Sigma(w)$) in st_0 in order for the hint to be useful.

System of Equations. In the above explanation, the reduction executed $\mathcal{A}_1$ with some arbitrary challenge $j \in [n]$. In the actual proof, the reduction cares about specific challenges, in particular, the subset $J \subseteq [0, n)$ of challenges that are answered within in at most t rounds (t will be set later). For every $j \in J$, the reduction starts $\mathcal{A}_1(\mathsf{st}_0, j)$. Then,

[8]The following explanation is high-level and thus omits technicalities such as collision handling.

it executes all instances of $\mathcal{A}_1$ in parallel in lockstep while only keeping track of a *single* table T. In more detail: First, the reduction initializes the empty table T. Second, for every $j \in J$, it runs the $\mathcal{A}_1(\mathsf{st}_0, j)$ until the first batch of parallel queries to the GGM oracle is made. Iterating over all $j \in J$, the reduction responds to the queries and adds labels to T as described above. Then, it resumes the execution of all $\mathcal{A}_1$ instances until they query the GGM a second time. The reduction repeats this procedure for t rounds.

This parallel and lockstep execution gives rise to a system of linear equations corresponding to the challenges $j \in J$ as follows. By definition of J, all challenges j lead to $\mathcal{A}_1(\mathsf{st}_0, j)$ querying the label $\Sigma(w2^j)$ within t rounds. In turn, the table must contain this label with a corresponding representation of the form $a_{j,1}x_1 + \cdots + a_{j,m}x_m$ (here in, say, m indeterminates). By correctness, it must satisfy

$$a_{j,1}x_1 + \cdots + a_{j,m}x_m = w2^j \bmod N. \tag{1}$$

Collecting these equations for all $j \in J$, we obtain a system of equations over $\mathbb{Z}_N$ of the form $A\boldsymbol{x} = \boldsymbol{b}$, where $A = (a_{ji})_{j\in J, i\in[m]}$ and $\boldsymbol{b} = (2^j w \bmod N \mid j \in J)^\top$. Since Equation (1) holds for every equation, the system must have a solution $\boldsymbol{x} \in \mathbb{Z}_m$. Intuitively, the solution $\boldsymbol{x}$ corresponds to the discrete logarithm of group elements queried by $\mathcal{A}_1$ out of the blue (represented by indeterminates) of which there are m in total. In practice, $\mathcal{A}_1$ must have (mostly) stored these group elements in st_0 as a randomly guessed group element will most likely not be useful in computing a challenge.

Groups of Unknown Order. The above characterization of A requires knowledge of the group order N. However, assuming that factoring is hard, the reduction and, more importantly, $\mathcal{A}_1$ do not know N. Therefore, we will argue that it is almost always valid to consider the system $A\boldsymbol{x} = \boldsymbol{b}$ as being defined over $\mathbb{Z}$ instead of $\mathbb{Z}_N$. That is, $A \in \mathbb{Z}^{\ell\times m}$, $\boldsymbol{x} \in \mathbb{Z}^m$, and $\boldsymbol{b} \in \mathbb{Z}^\ell$ with $\boldsymbol{b} = (2^j w \mid j \in J)^\top$ Since $\mathcal{A}_1$ is executed for t rounds, for every entry $a_{ji} \in A$, we have $|a_{ji}| \leq 2^t$ as a consequence.[9] Furthermore, $A\boldsymbol{x} = \boldsymbol{b}$ constitutes a system of Diophantine equations since A, $\boldsymbol{x}$, and $\boldsymbol{b}$ only have integer components.[10] Since we are working with integers, this system might not have a solution $\boldsymbol{x}$. However, looking ahead, it almost always will, assuming that factoring is hard.

Solving a system of equations $A\boldsymbol{x} = \boldsymbol{b}$ over $\mathbb{Z}$ works similarly to solving one over a field. Recall that in a field one commonly uses Gaussian elimination to transform A into a row- or column-reduced echelon form (i.e., a triangular matrix with pivot entries equal to 1) because then the solution can easily be found algorithmically. The integer analogue to the column reduced echelon form is the well-known column-style *Hermite normal form (HNF)* which we define as in [MW01].

Definition 7 (Hermite Normal Form)**.** A matrix $H \in \mathbb{Z}^{\ell\times m}$ is in Hermite normal form if

(i) there exists a sequence of integers $1 \leq i_1 < \cdots < i_r \leq \ell$ such that $h_{ij} = 0$ for all $i < i_j$ where r is the number of non-zero columns; and

(ii) $0 \leq h_{i_j k} < h_{i_j j}$ for all $0 \leq k < j \leq m$.

[9]In fact, for every row $\boldsymbol{a}_j$, one can even show that $\|\boldsymbol{a}_j\|_1 \leq 2^t$ (cf. Appendix B), but the weaker statement $\|\boldsymbol{a}_j\|_\infty \leq 2^t$ is sufficient for our proof.

[10]A precise definition of A, $\boldsymbol{x}$, and $\boldsymbol{b}$ appears in Appendix C.2 as part of the proof contained therein.

So by Item (i), H is a lower-triangular matrix and all its zero columns are to the right. Furthermore, taking Item (ii) into account, for every matrix A, there exists a unique HNF H such that $H = AU$ where $U \in \mathbb{Z}^{m\times m}$ is unimodular (i.e., invertible over $\mathbb{Z}$). Moreover, H has $r := \operatorname{rank}(A)$ non-zero columns [Her09]. In the rest of the paper, we denote the HNF of A by $\mathsf{HNF}(A) = (H, U)$.

Note that the definition of $\operatorname{rank}(\cdot)$ might differ to one's own intuitive definition. This is due to $\mathbb{Z}^{\ell\times m}$ being a module and not a vector space. In contrast to linear algebra, different definitions of rank are not equivalent, so we define $\operatorname{rank}(\cdot)$ below.

Definition 8 (Rank)**.** $\operatorname{rank}(A)$ is the size of largest subset of A's columns that are linearly independent. A set of columns $\{\boldsymbol{c}_1, \dots, \boldsymbol{c}_k\}$ is linearly dependent if there exist $\alpha_1, \dots, \alpha_k \in \mathbb{Z}$ with at least one $\alpha_i \neq 0$ such that $\sum_{i=1}^{k} \alpha_i \boldsymbol{c}_i = \boldsymbol{0}$.

5.2 Proof Skeleton

We first give a high-level overview of the proof. Assume that Lemma 1 does not hold. So we have a large set of parameters bad where the trade-off does not hold for at least half of the challenges. That is, we have an upper bound on how many rounds the adversary needs to answer them. Thus, we can derive a system of equations $A\boldsymbol{x} = \boldsymbol{b}$ over $\mathbb{Z}$ where the $a_{ij} \in A$ are bounded in magnitude due to the trade-off not holding. Then, we prove the following claims about this system:

- Assuming factoring is hard, $A\boldsymbol{x} = \boldsymbol{b}$ behaves identically over $\mathbb{Z}$ and $\mathbb{Z}_N$ almost always. For example, $A\boldsymbol{x} = \boldsymbol{b}$ having a solution over $\mathbb{Z}_N$ does not necessarily imply the same holds over $\mathbb{Z}$, yet the implication will almost always hold under the factoring assumption.

- Assuming that $A\boldsymbol{x} = \boldsymbol{b}$ over $\mathbb{Z}$ has a solution, $\operatorname{rank}(A) \geq \rho$ for every choice of parameters in bad.

- $\operatorname{rank}(A) < \rho$ for some choice of parameters in bad. Otherwise, a random labeling function Σ could be compressed more efficiently than information-theoretically possible.

Notice that the last two claims contradict each other, so Lemma 1 must hold. We now give the detailed version of the above sketch.

Proof of Lemma 1. We distinguish two cases depending on ρ, the first one being trivial.

Case $\rho > n/(6\log(n/2))$: Then $n/(6\rho\log(n/2)) < 1$ and so the trade-off holds for any $j \in [0, n)$ since $\mathsf{TimeSC} \geq 1$ by Definition 6.

Case $\rho \leq n/(6\log(n/2))$: Towards contradiction, assume that Lemma 1 does not hold. That is, there exists a pair of deterministic parallel oracle machines $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, $c \in \mathbb{N}$ with security parameter λ large enough and, for infinitely many security parameters λ, there exist $n, M, Q \in \operatorname{poly}(\lambda)$ with $n \geq 8$ and a subset $\mathsf{bad} \subseteq \mathsf{params}(\lambda)$, such that, for every $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$, $\mathcal{A}_1$ makes at most Q queries, $\mathsf{st}_0 := \mathcal{A}_0^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$ is of size $|\mathsf{st}_0| \leq M$ and

$$\Pr_{j \xleftarrow{\$} [0,n)} \left[\mathsf{TimeSC}_{\mathcal{A}_1}^{\Sigma,\mathsf{h}}(\mathsf{st}_0, j) \leq \frac{n}{6\rho\log(n/2)} \right] \geq 1/2$$

where $\rho = (M + \log n + \log Q + c \log \lambda + 1)/(\log(N) - 3(\log Q - \log n) - 3)$ and $|\mathsf{bad}| \geq \lambda^{-c} |\mathsf{params}(\lambda)|$.

So for parameters in bad, it holds that $\mathcal{A}_1$ will answer at least $n/2$ challenges correctly by the end of round $\frac{n}{6\rho \log(n/2)}$. By the definition of rounds (cf. Section 2.4), to compute these answers, it can make and receive the results of $t := \frac{n}{6\rho \log(n/2)} - 1$ rounds of queries.

As in Section 5.1, consider the system of Diophantine equations $A\boldsymbol{x} = \boldsymbol{b}$ where $A \in \mathbb{Z}^{(n/2)\times m}$ for some $m \in \mathbb{N}^+$ arising from $\mathcal{A}_1$'s query behavior. In the case that $\mathcal{A}_1$ answers more than $n/2$ challenges, we avoid ambiguities by selecting $n/2$ challenges in some deterministic order (e.g., low challenge numbers first). Note that every matrix element $|a_{ij}| \leq 2^t$ and that the vector $\boldsymbol{b}$ does not contain 0, its elements are pairwise distinct, and of the form $b_i = 2^j w$ for $j \in [0, n)$.

First, assuming factoring is hard, $A\boldsymbol{x} = \boldsymbol{b}$ behaves similarly to $A\boldsymbol{x} = \boldsymbol{b} \bmod N$ almost always. It is proven in Section 5.3.

Claim 1. *If Lemma 1 does not hold, there exists a negligible function* $\epsilon(\lambda) \in \mathrm{negl}(\lambda)$ *such that*

(i) $A\boldsymbol{x} = \boldsymbol{b}$ *has a solution;*

(ii) $\mathsf{HNF}(A)$ *and* $\mathsf{HNF}(A) \bmod N$ *have the same shape (i.e.,* $\mathsf{HNF}(A) \bmod N$ *has no additional zero entries); and*

(iii) the diagonal entries of $\mathsf{HNF}(A) \bmod N$ *are invertible over* $\mathbb{Z}_N$

except for probability $\epsilon(\lambda)$ *where the probability is taken over the choice of* $(N, \Sigma, \mathsf{h}, w) \in \mathsf{params}(\lambda)$.

Second, since $\mathcal{A}$ answers a lot of challenges quickly, its rank must be large as proven in Section 5.4.

Claim 2. *If Lemma 1 does not hold and* $\rho \leq n/(6 \log n)$*, for all* $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$*,* $\mathrm{rank}(A) \geq \rho$*.*

Last, $\mathrm{rank}(A)$ cannot be too large by an incompressibility argument presented in Section 5.5.

Claim 3. *If Lemma 1 does not hold, there exists a* $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$*, such that* $\mathrm{rank}(A) < \rho$*.*

The latter two claims contradict each other, completing the proof. □

5.3 Analyzing the Behavior of $A\boldsymbol{x} = \boldsymbol{b}$

The proof of Claim 1 follows from the assumed hardness of factoring. In particular, Item (i) and Item (ii) reduce to factoring N', and Item (iii) to factoring N.

Proof of Claim 1. Let E_i denote the event that item i does not hold. We will show that all events happen with negligible probability.

- There exists a negligible function $\epsilon(\lambda) \in \mathrm{negl}(\lambda)$ such that $\Pr_{\mathsf{params}}\left[E_{(\mathrm{iii})}\right] \leq \epsilon(\lambda)$. To see this, consider the algorithm $\mathcal{B}$ playing the game $\mathsf{Fac}_{\mathsf{GenSP},\mathcal{B}}(\lambda)$ (cf. Definition 3). It uses $\mathcal{A}$ (guaranteed by the assumption that Lemma 1 does not hold) as a subroutine.

