On the Memory-Tightness of Hashed ElGamal

Ashrujit Ghoshal(✉) and Stefano Tessaro(✉)

Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, USA
{ashrujit,tessaro}@cs.washington.edu

Abstract. We study the memory-tightness of security reductions in public-key cryptography, focusing in particular on Hashed ElGamal. We prove that any *straightline* (i.e., without rewinding) black-box reduction needs memory which grows linearly with the number of queries of the adversary it has access to, as long as this reduction treats the underlying group generically. This makes progress towards proving a conjecture by Auerbach *et al.* (CRYPTO 2017), and is also the first lower bound on memory-tightness for a concrete cryptographic scheme (as opposed to generalized reductions across security notions). Our proof relies on compression arguments in the generic group model.

Keywords: Public-key cryptography · Memory-tightness · Lower bounds · Generic group model · Foundations · Compression arguments

1 Introduction

Security proofs rely on *reductions*, i.e., they show how to transform an adversary $\mathcal{A}$ breaking a scheme into an adversary $\mathcal{B}$ solving some underlying assumed-to-be-hard problem. Generally, the reduction ought to be *tight* – the resources used by $\mathcal{B}$, as well as the attained advantage, should be as close as possible to those of $\mathcal{A}$. Indeed, the more resources $\mathcal{B}$ needs, or the smaller its advantage, the weaker the reduction becomes.

Auerbach *et al.* [2] were the first to explicitly point out that *memory* resources have been ignored in reductions, and that this leads to a loss of quality in security results. Indeed, it is conceivable that $\mathcal{A}$'s memory is naturally bounded (say, at most 2^{64} bits), and the underlying problem is very sensitive to memory. For example, the best-known algorithm for discrete logarithms in a 4096-bit prime field runs in time (roughly) 2^{156} using memory 2^{80}. With less memory, the best algorithm is the generic one, requiring time $\Theta(\sqrt{p}) \approx 2^{2048}$. Therefore, if $\mathcal{B}$ also uses memory at most 2^{64}, we can infer a larger lower bound on the necessary time complexity for $\mathcal{A}$ to break the scheme, compared to the case where $\mathcal{B}$ uses 2^{100} bits instead.

What can be memory-tight? One should therefore target reductions that are *memory-tight*, i.e., the memory usage of $\mathcal{B}$ is similar to that of $\mathcal{A}$.[1] The work

[1] Generally, $\mathcal{B} = \mathcal{R}^{\mathcal{A}}$ for a black-box reduction $\mathcal{R}$, and one imposes the slightly stronger requirement that $\mathcal{R}$ uses small memory, independent of that of $\mathcal{A}$.

A. Canteaut and Y. Ishai (Eds.): EUROCRYPT 2020, LNCS 12106, pp. 33–62, 2020.
https://doi.org/10.1007/978-3-030-45724-2_2

of Auerbach *et al.* [2], and its follow-up by Wang *et al.* [13], pioneered the study of memory-tight reductions. In particular, and most relevant to this work, they show *negative* results (i.e., that certain reductions cannot be memory tight) using *streaming lower bounds.*

Still, these lower bounds are tailored at general notions (e.g., single- to multi-challenge reductions), and lower bounds follow from a natural connection with classical frequency problems on streams. This paper tackles the more ambitious question of proving impossibility of memory-tight reductions for concrete *schemes*, especially those based on algebraic structures. This was left as an open problem by prior works.

Hashed ElGamal. Motivated by a concrete open question posed in [2], we consider here the CCA-security of Hashed ElGamal. In its KEM variant, the scheme is based on a cyclic group $G = \langle g \rangle$ – the secret key sk is a random element from $\mathbb{Z}_{|G|}$, whereas the public key is $\mathsf{pk} = g^{\mathsf{sk}}$. Then, encapsulation produces a ciphertext-key pair

$$C \leftarrow g^r \ , \ \ K \leftarrow \mathsf{H}(\mathsf{pk}^r).$$

for $r \leftarrow \mathbb{Z}_{|G|}$ and a hash function $\mathsf{H} : G \rightarrow \{0,1\}^{\ell}$. Decapsulation occurs by computing $K \leftarrow \mathsf{H}(C^{\mathsf{sk}})$.

The CCA-security of Hashed ElGamal in the random-oracle model was proved by Abdalla, Bellare, and Rogaway [1] based on the *Strong Diffie-Hellman* (SDH) assumption (also often called GapDH), and we briefly review the proof.[2] First, recall that in the SDH assumption, the attacker is asked to compute g^{uv} from g^u and g^v, given additionally access to a *decision* oracle $\mathsf{O_v}$ which on input $h, y \in G$, tells us whether $h^v = y$.

The reduction sets the Hashed ElGamal public-key to $\mathsf{pk} = g^v$ (setting implicitly $\mathsf{sk} = v$), the challenge ciphertext to be $C^* = g^u$, and the corresponding key K^* to be a random string. Then, it simulates both the random oracle and the decapsulation oracle to the adversary $\mathcal{A}$ (which is run on inputs pk, C^* and K