  On input N, $\mathcal{B}$ picks $w \leftarrow \mathbb{Z}_N \setminus \{0\}$, and lazily samples h and Σ to respond to $\mathcal{A}$'s queries. Note that $\mathcal{B}$ can simulate Σ and h perfectly because it knows N. Assuming that N, w and what has been sampled of Σ and h so far is consistent with bad, $\mathcal{B}$ is able to derive the system $A\boldsymbol{x} = \boldsymbol{b}$. Then, it computes $\mathsf{HNF}(A) \bmod N$ and checks whether any diagonal element a cannot be inverted. If so, $\mathcal{B}$ outputs (p^*, q^*) where $p^* = \gcd(a, N)$ and $q^* = N/p^*$.

  $\mathcal{B}$ is a PPT algorithm and $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$ with probability at least λ^{-c}. Hence,

  $$\Pr\left[E_{(\mathrm{iii})}\right] \leq \lambda^c \, \mathrm{Adv}^{\mathsf{Fac}}_{\mathsf{GenSP},\mathcal{B}}(\lambda)$$

  which is negligible as factoring N is assumed to be hard.

- There exists a negligible function $\epsilon(\lambda) \in \mathrm{negl}(\lambda)$ such that $\Pr_{\mathsf{params}}\left[E_{(\mathrm{i})}\right] + \Pr_{\mathsf{params}}\left[E_{(\mathrm{ii})}\right] \leq \epsilon(\lambda)$.

  The reduction is similar to the previous item except for two differences. First, $\mathcal{B}$ only knows N' but not N, so it cannot simulate Σ perfectly. However, this is not an issue as $\mathcal{B}$ can approximate lazy sampling Σ by using $N'/4$ instead of N. Since $N'/4 = N + O(\sqrt{N})$, answering a polynomial number of Σ queries using lazy sampling with $N'/4$ is statistically close to lazy sampling with N.

  Second, $\mathcal{B}$ factors N' instead of N. If $A\boldsymbol{x} = \boldsymbol{b}$ has a solution over $\mathbb{Z}$ but not $\mathbb{Z}_N$ or if a matrix entry is non-zero over $\mathbb{Z}$ but zero over $\mathbb{Z}_N$, $\mathcal{B}$ learns a multiple of the group order N. Such a multiple can be used to factor N' with probability at least $1/2$ [KL14, Thm. 8.50].

Combining the two items above completes the proof. □

5.4 Combinatorial Proof of the $\mathrm{rank}(A)$ Lower Bound

The key result of this section is the following number-theoretical lemma. Given a system of Diophantine equations $A\boldsymbol{x} = \boldsymbol{b}$, it bounds $\mathrm{rank}(A)$ from below as a function of A's number of rows and the magnitude of its entries.

Lemma 2. *Let $\ell \in \mathbb{N}$ with $\ell \geq 4$, $t \in \mathbb{N}$, and $m, w \in \mathbb{N}^+$. For any matrix $A = (a_{i,j})_{i\in[\ell],j\in[m]} \in \mathbb{Z}^{\ell\times m}$ with $|a_{i,j}| \leq 2^t$ for all i, j and vector $\boldsymbol{b} = (2^{j_1}w, \ldots, 2^{j_\ell}w)^\top$ with pairwise distinct $j_1, \ldots, j_\ell \in \mathbb{N}$, if the system $A\boldsymbol{x} = \boldsymbol{b}$ has a solution $\boldsymbol{x} \in \mathbb{Z}^m$, then*

$$\mathrm{rank}(A) \geq \frac{\ell}{3\max\{t, \log \ell\}}.$$

In the context of the single-challenge trade-off, the lemma states the following: If $\mathcal{A}_1$ solves a lot of challenges, either it must take a long time or $\mathrm{rank}(A)$ is high. This makes sense since the rank approximately corresponds to the number of group elements stored in st_0 which is at most $M/\log N \approx \rho$. Note that Claim 2 formalizes precisely this intuition, so it follows from Lemma 2 almost directly.

Proof of Claim 2. By the assumption that Lemma 1 does not hold, $\mathcal{A}_1$ answers at least $\ell := n/2 \geq 4$ challenges within $t := n/(6\rho \log(n/2)) - 1$ rounds of queries. By Claim 1 Item (i), we may assume that bad only contains parameters where the system $A\boldsymbol{x} = \boldsymbol{b}$ has a solution. So we apply Lemma 2 to get

$$\operatorname{rank}(A) \geq \frac{n}{6 \max\{t, \log(n/2)\}}$$

and we consider two cases depending on max: First, if $\operatorname{rank}(A) \geq n/(6t)$, it follows that $\operatorname{rank}(A) \geq n/(6(t+1))$ and substituting for t, we arrive at $\operatorname{rank}(A) \geq \rho \log(n/2) \geq \rho$ as desired since $\log(n/2) \geq 2$. Second, if $\operatorname{rank}(A) \geq n/(6\log(n/2))$, by the assumption that $n/(6\log(n/2)) \geq \rho$, we also get $\operatorname{rank}(A) \geq \rho$ completing the proof. □

We now turn to the proof of Lemma 2. The first ingredient is an elegant number-theoretic lemma due to van der Waerden (restated and proven in [Laz96]) which precisely describes when a system of Diophantine equations has a solution.

Lemma 3 (van der Waerden)**.** *Consider the Diophantine system of linear equations $A\boldsymbol{x} = \boldsymbol{b}$ with $A \in \mathbb{Z}^{\ell \times m}$ and $\boldsymbol{b} \in \mathbb{Z}^{\ell}$. The system has a solution $\boldsymbol{x} \in \mathbb{Z}^m$ if and only if, for every rational vector $\boldsymbol{v} \in \mathbb{Q}^{\ell}$ such that $\boldsymbol{v}A \in \mathbb{Z}^m$, it also holds that $\boldsymbol{vb} \in \mathbb{Z}$.*

We will show that if $\operatorname{rank}(A)$ is too small, such a vector $\boldsymbol{v}$ always exists. To this end, we introduce the second ingredient which is a combinatorial claim. It is a vector version of the famous distinct subset sums problem and associated conjecture[11] due to Erdős [Guy94, C8].

Claim 4. *For $t, m, \ell \in \mathbb{N}$ with $\ell \geq 4$, let $R \subset \mathbb{Z}^m$ with $|R| = \ell$ and every $\boldsymbol{r} \in R$ satisfying $\|\boldsymbol{r}\|_\infty \leq 2^t$.*[12] *If $m < \ell/(3\max\{t, \log \ell\})$, two subsets $R_1, R_2 \subseteq R$ exist such that at least one subset is non-empty ($R_1 \cup R_2 \neq \emptyset$), they are disjoint ($R_1 \cap R_2 = \emptyset$), and their sums are equal ($\sum_{\boldsymbol{r}_1 \in R_1} \boldsymbol{r}_1 = \sum_{\boldsymbol{r}_2 \in R_2} \boldsymbol{r}_2$).*

Note that the above claim is slightly more general than the vector version considered by Costa et al. [CDD23, Prop. 2.1], but their proof is easily adapted to our setting.

Proof. First, for any subset's sum, notice that the largest absolute value of any resulting vector coordinate is bounded by $2^t\ell$, which follows directly from the triangle inequality and the fact that subsets contain at most ℓ elements.

Second, there exist 2^ℓ distinct (not necessarily disjoint) subsets. Assuming that all subsets' sums are distinct, by the pigeonhole principle, it must hold that

$$\begin{aligned} |\{-2^t\ell, -2^t\ell + 1, \dots, 2^t\ell - 1, 2^t\ell\}|^m &\geq 2^\ell \\ (2^{t+2}\ell)^m &\geq 2^\ell \\ m(t + 2 + \log \ell) &\geq \ell \\ m &\geq \ell/(3\max\{t, \log \ell\}) \end{aligned}$$

[11]Consider a subset $S \subseteq \{1, \dots, 2^t\}$ of size $|S| = m$. *What is the maximum size m so that all subsets of S have distinct sums?* The best-known lower bound is $t + 2 \leq m$ and upper bounds are roughly $m < t + \log t$ (ignoring constants). Erdős conjectured that $m = t + O(1)$ and offered \$500 for proof or refutation [Guy94, C8].

[12]$\|\boldsymbol{r}\|_\infty$ is the infinity norm of the vector $\boldsymbol{r} = (r_1, \dots, r_m)$ and defined as $\max_i |r_i|$.

where the second line follows by overapproximating the size of the set, the third by applying log to both sides, and the fourth by $2 \leq \log \ell \leq \max\{t, \log \ell\}$ since $\ell \geq 4$ by assumption. As a consequence, if $m < \ell/(3\max\{t, \log \ell\})$, then there exist two distinct, but not necessarily disjoint subsets R_1' and R_2' with equal subset-sum.

To get disjoint sets R_1 and R_2, we remove all common elements by defining $R_i := R_i \setminus (R_1' \cap R_2')$ for $i \in \{1, 2\}$. Notice that at least one set is non-empty since R_1' and R_2' are distinct. This completes the proof. □

Equipped with these tools, the proof of Lemma 2 is straightforward.

Proof of Lemma 2. Towards contradiction, assume that there exist A as well as $\boldsymbol{b}$ satisfying the constraints in the lemma's statement and that the system $A\boldsymbol{x} = \boldsymbol{b}$ has a solution, but

$$\operatorname{rank}(A) < \frac{\ell}{3\max\{t, \log \ell\}}.$$

We will show that this implies the existence of a vector $\boldsymbol{v} \in \{-1, 0, 1\}^\ell$ with $\boldsymbol{v} \neq \boldsymbol{0}$ and $\boldsymbol{v}A = \boldsymbol{0}$.

For finding such a $\boldsymbol{v}$, we may assume that $\operatorname{rank}(A) = m$ (i.e., A is full-rank). To see this, assume that $A \in \mathbb{Z}^{\ell \times m}$ is not full rank. We will see that it is sufficient to consider $A' \in \mathbb{Z}^{\ell \times \operatorname{rank}(A)}$, the submatrix containing a linearly independent subset of A's columns that is of maximum size, i.e., $\operatorname{rank}(A)$.

Let the columns of A be $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_m$ and, without loss of generality, assume that $\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\operatorname{rank}(A)}$ are linearly independent, so $A' = (\boldsymbol{c}_1, \ldots, \boldsymbol{c}_{\operatorname{rank}(A)})$. By definition of linear dependence (cf. Definition 8), for any column j with $\operatorname{rank}(A) < j \leq m$, there exist coefficients α_i (with at least one $\alpha_i \neq 0$) such that

$$\alpha_1 \boldsymbol{c}_1 + \cdots + \alpha_{\operatorname{rank}(A)} \boldsymbol{c}_{\operatorname{rank}(A)} + \alpha_j \boldsymbol{c}_j = \boldsymbol{0}.$$

Multiplying by $\boldsymbol{v}$, we get

$$\alpha_1 \boldsymbol{v}\boldsymbol{c}_1 + \cdots + \alpha_{\operatorname{rank}(A)} \boldsymbol{v}\boldsymbol{c}_{\operatorname{rank}(A)} + \alpha_j \boldsymbol{v}\boldsymbol{c}_j = \boldsymbol{v}\boldsymbol{0}$$

which simplifies to $\alpha_j \boldsymbol{v}\boldsymbol{c}_j = \boldsymbol{0}$ since $\boldsymbol{v}A' = \boldsymbol{0}$ by assumption. Note that $\alpha_j \neq 0$ as otherwise some α_i where $1 \leq i \leq \operatorname{rank}(A)$ would need to be non-zero which would imply that $\boldsymbol{c}_1 \ldots, \boldsymbol{c}_{\operatorname{rank}(A)}$ are not linearly independent. Dividing by α_j, it follows that $\boldsymbol{v}\boldsymbol{c}_j = \boldsymbol{0}$ for all $\operatorname{rank}(A) < j \leq m$ which implies $\boldsymbol{v}A = 0$. It follows that we can restrict our attention to the full-rank case.

Now we use *Claim* 4 to construct $\boldsymbol{v}$. Whenever A contains two identical rows, it is straightforward to define an appropriate $\boldsymbol{v}$. Thus, we may assume that the rows of A are distinct and form a set $R \subset \mathbb{Z}^m$ of size $|R| = \ell$. Combining this with the assumption $m = \operatorname{rank}(A) < \ell/(3\max\{t, \log \ell\})$, Claim 4 applies, so there exist two disjoint subsets R_1 and R_2 with equal sum with one of them being non-empty. Hence, we initialize $\boldsymbol{v} := \boldsymbol{0}$, and set $v_i := 1$ if $\boldsymbol{r}_i \in R_1$ and $v_j := -1$ if $\boldsymbol{r}_j \in R_2$. This is unambiguous as the subsets are disjoint. Further, notice that $\boldsymbol{v} \neq \boldsymbol{0}$ since one subset is non-empty, and, since the subsets' sums are equal, $\boldsymbol{v}A = \boldsymbol{0}$ as required.

Last, we apply Lemma 3 to $A\boldsymbol{x} = \boldsymbol{b}$ and $\boldsymbol{v}$. By definition, $\boldsymbol{b} \in \mathbb{Z}^\ell$ contains no element $b_i = 0$, all b_i are pairwise distinct and exponentially far apart. Thus, $\boldsymbol{v}\boldsymbol{b} = \beta \in \mathbb{Z}$ with $\beta \neq 0$ since $\boldsymbol{v} \in \{-1, 0, 1\}$. Setting $\boldsymbol{v}' := \frac{1}{|\beta|+1}\boldsymbol{v}$, we arrive at the conclusion that the system $A\boldsymbol{x} = \boldsymbol{b}$ does not have a solution by van der Waerden (Lemma 3). This is a contradiction, completing the proof. □

5.5 Incompressibility Argument

Recall that Claim 3 claims the existence of parameters in bad such that $\mathrm{rank}(A) < \rho$. To this end, we will show that $\mathsf{rank}_{\mathsf{min}} < \rho$ where $\mathsf{rank}_{\mathsf{min}}$ is the minimum rank over all possible parameters in bad, that is,

$$\mathsf{rank}_{\mathsf{min}} := \min_{\mathsf{bad}}(\mathrm{rank}(A)). \tag{2}$$

Our proof use an incompressibility argument (e.g., [Yao90, DTT10, ACP+17, AAC+17, CK18]) to bound $\mathsf{rank}_{\mathsf{min}}$ in terms of $|\mathsf{st}_0| = M$ and since ρ depends on M, we get the desired bound. Formally, we follow the framework of De, Trevisan, and Tulsiani [DTT10] stated below.

Lemma 4 (De, Trevisan, and Tulsiani [DTT10])**.** *Let* $\mathsf{Enc}\colon \mathcal{S} \times \{0,1\}^\mu \to \{0,1\}^r$ *and* $\mathsf{Dec}\colon \{0,1\}^\mu \times \{0,1\}^r \to \mathcal{S}$ *be randomized encoding and decoding procedures such that, for every* $s \in \mathcal{S}$,

$$\Pr_{r \leftarrow \{0,1\}^r}[\mathsf{Dec}(\mathsf{Enc}(s;r);r) = s] \geq \delta.$$

Then $\mu \geq \log|\mathcal{S}| - \log 1/\delta$.

Intuitively, the analysis of preprocessing algorithms $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ using the framework proceeds as follows. $\mathcal{S}$ is usually a set of functions, injections, etc. where $\mathcal{A}$ has oracle access to one $s \in \mathcal{S}$. Enc, knowing $s \in \mathcal{S}$, runs $\mathcal{A}_0^s$ to get st_0 and then $\mathcal{A}_1(\mathsf{st}_0)^s$ such that $\mathcal{A}_1$'s query behavior implicitly allows one to reconstruct parts of s. It outputs an encoding comprised of st_0 and an additional hint. Dec receives this encoding and runs $\mathcal{A}_1(\mathsf{st}_0)^{\mathrm{hint}}$ where the hint is used to simulate the oracle. Given $\mathcal{A}_1$'s query behavior, Dec recovers s at the cost of $\|\mathsf{st}_0\| + \|\mathrm{hint}\|$ bits where the hint is carefully designed to be smaller than $\log|\mathcal{S}|$ bits. Since encoding s using less than $\log|\mathcal{S}|$ bits would constitute information-theoretically impossible compression, st_0 must be large enough to contain the remaining information.

In our case, we work with $\mathcal{S} := \mathsf{bad}$, but effectively we only efficiently encode the labeling function $\Sigma \in \mathrm{Inj}(\mathbb{Z}_N, \mathcal{L})$ while using a naive encoding for N, h, and w. Our argument then follows the strategy outlined in the previous paragraph. As in Section 5.1, $\mathcal{A}_1$ induces the system of Diophantine equations $A\boldsymbol{x} = \boldsymbol{b}$ where A and $\boldsymbol{b}$ are known. Every element of $\boldsymbol{x}$ mod N corresponds to the discrete logarithm of some label in $\mathcal{L}$ and Dec can solve the system to extract $\boldsymbol{x}$, effectively learning some discrete logarithms, and, in turn, the labels for the corresponding group elements without them being encoded in the hint. Given usual linear algebra intuition, the solvability of this system and the uniqueness of its solution are related to the rank of A. Here, this is $\mathsf{rank}_{\mathsf{min}}$ which allows us to connect $\|\mathsf{st}_0\| = M$ to $\mathsf{rank}_{\mathsf{min}}$.

Equipped with this high-level explanation, we now state Claim 5. It posits the existence of $(\mathsf{Enc}, \mathsf{Dec})$ that efficiently encode the labeling function $\Sigma \in \mathrm{Inj}(\mathbb{Z}_N, \mathcal{L})$ where the encoding size depends on M and $\mathsf{rank}_{\mathsf{min}}$.

Claim 5. *If Lemma 1 does not hold, for* $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$, *there exist* $(\mathsf{Enc}, \mathsf{Dec})$ *which, given auxiliary inputs* N, h, *and* w, *encodes the labeling function* Σ *using*

$$\log\binom{|\mathcal{L}|}{N} + \log(N!) + M + \log n + \log Q - \mathsf{rank}_{\mathsf{min}} \cdot (\log N - 3(\log n + \log Q) - 3)$$

bits with probability 1.

Before constructing such $(\mathsf{Enc}, \mathsf{Dec})$, we show how their existence implies Claim 3.

Proof of Claim 3. In general, optimally encoding an element of params requires $\log|\mathsf{params}| = \mu' + \log\binom{|\mathcal{L}|}{N} + \log(N!)$ bits where μ' is the length of an optimal encoding of N, h, and w. Since bad contains at least a $1/\lambda^c$ fraction of params, it follows from elementary logarithmic identities that an optimal encoding of a tuple in bad requires

$$\mu' + \log\binom{|\mathcal{L}|}{N} + \log(N!) + \log(\lambda^{-c}) = \mu' + \log\binom{|\mathcal{L}|}{N} + \log(N!) - c\log\lambda$$

bits. Now, by Claim 5 and applying Lemma 4 while canceling common terms (implicitly accounting for μ'), it must hold that

$$-c\log\lambda \leq M + \log n + \log Q - \mathsf{rank}_{\mathsf{min}} \cdot (\log N - 3(\log n + \log Q) - 3)$$

which implies

$$\mathsf{rank}_{\mathsf{min}} \leq \frac{M + \log n + \log Q + c\log\lambda}{\log N - 3(\log Q - \log n) - 3}.$$

By the definition of ρ in Lemma 1, it follows that $\mathsf{rank}_{\mathsf{min}} < \rho$, so there exists a $(N, \Sigma, \mathsf{h}, w) \in \mathsf{bad}$ such that $\operatorname{rank}(A) < \rho$ as desired. $\square$

The only thing left is to show is that $(\mathsf{Enc}, \mathsf{Dec})$ for Σ as in Claim 5 exist. This requires careful handling of various technicalities, so we will only sketch the proof here and defer the complete proof to Appendix C.

Proof of Claim 5 (Sketch). The high-level idea is that Enc and Dec both run $\mathcal{A}_1$ as a subroutine. $\mathcal{A}_1$ induces a system of equations $A\boldsymbol{x} = \boldsymbol{b}$ with $\operatorname{rank}(A) \geq \mathsf{rank}_{\mathsf{min}}$. This allows Enc to encode the discrete logarithm of $\mathsf{rank}_{\mathsf{min}}$ many group element labels using a short hint instead of the naive encoding that costs roughly $\log N$ bits.

Given input Σ and auxiliary inputs N, h and w, Enc computes st_0 by running $\mathcal{A}_0^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$ and then executes $\mathcal{A}_1^{\Sigma,\mathsf{h}}(\mathsf{st}_0)$ on every challenge $j \in [0, n)$ for at most $t := \frac{n}{6\rho\log(n/2)} - 1$ rounds in parallel in lockstep. During this, Enc successively constructs an encoding of Σ. This encoding contains the information necessary for Dec to also run $\mathcal{A}_1$ without knowing the oracle Σ.

In particular, Enc (and also Dec) track the algebraic representation of queries as in Section 5.1. The algebraic representation of a label might include some indeterminates which correspond to labels that $\mathcal{A}_1$ input to Σ out of the blue (i.e., either it guessed the label or it extracted it from st_0).

On the one hand, Enc needs to ensure that Dec can answer oracle queries to the labeling function Σ correctly. To this end, it first encodes the image of Σ using $\log\binom{|\mathcal{L}|}{N}$ bits, and, for some queries, it adds a hint of circa $\log N$ bits which specify the label to be returned. Note that this hint is only required for *some* queries since the output of a query is sometimes uniquely determined by the algebraic representations of the preceding queries (e.g., when $\mathcal{A}_1$ repeats a query). Since Dec tracks those representations, it can answer such queries without additional help from Enc.

On the other hand, Enc also needs to tell Dec when the label $\Sigma(w^{2^j})$ is queried to h. This requires a hint of $\log n + \log Q$ bits; $\log n$ bits to identify the instance of $\mathcal{A}_1$ and $\log Q$ to specify the query within that instance. Given $\mathsf{rank}_{\mathsf{min}}$ many hints, Dec derives a system of equations $A\boldsymbol{x} = \boldsymbol{b}$. The vector $\boldsymbol{b} \in \mathbb{Z}^{\mathsf{rank}_{\mathsf{min}}}$ contains the $2^i w$ specified by the hints and the rows of A the corresponding algebraic representations. So $\boldsymbol{x} \in \mathbb{Z}^m$, when taken modulo N, corresponds to the discrete logarithms of some indeterminates within these algebraic representations.

Computing $\mathsf{HNF}(A) = (H, U)$ yields a lower triangular matrix H such that $A\boldsymbol{x} = \boldsymbol{b}$ has a solution if and only if $H\boldsymbol{y} = \boldsymbol{b}$ has a solution where $\boldsymbol{x} = U\boldsymbol{y}$. Recall that H has $\mathsf{rank}_{\mathsf{min}}$ many non-zero columns and, due to the triangular shape of H, every non-zero column constrains one entry of $\boldsymbol{x}$. For example, consider

$$\begin{pmatrix} h_{11} & & \\ h_{21} & h_{22} & \\ h_{31} & h_{32} & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 2^i w \\ 2^j w \\ 2^k w \end{pmatrix}$$

with $\mathsf{rank}_{\mathsf{min}} = 2$ and pivot elements h_{11} and h_{22}.

Next, reduce every matrix and vector modulo N. Notice that Claim 1 Items (ii) and (iii) guarantee that $H \bmod N$ still has the same shape and that the pivot elements are invertible. This implies that every non-zero column of $H \bmod N$ uniquely constrains one entry in $\boldsymbol{y} \bmod N$ and thus effectively one indeterminate. So given $\mathcal{A}_1$'s query behavior, Dec derives the discrete logarithm of $\mathsf{rank}_{\mathsf{min}}$ many indeterminates with a hint of only $\mathsf{rank}_{\mathsf{min}}(\log n + \log Q)$ bits.

Last, we analyze the encoding size relative to the naive encoding of $\log \binom{|\mathcal{L}|}{N} + \log(N!)$ bits. Answering queries in general requires no more bits than the naive encoding: $\log \binom{|\mathcal{L}|}{N}$ bits for the image and roughly $\log N$ bits for every query. For every indeterminate recovered from the system of equations, the hint is only $\log n + \log Q$ bits which is smaller than $\log(N/2)$ for λ large enough. Thus, the total encoding size is

$$\underbrace{\log \binom{|\mathcal{L}|}{N} + \log(N!)}_{\text{Cost of DLogs in general}} + \underbrace{M}_{|\mathsf{st}_0|} - \underbrace{\mathsf{rank}_{\mathsf{min}} \cdot (\log N - 1 - \log n - \log Q)}_{\text{Savings due to DLogs from } A\boldsymbol{x}=\boldsymbol{b}}.$$

□

Note that the savings in the claim's actual statement are lower since Enc has to handle colliding Σ queries which introduce additional overhead.

6 Multi-challenge Memory Complexity

We now consider the whole evaluation of TdScrypt, and we want to prove that it requires $\Omega(\frac{n^2}{\log n} \log N)$ memory almost always—no matter the evaluation strategy. This is formalized in the theorem below which is in the spirit of Alwen et al. [ACP+17, Thm. 1].

Theorem 1. *Let $n \in \operatorname{poly}(\lambda)$ with $n \geq 8$ and let $\mathcal{A}$ be a deterministic parallel oracle machine that evaluates* TdScrypt *correctly with probability $\chi(\lambda)$ over the choice of the parameters and input W. Then, assuming that factoring is hard, in the GGM and ROM with probability at least $\chi(\lambda) - \epsilon(\lambda)$,*

$$\mathsf{cc}_{\mathsf{mem}}(\mathcal{A}^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))) \in \Omega\left(\frac{n^2}{\log n} \log N\right)$$

where $\epsilon(\lambda) \in \mathrm{negl}(\lambda)$*, and the probability is taken over* $(N, \Sigma, \mathsf{h}, w) \xleftarrow{\$} \mathsf{params}(\lambda)$*.*

As stated before, our proof follows Alwen et al.'s [ACP+17] line of reasoning closely, that is, considering the single-challenge game and then generalizing it to the multi-challenge setting which roughly equals any actual evaluation. While our single-challenge trade-off proof differed considerably, the multi-challenge proof is almost identical. On a high level, the single-challenge trade-offs can simply be swapped out, and this only affects constants. Alwen et al. [ACP+17] took great care to state these explicitly. In contrast, we stick to asymptotics since our primary focus is showing that a TMHF exists.

TdScrypt **Definition.** Recall the definition of TdScrypt. Given input $\Sigma(w)$, compute the powers $\Sigma(2^i w)$, $0 \leq i < n$. Then, define $S_0 = \mathsf{h}(\Sigma(2^n w), 0^\ell)$ and, for $1 < i \leq n$, $S_i = \mathsf{h}(\Sigma(2^{j_i} w), S_{i-1})$ where $j_i = S_{i-1} \bmod n$ is the *ith challenge*. For $k \in [n]$, we define s_k to be the round in which the kth challenge is issued, i.e., when the value S_{k-1} is the result of a query.

High-level Proof Strategy. On a high-level, the proof strategy is as follows. We modify Corollary 1 to work with a single adversary $\mathcal{A}$ that can be thought of as a (potentially dishonest) evaluation strategy for TdScrypt. For some $k \in [n]$, we split the execution of $\mathcal{A}$ into two parts, up to round s_k when the kth challenge $j_k = S_{k-1} \bmod n$ is issued and afterward. These parts map to $\mathcal{A}_0$ and $\mathcal{A}_1$ as in Corollary 1 where the explicit input j to $\mathcal{A}_1$ is replaced by slightly altering (i.e., programming) the random oracle such that $S_{k-1} \bmod n = j$ (but $\lfloor S_{k-1}/n \rfloor$ is unchanged) and the advice st_0 is now the input state st_{s_k}. So we get a time-memory trade-off in the style of Corollary 1 that intuitively states: For at least half of the n possible values j_k may take, either $\mathcal{A}$ needs a lot of rounds to answer the challenge, or all the input state st_r with $r \leq s_k$ have a size $\|\mathsf{st}_r\|$ that is increasing with r and reasonably large when r is closer to s_k. Then, we can argue that this trade-off must hold for roughly half of all n challenges by Hoeffding's inequality, and we can thus add the input state sizes $\|\mathsf{st}_r\|$ for all n challenges to get the desired memory complexity.

Proof Sketch. As a first step, we characterize parameter combinations that are not amenable to our proof strategy.

Definition 9 (Collisions)**.** For given N, Σ, and w, the set $\mathsf{colliding}_k$ contains all random oracles h that cause a collision amongst the preceding $\{S_0, \ldots, S_{k-1}\}$ during the honest $\mathsf{Eval}(\mathsf{pp}, \Sigma(w))$.

Definition 10 (Rounding Impossible)**.** For given N, Σ, and w, the set $\mathsf{rounding}_k(n)$ contains all random oracles h that are not amenable to programming some challenge $j \in [0, n)$. That is, there exists $0 \leq i \leq k$ such that $S_i > \lfloor 2^\ell/n \rfloor n - 1$.

Intuitively, the above two definitions characterize random oracles where we cannot program $j_k = S_{k-1} \bmod n$ uniformly. Either the value is not independent of previously issued challenges due to collisions, or S_{k-1} might overflow for some choices of j_k. The next definition then tells us that any fixed adversary $\mathcal{A}$ almost always learns the values $(S_i)_{0 \leq i \leq n}$ in sequential order.

Definition 11 (Wrong Evaluation Order)**.** For given N, Σ, w, and $\mathcal{A}$, the set $\mathsf{wrongOrder}_k$ contains all random oracles h where during the execution of $\mathcal{A}^{\Sigma,\mathsf{h}}(\Sigma(w))$ a query to h of the form $(\cdot, S_j)$ occurs before one of the form $(\cdot, S_i)$ for $0 \leq i < j < k$.

With the above definitions in place, we can now define the time-memory trade-off. Essentially, this is a modified version of the single-challenge game (cf. Definition 6) merged with the time-memory trade-off Lemma 1 (resp. Corollary 1).

Definition 12 (Hard Challenge)**.** Let $\lambda, n \in \mathbb{N}$ with $n \geq 8$, $k, j \in [0, n)$, $(N, \Sigma, \mathsf{h}, w) \in \mathsf{params}(\lambda)$ with $\mathsf{h} \notin \mathsf{colliding}_{k-1} \cup \mathsf{rounding}_{k-1}(n)$, and $\mathcal{A}$ a deterministic oracle machine making at most $Q \in \text{poly}(\lambda)$ queries. Consider the execution $\mathcal{A}^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$ up to round $s_k \in \mathbb{N}^+$, which is the earliest round where the query $(W^{2^{j_{k-1}}}, S_{k-2}) \in \mathbf{qrs}^{\mathsf{h}}$ with $j_{k-1} = S_{k-2} \bmod n$ occurs (in the case $k = 1$, the query is $(W^{2^n}, 0^\ell)$). For any round r with $0 \leq r \leq s_k$, let $M_r = \|\mathsf{st}_r\|$, and associate a fixed value ρ_r to it similarly to Lemma 1.[13] It follows that

$$\rho_r \in \Theta(M_r + \log n + \log Q + \log \lambda)/(\log(N) - \log Q - \log n).$$

Define $\mathsf{h}' = \mathsf{h}$ except for S_{k-1} which will be changed to S'_{k-1} with $\lfloor S'_{k-1}/n \rfloor = \lfloor S_{k-1}/n \rfloor$ but $S'_{k-1} \bmod n = j$, and consider the execution $\mathcal{A}^{\Sigma,\mathsf{h}'}(\Sigma(1), \Sigma(x))$. Let $t_{k,j,r} \in \mathbb{N}^+$ be minimal such that the query $(W^{2^j}, S_{k-1}) \in \mathbf{qrs}^{\mathsf{h}'}$ occurs after s_k in round $r + t_{k,j,r} > s_k$ where $t_{k,j,r} = \infty$ if it never does. Then, let

$$r_k = \underset{0 \leq r \leq s_k}{\arg\max} \left\{ \frac{n}{6\rho_r \log(n/2)} - (s_k - r) \right\},$$

and define

$$\mathsf{Hard}^{\Sigma,\mathsf{h},n}_{\mathcal{A}(w)}(k, j) = 1 \text{ if and only if } t_{k,j,r_k} > \frac{n}{6\rho_{r_k} \log(n/2)}.$$

We will re-use the notation introduced in these definitions and use it to state the modified single-challenge trade-off. Essentially, it says that most parameters do not fall into $\mathsf{colliding}$, $\mathsf{rounding}$ or $\mathsf{wrongOrder}$, and that the time-memory trade-off holds for them.

Lemma 5. *For every deterministic parallel oracle machine $\mathcal{A}$, there exist negligible functions $\epsilon(\lambda) \in \text{negl}(\lambda)$ such that, for all $n \in \text{poly}(\lambda)$ with $n \geq 8$, there exists a subset $\mathsf{good} \subseteq \mathsf{params}(\lambda)$ such that, for every $(N, \Sigma, \mathsf{h}, w) \in \mathsf{good}$ and $k \in \mathbb{N}$ with $1 \leq k \leq n$, $|\mathsf{good}| \geq (1 - \epsilon(\lambda))|\mathsf{params}(\lambda)|$ and the following holds:*

- $\mathsf{h} \notin \mathsf{colliding}_{k-1}$*;*
- $\mathsf{h} \notin \mathsf{rounding}_{k-1}(n)$*;*
- $\mathsf{h} \notin \mathsf{wrongOrder}_{k-1}$*; and*
- $\Pr_{j \xleftarrow{\$} [0,n)}\left[\mathsf{Hard}^{\Sigma,\mathsf{h},n}_{\mathcal{A}(w)}(k, j)\right] \geq 1/2.$

[13]To be precise, $\rho = (M + \log n + 2\log Q + c \log \lambda + 1)/(\log(N) - 3(\log Q - \log n) - 3)$ where c is as in Lemma 1.

Proof (Sketch). As in the proof of Lemma 1, towards contradiction, we assume that for some n and k, bad is large, i.e., $|\mathsf{bad}| \geq \lambda^{-c}|\,\mathsf{params}(\lambda)$, and we consider four cases. We will show that every case leads to a contradiction.

- $\mathsf{h} \in \mathsf{colliding}_{k-1}$: View N, Σ and w as fixed and count how many choices of h cause a collision. There are at most $k^2 \leq n^2$ possible colliding pairs S_i and S_j and, for every pair, a collision happens for at most a $2^{-\log N}$ fraction of random oracles as h's output length is at least $\log N$. So, at most an $n^2/2^{\log N}$ fraction of random oracles might cause a collision which is negligible and therefore contradicts the assumption that bad covers a polynomial fraction of params (cf. [ACP$^+$17, Clm. 15]).
- $\mathsf{h} \in \mathsf{rounding}_{k-1}$ and $\mathsf{h} \in \mathsf{wrongOrder}_{k-1}$: Both analyses are analogous the previous case with the fraction of random oracles being in the order of $1/N$ as well (cf. [ACP$^+$17, Clms. 16 & 18]).
- $\Pr_{j \xleftarrow{\$} [n]}\left[\mathsf{Hard}^{\Sigma,\mathsf{h},n}_{\mathcal{A}(w)}(k,j)\right] < 1/2$: This case follows the proof of Lemma 1 with some modifications. Instead of considering explicit preprocessing- and online algorithms $(\mathcal{A}_0, \mathcal{A}_1)$, we split the execution of $\mathcal{A}$ into two parts—up to round r (preprocessing) and everything afterward (online). In this setting, we cannot pass the challenge j to the online algorithm explicitly anymore. Instead, we will program the random oracle h to include the challenge implicitly, i.e., setting $S_{k-1} \bmod n = j$. Programming is always possible as long as $\mathsf{h} \notin \mathsf{colliding}_{k-1} \cup \mathsf{rounding}_{k-1}(n)$, and we may assume that this is the case by the preceding case analyses. For Dec to know when to program h, $\log Q$ additional bits of hint are required to recognize the query $(W^{2^{j_{k-1}}}, S_{k-2})$ (cf. Definition 12). Apart from these modifications, the proof is identical to the one of Lemma 1.

This discussion completes the proof sketch. $\square$

We now use Lemma 5's trade-off to lower bound the sum $\sum_{r=1}^{s_n+1} \rho_r$ where we note that ρ_r is a quantity related to the size of the input state $\|\mathsf{st}_r\|$. Note that the sum covers all rounds up to round $s_n + 1$, i.e., the round in which $\mathcal{A}$ first computes S_n, the output of TdScrypt. This strategy is taken from [ACP$^+$17, Sec. 5] so we refer interested readers there for more details.

Claim 6 (Lower Bound on $\sum \rho_r$)**.** *If $(N, \Sigma, \mathsf{h}, w) \in \mathsf{good}$ and $\mathcal{A}^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$ queries S_n, then $\sum_{r=1}^{s_n+1} \rho_r \in \Omega(n^2/\log n)$.*

Proof (Sketch). First, note that if the parameters are in good, whenever $\mathcal{A}$ queries S_n, it must have received all $\{S_i\}_{0\leq i\leq n}$ during its execution, and it must receive them in order. Second, if $\mathsf{Hard}^{\Sigma,\mathsf{h},n}_{\mathcal{A}(w)}(k,j) = 1$, then, for any $0 \leq r \leq s_k$,

$$t_{k,j,r} > \frac{n}{6\rho_r \log(n/2)} \tag{3}$$

by the choice of r_k (cf. [ACP$^+$17, Clms. 5 & 10]). Third, by a generalization of Hoeffding's inequality (cf. [ACP$^+$17, Clm. 7]), we argue that for any fixing of challenges, at least $n(1/2 - \epsilon)$ challenges are Hard where $\epsilon > 0$ (cf. [ACP$^+$17, Clm. 19]).

Given the above, we apply [ACP$^+$17, Clm. 8] to $\rho_r > \frac{n}{6t_{k,j,r}\log(n/2)}$, which is Equation (3) rearranged. This gets us $\sum_{r=1}^{s_n+1} \rho_r \in \Omega(n^2/\log n)$ as desired (cf. [ACP$^+$17, Clm. 11]). $\square$

Finally, we prove Theorem 1 by converting the quantity $\sum \rho_r$ to memory complexity.

Proof of Theorem 1 (Sketch). By Lemma 5, an overwhelming fraction of parameters is in good, and $\mathcal{A}$ queries S_n with probability $\chi(\lambda)$. Applying Claim 6, we get a lower bound of $\sum_{r=1}^{s_n+1} \rho_r \in \Omega(n^2/\log n)$. Since $M_r \in \Omega(\rho_r \log N)$ for all $r \in \mathbb{N}$ by definition, it follows that $\sum_{r=1}^{s_n+1} M_r \in \Omega(\frac{n^2}{\log n} \log N)$. Note that this sum is equivalent to the $\mathsf{cc_{mem}}$ of $\mathcal{A}$ which completes the proof sketch. $\square$

Acknowledgements. We thank the Eurocrypt reviewers for their thorough review and for pointing out related works. This research was funded in whole or in part by the Austrian Science Fund (FWF) 10.55776/F85.

References

[AAC⁺17] Hamza Abusalah, Joël Alwen, Bram Cohen, Danylo Khilko, Krzysztof Pietrzak, and Leonid Reyzin. Beyond hellman's time-memory trade-offs with applications to proofs of space. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part II*, volume 10625 of *LNCS*, pages 357–379. Springer, Heidelberg, December 2017.

[ABB22] Mohammad Hassan Ameri, Alexander R. Block, and Jeremiah Blocki. Memory-hard puzzles in the standard model with applications to memory-hard functions and resource-bounded locally decodable codes. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks*, pages 45–68, Cham, 2022. Springer International Publishing.

[ABH17] Joël Alwen, Jeremiah Blocki, and Ben Harsha. Practical graphs for optimal side-channel resistant memory-hard functions. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1001–1017. ACM Press, October / November 2017.

[ABP17] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Depth-robust graphs and their cumulative memory complexity. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 3–32. Springer, Heidelberg, April / May 2017.

[ABP18] Joël Alwen, Jeremiah Blocki, and Krzysztof Pietrzak. Sustained space complexity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 99–130. Springer, Heidelberg, April / May 2018.

[ABZ20] Mohammad Hassan Ameri, Jeremiah Blocki, and Samson Zhou. Computationally data-independent memory hard functions. In Thomas Vidick, editor, *ITCS 2020*, volume 151, pages 36:1–36:28. LIPIcs, January 2020.

[ACK⁺16] Joël Alwen, Binyi Chen, Chethan Kamath, Vladimir Kolmogorov, Krzysztof Pietrzak, and Stefano Tessaro. On the complexity of scrypt and proofs of space in the parallel random oracle model. In Marc Fischlin and Jean-Sébastien

Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 358–387. Springer, Heidelberg, May 2016.

[ACP+17] Joël Alwen, Binyi Chen, Krzysztof Pietrzak, Leonid Reyzin, and Stefano Tessaro. Scrypt is maximally memory-hard. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*, pages 33–62. Springer, Heidelberg, April / May 2017.

[AS15] Joël Alwen and Vladimir Serbinenko. High parallel complexity graphs and memory-hard functions. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 595–603. ACM Press, June 2015.

[BCS16] Dan Boneh, Henry Corrigan-Gibbs, and Stuart E. Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 220–248. Springer, Heidelberg, December 2016.

[BDK16] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. Argon2: New generation of memory-hard functions for password hashing and other applications. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 292–302, 2016.

[BH22] Jeremiah Blocki and Blake Holman. Sustained space and cumulative complexity trade-offs for data-dependent memory-hard functions. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part III*, volume 13509 of *LNCS*, pages 222–251. Springer, Heidelberg, August 2022.

[BHK+19] Jeremiah Blocki, Benjamin Harsha, Siteng Kang, Seunghoon Lee, Lu Xing, and Samson Zhou. Data-independent memory hard functions: New attacks and stronger constructions. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part II*, volume 11693 of *LNCS*, pages 573–607. Springer, Heidelberg, August 2019.

[BLP23] Alex Biryukov and Marius Lombard-Platet. Pured: A unified framework for resource-hard functions. Cryptology ePrint Archive, Paper 2023/1809, 2023. https://eprint.iacr.org/2023/1809.

[BP17] Alex Biryukov and Léo Perrin. Symmetrically and asymmetrically hard cryptography. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part III*, volume 10626 of *LNCS*, pages 417–445. Springer, Heidelberg, December 2017.

[BRZ18] Jeremiah Blocki, Ling Ren, and Samson Zhou. Bandwidth-hard functions: Reductions and lower bounds. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1820–1836. ACM Press, October 2018.

[CDD23] Simone Costa, Marco Dalai, and Stefano Della Fiore. Variations on the Erdős distinct-sums problem. *Discrete Applied Mathematics*, 325:172–185, 2023.

[CK18] Henry Corrigan-Gibbs and Dmitry Kogan. The discrete-logarithm problem with preprocessing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 415–447. Springer, Heidelberg, April / May 2018.

[DN93] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 139–147. Springer, Heidelberg, August 1993.

[DTT10] Anindya De, Luca Trevisan, and Madhur Tulsiani. Time space tradeoffs for attacks against one-way functions and PRGs. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 649–665. Springer, Heidelberg, August 2010.

[Guy94] Richard K. Guy. *Unsolved Problems in Number Theory*. Problem Books in Mathematics. Springer, New York, NY, 2 edition, 1994.

[Her09] Charles Hermite. *Sur l'introduction des variables continues dans la théorie des nombres*, volume 1 of *Cambridge Library Collection - Mathematics*, page 164–192. Cambridge University Press, 2009.

[Kal00] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. Request for Comments 2898, Internet Engineering Task Force, September 2000.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

[KLX20] Jonathan Katz, Julian Loss, and Jiayu Xu. On the security of time-lock puzzles and timed commitments. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 390–413. Springer, Heidelberg, November 2020.

[Laz96] Felix Lazebnik. On systems of linear diophantine equations. *Mathematics Magazine*, 69(4):261–266, 1996.

[MVOV97] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC press, 1997.

[MW01] Daniele Micciancio and Bogdan Warinschi. A linear space algorithm for computing the hermite normal form. In *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*, ISSAC '01, page 231–236, New York, NY, USA, 2001. Association for Computing Machinery.

[Per09] Colin Percival. Stronger key derivation via sequential memory-hard functions. In *BSDCan 2009*, 2009.

[RD16] Ling Ren and Srinivas Devadas. Proof of space from stacked expanders. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 262–285. Springer, Heidelberg, October / November 2016.

[RD17] Ling Ren and Srinivas Devadas. Bandwidth hard functions for ASIC resistance. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 466–492. Springer, Heidelberg, November 2017.

[Rot22] Lior Rotem. Revisiting the uber assumption in the algebraic group model: Fine-grained bounds in hidden-order groups and improved reductions in bilinear groups. In Dana Dachman-Soled, editor, *3rd Conference on Information-Theoretic Cryptography (ITC 2022)*, volume 230 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:13, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[RS20] Lior Rotem and Gil Segev. Generically speeding-up repeated squaring is equivalent to factoring: Sharp thresholds for all generic-ring delay functions. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 481–509. Springer, Heidelberg, August 2020.

[RSW96] Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, USA, 1996.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[vzGS13] Joachim von zur Gathen and Igor E. Shparlinski. Generating safe primes. *Journal of Mathematical Cryptology*, 7(4):333–365, 2013.

[Yao90] Andrew Chi-Chih Yao. Coherent functions and program checkers (extended abstract). In *22nd ACM STOC*, pages 84–94. ACM Press, May 1990.

A Detailed Cumulative Memory Complexity Analysis

To analyze the cumulative memory complexity (CMC) of TdScrypt's Eval and TDEval, we first discuss the basic operations performed by them before describing their CMC.

Basic Operations. The relevant parameters are the RSA modulus N', the group size N, the output length ω_{h} of the hash function h, and the number of iterations n. Note that since $N \approx N'/4$ it follows that N and N' have essentially the same bit length. As stated in Section 2, we further have that $\omega_{\mathsf{h}} \in \Theta(\log(N))$ and $n \ll N$ since $n \in \mathrm{poly}(\lambda)$ (as otherwise Eval would not run in $\mathrm{poly}(\lambda)$).

We first give an overview of the operations used in Eval and TDEval, listing the amount of bit operations, i.e., additions and multiplications in the field $\mathbb{F}_2$, and the scratch space in terms of bits used in their computation. The results are summarized in Table 2.

Algorithm Eval performs group operations in $\mathbb{QR}_{N'}$ (Line 09), evaluates hash function h (Lines 11 and 14), and reduces outputs of h modulo n (Line 13). Recall that, by definition, evaluating h requires the same computational effort as computing a group operation, i.e., $\Theta(\log(N')^2)$ bit operations while using $\Theta(\log(N'))$ bits of scratch space (e.g., [MVOV97, Sec. 14.3.3]). The computation of $j_i := S_{i-1} \bmod n$, on the other hand,

Operation	Bit Operations	Scratch Space
Multiplication in $\mathbb{QR}_{N'}$	$\Theta(\log(N')^2)$	$\Theta(\log(N'))$
Exponentiation in $\mathbb{QR}_{N'}$	$\Theta(\log(N)\log(N')^2)$	$\Theta(\log(N'))$
Multiplication in $\mathbb{Z}_N$	$\Theta(\log(N)^2)$	$\Theta(\log(N))$
Evaluation of h	$\Theta(\log(N')^2)$	$\Theta(\log(N'))$
$S_{i-1} \bmod N$	$\Theta(\log(n)\log(N))$	$\Theta(\log(N))$
$2^j \bmod n$	$\Theta(\log(n)\log(N)^2)$	$\Theta(\log(N))$

Table 2: Time and memory usage of operations used in Eval and TDEval. Note that $\log(N) \approx \log(N') > \log(n)$.

requires $\Theta(\log(n)\log(N'))$ bit operations and scratch space of $\log(N)$ bits (e.g., [MVOV97, Sec. 14.2.5]), where we used that $\omega_{\mathsf{h}} \in \Theta(\log(N))$.

TDEval also computes h (Lines 19 and 24) and also reduces them modulo n (Line 21), but it additionally performs exponentiations in $\mathbb{QR}_{N'}$ (Lines 18 and 23) and reduces n-bit integers of the form 2^j with $j \in [0,n)$ modulo N (Lines 17 and 22). Exponentiations can be computed using square-and-multiply at the cost of $\Theta(\log(N))$ group operations. The operation $m := 2^j \bmod n$ can be implemented using a lookup table that, for $i \in [0, \lceil\log(n)\rceil]$, stores the values $2^{2^i} \bmod N$. Then, by writing $j := \sum_{i=0}^{\lceil\log(n)\rceil} b_i 2^i$, one can compute m using at most $\lceil\log(n)\rceil$ group operations in $\mathbb{Z}_N$ as

$$\prod_{i=0}^{\lceil\log(n)\rceil} b_i 2^{2^i} \bmod N = 2^{\sum_{i=0}^{\lceil\log(n)\rceil} b_i 2^i} \bmod N = 2^j \bmod N.$$

In turn, one computation of the form $m := 2^j \bmod N$ can be done using $\Theta(\log(n)\log(N)^2)$ bit operations while using $\Theta(\log(N))$ bits of scratch space (not accounting for the storage of the lookup table).

Note that the only operation with a time requirement that does not contain a term of the form $\log(N')^2$ or $\log(N)^2$ is the computation of the index $j_i := S_{i-1} \bmod n$ in Lines 13 and 21, but that the number of bit operations required for this operation is actually smaller since $\log(n) < \log(N) \approx \log(N')$. Thus, in the following computation of Eval and TDEval's cumulative memory complexity we will use $\log(N)^2 \approx \log(N')^2$, the number of bit operations required for one group operation in $\mathbb{QR}_{N'}$ and $\mathbb{Z}_N$, as our unit of time. Accordingly, group operations and hash evaluations require one time unit, exponentiation $\log(N)$ units, and computing $2^j \bmod N$ requires $\log(n)$ units.

Cumulative Memory Complexity of Eval. As algorithm Eval's scratch space requirements are of order $\log(N')$ (see Table 2), its memory usage is dominated by storing the group elements $W_0, \ldots, W_n$ amounting to $\Theta(n\log(N'))$ bits of memory sustained throughout the whole second phase, i.e., Lines 12–14. In the first phase of the evaluation the algorithm computes n group operations, in the second n hash evaluations and n reductions modulo n. As a consequence, its cumulative memory complexity in terms of bit storage and group operations is of order

$$\mathsf{cc}_{\mathsf{mem}}(\mathsf{Eval}(\mathsf{pp}, x)) \in \Theta(n\log(N') \cdot n) = \Theta(n^2 \log(N')).$$

Cumulative Memory Complexity of TDEval. In order to compute the W_i at any point in time, TDEval only has to keep track of at most two group elements and one exponent m_i. Further, it only requires scratch space of order $\log(N')$. Thus, its memory is dominated by the lookup table used to perform reductions modulo N which has a size of order $\Theta(\log(n)\log(N'))$ bits. In the first phase of the evaluation, the algorithm performs one reduction modulo N as well as one exponentiation in $\mathbb{QR}_{N'}$. In each iteration in the of the second phase of the evaluation TDEval has to perform the same operations as well as an additional reduction modulo n (line 21), thus requiring essentially $\log(N) + \log(n)$ group operations. Thus, its cumulative memory complexity in terms of bit storage and group operations amounts to

$$\begin{aligned}\mathsf{cc_{mem}}(\mathsf{TDEval}(\mathsf{pp}, \mathsf{td}, x)) &\in \Theta(\log(n)\log(N') \cdot n(\log(N') + \log(n)))\\ &= \Theta(n \log(N')^2 \log(n)).\end{aligned}$$

Summing up, TdScrypt is a $(n^2 \log(N'), n\log(n)\log(N')^2)$-TMHF.

B Bounding the Entries of A

In this section we analyze the maximal size that the entries of matrix $A \in \mathbb{Z}^{\ell \times m}$ induced by the query behavior of $\mathcal{A}_1$ to the group-operation oracle $\mathcal{G}$ can take. Recall that (as discussed in more detail in Section 5.1 and made formal in Appendix C.2) our encoder assigns all labels appearing as input to or output of $\mathcal{G}$ an algebraic representation of the form $\sum_{i=1}^{m} \boldsymbol{a_i}\boldsymbol{x_i}$ with $\boldsymbol{a} \in \mathbb{Z}^m$ which is stored in table T. Here, the indeterminates $\boldsymbol{x}$ correspond to the labels being queried out of the blue. Matrix A collects the representations of labels corresponding to challenges that are successfully answered within t rounds of parallel queries.

In the following we characterize the set of representations that possibly can be added to the table T. More precisely, using a simple abstraction we inductively define sets R_t that for $t \in \mathbb{N}^+$ contain all $\boldsymbol{a} \in \mathbb{Z}^m$ that can be added to T within t rounds of queries to $\mathcal{G}$ with unbounded parallelism.

As a special case, R_0 is to be understood as all representations corresponding to labels that are used as *input* to $\mathcal{G}$ within the first round of queries. Note that all of these labels must be out of the blue. The ith label queried out of the blue corresponds to indeterminate $\boldsymbol{x}_i$. Thus, its representation is the ith unit vector $\boldsymbol{e_i} \in \mathbb{Z}^m$. We define

$$R_0 := \{\pm \boldsymbol{e_i} \mid i \in \{1, \ldots, m\}\} \cup \{\boldsymbol{0}\}.$$

Note that R_0 contains the representations of all labels appearing out of the blue (irrespective of the actual round in which they are first queried) as well the ones of their inverses and the neutral element. This choice of R_0 overestimates the capabilities of $\mathcal{A}_1$ but will allow us to derive a clean characterization of the sequence of sets R_t.

We now turn to the definition of set R_t for $t \geq 1$. The representation of the label output in response to a query to $\mathcal{G}$ is given by the sum, difference, or inverse of the label(s) used as input, depending on whether the used operation was $+$, $-$, or inv. Accordingly, for $t \geq 1$ we define

$$R_t := \{\boldsymbol{a_1} \pm \boldsymbol{a_2} \mid \boldsymbol{a_1}, \boldsymbol{a_2} \in R_{t-1}\} \cup R_{t-1},$$

i.e., we add to R_{t-1} all vectors that can be obtained by adding or subtracting two vectors that are reachable within $t-1$ rounds. Note that this in particular captures inversion as $\mathbf{0} \in R_{t-1}$. Further, any upper bound on the elements of R_t also applies to the entries of A, since the inclusion of additional elements in R_0 only improves on $\mathcal{A}_1$'s computational capabilities.

To bound the elements of R_t note that with every additional query to $\mathcal{G}$ every vector entry can at most double. This immediately implies $\|\boldsymbol{a}\|_\infty \leq 2^t$ for all $\boldsymbol{a} \in R_t$. While this observation is already sufficient for the proof of our lower bound, a more careful analysis yields the following exact characterization of R_t in terms of the one-norm.

Lemma 6. *Let $m, t \in \mathbb{N}$ and R_t be defined as above. Then for $\boldsymbol{a} \in \mathbb{Z}^m$ we have*

$$\boldsymbol{a} \in R_t \quad \Longleftrightarrow \quad \|\boldsymbol{a}\|_1 \leq 2^t.$$

Proof. We prove the result by induction on t. The statement holds for $t = 0$ as we have $R_0 = \{\pm \boldsymbol{e_i} \mid i \in \{1, \dots, m\}\} \cup \{\mathbf{0}\}$, the set of integer vectors with one-norm bounded by 1.

Case $\Longrightarrow$: Assume the statement is true for all values up to t. First, consider $\boldsymbol{a} \in \mathbb{Z}^m$ with $\|\boldsymbol{a}\|_1 > 2^{t+1}$ and consider arbitrary $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{Z}^m$ satisfying $\boldsymbol{a} = \boldsymbol{b} \pm \boldsymbol{c}$. We have $2^{t+1} < \|\boldsymbol{a}\|_1 \leq \|\boldsymbol{b}\|_1 + \|\boldsymbol{c}\|_1$ implying that $\|\boldsymbol{b}\|_1 > 2^t$ or $\|\boldsymbol{c}\|_1 > 2^t$. By the induction hypothesis, one of the two vectors cannot have been computed within t group operations with unbounded parallelism. In turn, $\boldsymbol{a}$ cannot have been computed within $t+1$ group operations.

Case $\Longleftarrow$: Consider $\boldsymbol{a} \in \mathbb{Z}^m$ with $\|\boldsymbol{a}\|_1 \leq 2^{t+1}$ and let $S_1, S_2 \subseteq \{1, \dots, m\}$ be sets of (almost) equal size jointly containing all odd entries of $\boldsymbol{a}$. More precisely, we require $|S_2| \leq |S_1| \leq |S_2| + 1$ and that $\boldsymbol{a}_i$ is odd exactly if $i \in S_1 \cup S_2$. We define vectors $\boldsymbol{b}, \boldsymbol{c} \in \mathbb{N}^m$ as

$$\boldsymbol{b}_i = \begin{cases} \operatorname{sgn}(\boldsymbol{a}_i) \cdot \lceil |\boldsymbol{a}_i|/2 \rceil & \text{if } i \in S_1 \\ \operatorname{sgn}(\boldsymbol{a}_i) \cdot \lfloor |\boldsymbol{a}_i|/2 \rfloor & \text{if } i \in S_2 \\ \boldsymbol{a}_i/2 & \text{else} \end{cases} \quad \text{and} \quad \boldsymbol{c}_i = \begin{cases} \operatorname{sgn}(\boldsymbol{a}_i) \cdot \lfloor |\boldsymbol{a}_i|/2 \rfloor & \text{if } i \in S_1 \\ \operatorname{sgn}(\boldsymbol{a}_i) \cdot \lceil |\boldsymbol{a}_i|/2 \rceil & \text{if } i \in S_2 \\ \boldsymbol{a}_i/2 & \text{else} \end{cases},$$

where sgn denotes the sign of an integer. Note that by definition of $\boldsymbol{b}$ and $\boldsymbol{c}$ it holds that $\boldsymbol{a} = \boldsymbol{b} + \boldsymbol{c}$. Further, we have that

$$\begin{aligned} \|\boldsymbol{b}\|_1 = \sum_{i=1}^m |\boldsymbol{b}_i| &= 1/2 \cdot (|S_1| - |S_2|) + \sum_{i=1}^m |\boldsymbol{a}_i|/2 \\ &= 1/2 \cdot ((|S_1| - |S_2|) + \|\boldsymbol{a}\|_1). \end{aligned}$$

If $\|\boldsymbol{a}\|_1$ is even we have $|S_1| = |S_2|$ and thus $2^t \geq \|\boldsymbol{a}\|_1/2 = \|\boldsymbol{b}\|_1 = \|\boldsymbol{c}\|_1$. On the other hand, if $\|\boldsymbol{a}\|_1$ is odd, we have $2^{t+1} \geq \|\boldsymbol{a}\|_1 + 1$ and $|S_1| = |S_2| + 1$ which implies $2^t \geq 1/2(\|a\|_1 + 1) = \|\boldsymbol{b}\|_1 = \|\boldsymbol{c}\|_1 + 1$. Thus, by the induction hypothesis we have $\boldsymbol{b}, \boldsymbol{c} \in R_t$ and in turn $\boldsymbol{a} \in R_{t+1}$. □

C Proof of Claim 5

For the full proof, we first state Enc's pseudocode, explain it, and finally analyze its encoding size and success probability. We omit Dec's pseudocode since it follows naturally from Enc.

C.1 Notation and Data Structures

In the following, let $\mathrm{Im}(f)$ denote the image of the function f, and let $\mathfrak{x}_1, \mathfrak{x}_2, \ldots$ be indeterminates[14] in $\mathbb{Z}$. Further, we use the shorthand $\hat{\circ} := \circ \bmod N$ where $\circ$ might be a vector, matrix, indeterminate, etc. To signify when and why Enc aborts, we write $\mathsf{abort}_{\mathsf{reason}}$.

During Enc's execution, it constructs five datastructures: I, a list of hints; Ξ, a first-in-first-out queue; T, a table; S, a map of substitutions; and C, a list of *collision pointers*. We use the notation $x \to Y$ to denote that element x is appended or pushed to datastructure Y.

List of Hints I. Dec needs a *hint* to recognize when $\mathcal{A}_1(\mathsf{st}_0, j)$ queries the label corresponding to $2^j w$ to h for the first time. We are guaranteed that at least $n/2$ instances of $\mathcal{A}_1$ perform this query within t steps and that matrix $A \in \mathbb{Z}^{(n/2)\times m}$ induced by $\mathcal{A}_1$'s queries has $\mathsf{rank}_{\mathsf{min}} \leq m$. In the remainder of this proof, we will not work with A but with a different matrix $A' \in \mathbb{Z}^{\mathsf{rank}_{\mathsf{min}} \times m}$ because encoding A would be too costly for the compression argument. A' consists of the rows of A that contain a pivot element in H, the Hermite normal form (cf. Definition 7) of A. Note that $\mathrm{rank}(A') = \mathrm{rank}(A) = \mathsf{rank}_{\mathsf{min}}$ which follows from the HNF. Indeed, the HNF has $\mathsf{rank}_{\mathsf{min}}$ non-zero columns and hence $\mathsf{rank}_{\mathsf{min}}$ pivot elements. Simple, inefficient HNF algorithms iterate over the rows once, top to bottom, while only touching rows that will contain a pivot in the end.[15]

So for Dec to reconstruct A' from $\mathcal{A}_1$'s queries, it needs $\mathsf{rank}_{\mathsf{min}}$ many hints. In the remainder of this proof, we will not distinguish between A and A' but simply call it $A \in \mathbb{Z}^{\mathsf{rank}_{\mathsf{min}} \times m}$.

Queue Ξ. The queue contains labels that Dec needs to, e.g., correctly respond to $\mathcal{A}_1$'s queries. Enc fills Ξ with this information during its execution and, in the final step, appends Ξ to the encoding. The elements pushed to the queue vary in length, but this does not lead to ambiguities during decoding. In particular, we ensure that Dec always knows this length in advance before popping an element from the queue.

Table T. Intuitively, T helps Dec answer repeat queries or queries where previous responses uniquely determine the answer. It does so by mapping labels to algebraic representations over $\mathbb{Z}_N$, the domain of Σ, analogously to Section 5.1. In addition, T also stores an algebraic representation over $\mathbb{Z}$, enabling us to connect the compression argument to the query behavior of $\mathcal{A}_1$. This is necessary since $\mathcal{A}_1$ does not know to the group order N and therefore its queries must be interpreted over $\mathbb{Z}$. To summarize, T contains three columns: The rightmost column contains labels $\sigma_i \in \mathrm{Im}(\Sigma)$; the leftmost

[14]As we will explain later, in contrast to the proof sketch in Section 5, there is a minor technical difference between the indeterminates $\mathfrak{x}_i$ and the entries x_i belonging to the vector $\boldsymbol{x}$ of the system $A\boldsymbol{x} = \boldsymbol{b}$.

[15]The algorithm starts with $A = A_0$ and iteratively computes $A_i = \left(\begin{smallmatrix} * & 0 \\ * & A_{i+1} \end{smallmatrix}\right)$ where 0 is an all-zero and $*$ is an arbitrary submatrix. In every iteration, it computes the GCD of A_{i+1}'s topmost row, swaps the GCD to A_{i+1}'s leftmost column, and zeros out all elements to the right of it. This can be realized using elementary column operations and results in a lower-triangular matrix H. Then the algorithm performs further computation to ensure other HNF conditions. However, this does not change the shape of H, so it is not relevant for our argument.

column contains linear terms in the indeterminates $\hat{\mathfrak{x}}_j$ over $\mathbb{Z}_N$ that represent a unique algebraic representation of σ_i; and the middle column contains an algebraic (but not necessarily unique) representation of σ_i in the form of linear terms in the indeterminates $\mathfrak{x}_j$ over $\mathbb{Z}$ (see Table 3 for an example).

Representation over $\mathbb{Z}_N$	Representation over $\mathbb{Z}$	Label
	$\cdots$	
$2\hat{\mathfrak{x}}_{10}$	$2\mathfrak{x}_{10}$	σ_{20}
	$\cdots$	
$4\hat{\mathfrak{x}}_5$	$\mathfrak{x}_5 + \mathfrak{x}_7$	σ_{32}
	$\cdots$	

Table 3: Exemplary table T. Note that the second populated row shows the case where a collision has been resolved with $\hat{\mathfrak{x}}_7 = 3\hat{\mathfrak{x}}_5$.

We will ensure that the leftmost (i.e., the representation over $\mathbb{Z}_N$) and the rightmost (i.e., the label) columns will not contain duplicates. This allows for convenient notation regarding T. We write $\sigma_i \in T$ to check if there exists a row containing label σ_i and a row can be selected by either $T(\sigma)$ or $T(x)$ where x is an algebraic representation over $\mathbb{Z}_N$ (which is unique and thus unambiguous). A particular column is denoted by the subscripts $\mathbb{Z}_N$, $\mathbb{Z}$, and $\mathcal{L}$. For example, in Table 3, $T_{\mathbb{Z}_N}(\sigma_{32})$ returns $4\hat{\mathfrak{x}}_5$.

Mapping of Substitutions S. Recall that T must not contain any duplicate labels. However, $\mathcal{A}_1$ might perform queries such that it arrives at the same label in two different ways. So there exist two distinct algebraic representations over $\mathbb{Z}_N$ for the same label—a *collision*. Instead of adding a duplicate row, Enc uses the collision to express one indeterminate $\hat{\mathfrak{x}}_i$ in terms of its preceding indeterminates $\hat{\mathfrak{x}}_j$, $j < i$. That is, $\hat{\mathfrak{x}}_i = \sum_{j<i} c_j \hat{\mathfrak{x}}_j$ with $c_j \in \mathbb{Z}_N$. Subsequently, it replaces $\hat{\mathfrak{x}}_i$ (over $\mathbb{Z}_N$) in the leftmost column of T. Note that it cannot replace the corresponding indeterminate $\mathfrak{x}_i$ (over $\mathbb{Z}$) in the middle column. To handle this, S maps $\hat{\mathfrak{x}}_i$ to $\sum_{j<i} c_j \hat{\mathfrak{x}}_j$ keeping track of all substiutions. We write $\hat{\mathfrak{x}}_i \in S$ to check whether a mapping exists.

List of Collision Pointers C. Enc must tell Dec when the aforementioned collisions take place since Dec cannot detect them on its own. Thus, it stores C, the list of collision pointers, in the encoding. To this end, Enc encodes C's length $|C| \leq nQ$ using $\log n + \log Q$ bits followed by $|C|$ pointers which indicate when a collision occurs and how to resolve it.

C.2 Encoder

Pseudocode. The pseudocode of Enc is given in Figure 2.

Explanation. First, Enc runs $\mathcal{A}_0$ to get the state st_0. Recall that Enc knows Σ and h, so it can simulate $\mathcal{G}$ and h for $\mathcal{A}_0$ perfectly. Note that Enc does not perform any bookkeeping regarding the oracles yet.

Then, it runs $\mathcal{A}_1$ with state st_0 on every challenge $j \in [0, n)$ for at most t rounds. It executes all instances of $\mathcal{A}_1$ in parallel in lockstep. That is, Enc waits until all instances

Figure 2: Enc given input Σ and auxiliary inputs N, h, and $\mathcal{W}$.

```
01 Compute $\mathsf{st}_0 := \mathcal{A}_0^{\Sigma,\mathsf{h}}(\Sigma(1), \Sigma(w))$.
02 Initialize empty data structures $I$, $\Xi$, $T$, $S$, and $C$.
03 For all $j \in [0, n)$, run $\mathcal{A}_1(\mathsf{st}_0, j)$ in lockstep for $t$ rounds while simulating $\mathcal{G}$ (Figure 3)
   and $\mathsf{h}$ (Figure 4).
04 Using the equations $T_{\mathbb{Z}}(\tilde{\sigma}_j) = w^{2^j}$, define a system of equations $A\boldsymbol{x} = \boldsymbol{b}$ where
   $A \in \mathbb{Z}^{\mathsf{rank}_{\mathsf{min}} \times m}$ and function $f$ that maps every column $i$ to the indeterminate $\mathfrak{x}_{f(i)}$.
   ($m$ and $f$ are described in the explanation below).
05 Compute the Hermite normal form of $A$ by $\mathsf{HNF}(A) = (H, U)$.
06 If any $\hat{h}_{ij} = 0$ but $h_{ij} \neq 0$, $\mathsf{abort}_{\mathsf{hnf}}$.
07 Solve the system $\hat{A}\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}}$ such that $\hat{x}_i = \Sigma^{-1}(T_{\mathcal{L}}(\hat{\mathfrak{x}}_{f(i)}))$. During this, if any element
   of $\mathbb{Z}_N$ cannot be inverted, $\mathsf{abort}_{\mathsf{sol}}$:
08    Consider the equivalent system $\hat{H}\hat{\boldsymbol{y}} = \hat{\boldsymbol{b}}$ where $\hat{\boldsymbol{y}} = \hat{U}^{-1}\hat{\boldsymbol{x}}$.
09    For $i \leq \mathsf{rank}_{\mathsf{min}}$, fix the entries $\hat{y}_i \in \hat{\boldsymbol{y}}$ by the triangular shape of $\hat{H}$.
10    For every $i \leq f(m - \mathsf{rank}_{\mathsf{min}})$, if $\hat{\mathfrak{x}}_i \notin S$, push $\Sigma^{-1}(T_{\mathcal{L}}(\hat{\mathfrak{x}}_i)) \to \Xi$. This fixes the first
      $m - \mathsf{rank}_{\mathsf{min}}$ values of $\hat{\boldsymbol{x}}_i$.
11    Given these constraints on $\hat{\boldsymbol{y}}$ and $\hat{\boldsymbol{x}}$, solve $\hat{U}\boldsymbol{y} = \boldsymbol{x}$.
12 Substitute every $\hat{x}_i$ for $\hat{\mathfrak{x}}_{f(i)}$ in $T$.
13 For every remaining $\hat{\mathfrak{x}}_i \in T_{\mathbb{Z}_N}$, push $\Sigma^{-1}(T_{\mathcal{L}}(\hat{\mathfrak{x}}_i)) \to \Xi$.
14 Iterate over all $\sigma \in \mathrm{Im}(\Sigma)$ in lexicographical order and for any $\sigma_i \notin T$, push
   $\Sigma^{-1}(\sigma_i) \to \Xi$.
15 Output the encoding comprised of $\mathrm{Im}(\Sigma)$, $I$, $C$, $\mathsf{st}_0$, and $\Xi$.
```

have made queries in the current round before answering them all at the same time. Now Enc performs additional bookkeeping in order to aid Dec in simulating $\mathcal{G}$ and h correctly.

Enc simulates h (Figure 4) by forwarding the queries to h. It does not need to store the response in Ξ since Dec also gets h as an auxiliary input. However, Dec cannot recognize when $\mathcal{A}_1(\mathsf{st}_0, j)$ queries the label σ corresponding to the exponent $2^j w$ for the first time. Thus, it adds (j, i) to the list of hints I and also sets the variable $\tilde{\sigma}_j = \sigma$ for later use in Figure 2.

The bookkeeping for $\mathcal{G}$ is more elaborate. Enc responds to queries using Σ as before, but also populates the table T. For every query input $\sigma_i \notin T$, $i \in \{1, 2\}$, it adds a row containing the label and fresh indeterminate $\mathfrak{x}_j$. For the query output σ_3, Enc pushes it to Ξ if it is a new label, so that Dec can correctly simulate $\mathcal{G}$. Then, in any case, Enc computes the algebraic representations of σ_3 over $\mathbb{Z}_N$ and $\mathbb{Z}$. The representations are derived by adding/subtracting the representations of the query inputs σ_1 and σ_2. Here, Enc might encounter a collision, i.e., $\sigma_3 \in T$ and the computed algebraic representation over $\mathbb{Z}_N$ differs from the one assigned to σ_3 in the table.

A collision is beneficial for the compression since it gives a non-trivial relation between the indeterminates. In particular, Enc attempts to eliminate one indeterminate in T. To this end, it tries to express $\hat{\mathfrak{x}}_k$ (where k is the maximal index within the colliding representations) in terms of the other indeterminates. This might fail since not every element in $\mathbb{Z}_N$ has an inverse so Enc aborts in such cases. Enc replaces $\hat{\mathfrak{x}}_k$ in T and

Figure 3: Answering $\mathcal{A}_1(\mathsf{st}_0, j)$'s ith query $(\circ \in \{+, -, \mathsf{inv}\}, \sigma_1, \sigma_2)$ to $\mathcal{G}$.

```
01 If $(\sigma_1, \sigma_2) \notin \mathrm{Im}(\Sigma) \times \mathrm{Im}(\Sigma)$, respond with $\perp$.
02 Else:
03   If $\circ \in \{+, -\}$:
04     Respond with $\sigma_3 = \Sigma\left(\Sigma^{-1}(\sigma_1) \circ \Sigma^{-1}(\sigma_2)\right)$.
05     For $i \in \{1, 2\}$, if $\sigma_i \notin T$, choose a fresh indeterminate $\mathfrak{x}_j$ and append $(\hat{\mathfrak{x}}_i, \mathfrak{x}_i, \sigma_i) \to T$.
06     If $\sigma_3 \notin T$, add the $\left(T_{\mathbb{Z}_N}(\sigma_1) \circ T_{\mathbb{Z}_N}(\sigma_2) \bmod N, T_{\mathbb{Z}}(\sigma_1) \circ T_{\mathbb{Z}}(\sigma_2), \sigma_3\right) \to T$ and push $\sigma_3 \to \Xi$.
07     Else, if $\sigma_3 \in T$ but $T_{\mathbb{Z}_N}(\sigma_1) \circ T_{\mathbb{Z}_N}(\sigma_2) \neq T_{\mathbb{Z}_N}(\sigma_3) \bmod N$:
08       Append the collision $(j, i, T(\sigma_3)) \to C$.
09       Rearrange $T_{\mathbb{Z}_N}(\sigma_1) \circ T_{\mathbb{Z}_N}(\sigma_2) = T_{\mathbb{Z}_N}(\sigma_3) \bmod N$ to be of the form $\hat{\mathfrak{x}}_k = \Sigma_{l<k} c_l \hat{\mathfrak{x}}_l$ where $k$ is the largest index within the equation and $c_l \in \mathbb{Z}_N$. If rearranging fails (i.e., an element of $\mathbb{Z}_N$ cannot be inverted), $\mathsf{abort}_{\mathsf{col}}$.
10       Add $(\hat{\mathfrak{x}}_k, \Sigma_{l<k} c_l \hat{\mathfrak{x}}_l) \to S$.
11       Substitute $\Sigma_{l<k} c_l \hat{\mathfrak{x}}_l$ for $\hat{\mathfrak{x}}_k$ in the leftmost column $T_{\mathbb{Z}_N}$.
12   Else, $\circ = \mathsf{inv}$, so respond to an inversion query in the natural manner analogously to the case $\circ \in \{+, -\}$.
```

Figure 4: Answering $\mathcal{A}_1(\mathsf{st}_0, j)$'s ith query $(\sigma, \ldots)$ to h.

```
01 Respond with $\mathsf{h}(\sigma, \ldots) \to \Xi$.
02 If $\Sigma(w^{2^j}) = \sigma$, set $\tilde{\sigma}_j := \sigma$ and append $(j, i) \to I$.
```

also keeps track of this replacement by adding the appropriate mapping to the list of substitutions S.

After executing all instances of $\mathcal{A}_1$ for at most t rounds, Enc (and also Dec) holds the table T, substitutions S, and $\mathsf{rank}_{\mathsf{min}}$ many labels $\tilde{\sigma}_j = \Sigma(w^{2^j})$. It chooses the first $\mathsf{rank}_{\mathsf{min}}$ of these labels, and creates a system of equations using the column $T_{\mathbb{Z}_N}$, i.e.,

$$\begin{aligned}
a_{11}\mathfrak{x}_{f(1)} + a_{12}\mathfrak{x}_{f(2)} + \cdots + a_{1m}\mathfrak{x}_{f(m)} &= 2^{j_1} w \\
a_{21}\mathfrak{x}_{f(1)} + a_{22}\mathfrak{x}_{f(2)} + \cdots + a_{2m}\mathfrak{x}_{f(m)} &= 2^{j_2} w \\
&\vdots \\
a_{\mathsf{rank}_{\mathsf{min}}1}\mathfrak{x}_{f(1)} + a_{\mathsf{rank}_{\mathsf{min}}2}\mathfrak{x}_{f(2)} + \cdots + a_{rm}\mathfrak{x}_{f(m)} &= 2^{j_{\mathsf{rank}_{\mathsf{min}}}} w
\end{aligned}$$

which can be viewed as a matrix $A \in \mathbb{Z}^{\mathsf{rank}_{\mathsf{min}} \times m}$ and vector $\boldsymbol{b} \in \mathbb{Z}^m$. Here, m is the number of indeterminates $\hat{\mathfrak{x}}_i$ that occur in the system of equations. Note that these indeterminates might only be a subset of all indeterminates, so the function f translates between column indices and indeterminate subscripts. So $i \leq f(i)$ and f might skip some indices.

Next, Enc computes the Hermite normal form of A resulting in a lower-triangular matrix H and unimodular (i.e., invertible over $\mathbb{Z}$) matrix U such that $AU = H$. Recall that H is a lower triangular matrix with $\mathsf{rank}_{\mathsf{min}}$ non-zero columns on the left and the height of these columns is strictly decreasing. Enc aborts when H and $\hat{H}$ differ in their shape, that is, $\hat{H}$ has additional zero entries because an entry of H was a multiple of N. On a high level, this step is necessary to connect our compression argument, which mainly works in $\mathbb{Z}_N$, with the query behavior of $\mathcal{A}_1$, which must be interpreted over $\mathbb{Z}$ in hidden order groups.

Enc solves the system $\hat{A}\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}}$ by solving $\hat{H}\hat{\boldsymbol{y}} = \hat{\boldsymbol{b}}$ where $\hat{\boldsymbol{x}} = \hat{U}\hat{\boldsymbol{y}}$. The first $\mathsf{rank}_{\mathsf{min}}$ entries of $\boldsymbol{y}$ are fixed because of the triangular shape of $\hat{H}$ and because every element can be inverted (otherwise, Enc aborts). If $\mathsf{rank}_{\mathsf{min}} = m$, then this leads to a unique solution. Otherwise, $m - \mathsf{rank}_{\mathsf{min}}$ entries in $\hat{\boldsymbol{y}}$ are not uniquely defined and there might be multiple solutions for $\hat{\boldsymbol{x}}$. Hence, Enc needs to add additional constraints. It does so by fixing the first $m - \mathsf{rank}_{\mathsf{min}}$ entries of $\hat{\boldsymbol{x}}$. To this end, Enc pushes the discrete logarithm of all $\hat{\mathfrak{x}}_i$ with $i \leq f(m - \mathsf{rank}_{\mathsf{min}})$ to Ξ whenever $\hat{\mathfrak{x}}_i$ is not in S. Because if this is the case, then Dec can compute $\hat{\mathfrak{x}}_i$ from the preceding $\hat{\mathfrak{x}}_j,\ j < i$.

Now that the system $\hat{U}\hat{\boldsymbol{y}} = \hat{\boldsymbol{x}}$ is uniquely determined and can be solved to yield $\hat{\boldsymbol{x}}$. As a result, up to $\mathsf{rank}_{\mathsf{min}}$ many indeterminates in T might be replaced. Note that only "up to" because some might have already been eliminated due to collisions.

Still, T might contain some indeterminates $\hat{\mathfrak{x}}_i$, so Enc pushes the remaining $\hat{\mathfrak{x}}_i$ in T to Ξ. Finally, the column $T_{\mathbb{Z}_N}$ only contains constants which correspond to the discrete logarithm of the labels. T does not contain all labels so Enc pushes the remaining discrete logarithms to Ξ in lexicorgraphical order of the labels.

The encoding output by Enc contains the image $\mathrm{Im}(\Sigma)$, the list of hints I and of collisions C, the state st_0 output by $\mathcal{A}_0$, and Ξ. Dec follows from the description of Enc and this explanation so it is omitted.

C.3 Encoding Size

Let us analyze the size of Enc's encoding.

- $\mathrm{Im}(\Sigma)$ which requires $\log \binom{|\mathcal{L}|}{N}$.
- I is a list of fixed length $\mathsf{rank}_{\mathsf{min}}$ and every element is a pointer to a query. Identifying a query requires $\log n$ bits to specify the challenge $j \in [0, n)$ and $\log Q$ bits to indicate the query's index within the execution $\mathcal{A}_1(\mathsf{st}_0, j)$. Thus, $\|I\| = \mathsf{rank}_{\mathsf{min}}(\log n + \log Q)$.
- C is a list of variable length and every element contains a pointer to a query where a collision happens as well as a row of T. Every row of T can be identified by the query in which it was added plus $\lceil \log 3 \rceil = 2$ bits as every query adds at most three rows. So $\|C\| = (\log n + \log Q) + |C|(2(\log n + \log Q) + 2)$ where $(\log n + \log Q)$ encodes the variable length $|C| \leq nQ$. We do not have any non-trivial bound on $|C|$, but this will not be an issue for our analysis.
- $\|\mathsf{st}_0\| = M$ by definition.
- $\|\Xi\|$ is not bounded as well, but, again, this is not problematic for our analysis.

Now, to handle the variable length of C and Ξ, we compare the size of this encoding to the naive encoding of Σ. The naive encoding also encodes $\mathrm{Im}(\Sigma)$ with $\log \binom{|\mathcal{L}|}{N}$ bits and then uses $\log(N!)$ bits to assign discrete logarithms to all labels in the image. Hence, we can ignore the size of the image and will focus on how much Enc pays (in terms of bits) for every discrete logarithm. We will show that any element contained in C and Ξ leads to one discrete logarithm and that it is at most as expensive as in the naive encoding.

This is directly related to how much is paid for every row of T as every row of T yields one discrete logarithm in the end.

- Unexpected query inputs require 0 bits initially as they are assigned an indeterminate $\hat{\mathfrak{x}}_i$.
- Responses without collision (stored in Ξ) cost $\log(N - r)$ bits where r is the number of rows in T (and thus the number of labels already assigned) at the time of the query.
- Responses with collision (stored in C) cost $2(\log n + \log Q) + 2$ bits but at the same time eliminate one indeterminate $\hat{\mathfrak{x}}_i$ in T.

Thus, responses without collisions cost as much as in the naive encoding and collisions are cheaper because $2(\log n + \log Q) + 2 < \log(N) - 1$ for λ large enough. Indeed, n and Q are polynomial, but N is superpolynomial in λ. So far, our encoding is at most as expensive as the naive one without accounting for the indeterminates and additional bookkeeping yet.

When solving the system of equations, we get $\mathsf{rank}_{\mathsf{min}}$ many indeterminates for the cost of a hint of $\log n + \log Q$ bits. However, we might have already resolved all of those indeterminates by collisions, so in the worst case we pay $3(\log n + \log Q) + 2$ bits for $\mathsf{rank}_{\mathsf{min}}$ many indeterminates. By the same reasoning as above, $3(\log n + \log Q) + 2 < \log(N) - 1$, so we have a net profit of at least $0 < \log(N) - 1 - (3(\log n + \log Q) + 2)$ bits for $\mathsf{rank}_{\mathsf{min}}$ many indeterminates.

In the course of solving the system of equations, we might have to fix some values of $\hat{x}_i$ by fixing some indeterminates (stored in Ξ). Since we are fixing these indeterminates top-down according to the rows of T, we pay at most $\log(N - r)$ bits per indeterminant—the same as the naive algorithm. Note that we account for collisions in this step by using the mapping S. This ensures that we do not overpay for any indeterminate that has already been resolved by a collision. The remaining indeterminants and discrete logarithms (both stored in Ξ) cost at most $\log(N - r)$ bits again.

To summarize, we pay fewer bits for $\mathsf{rank}_{\mathsf{min}}$ discrete logarithms and, accounting for the additional bookkeeping data, pay at most

$$\underbrace{\log\binom{|\mathcal{L}|}{N} + \log(N!)}_{\text{Naive encoding of } \Sigma} + \underbrace{\log n + \log Q}_{\text{Encoding of } |C|} + \underbrace{M}_{\|\mathsf{st}_0\|} - \underbrace{\mathsf{rank}_{\mathsf{min}}(\log(N) - 3(\log n + \log Q) - 3)}_{\text{Profit over the naive encoding}}$$

bits in total relative to the naive algorithm.

C.4 Success Probability

$\mathsf{abort}_{\mathsf{hnf}}$ and $\mathsf{abort}_{\mathsf{sol}}$ do not happen for any element in bad as guaranteed by Claim 1. However, the lemma does not cover $\mathsf{abort}_{\mathsf{col}}$. Noting that $\mathsf{abort}_{\mathsf{col}}$ may be used to factor

N, we modify Claim 1 to account for it as well. So Enc never aborts, so it is successful with probability 1.

The discussion in this section gives a proof of Claim 5. $\square$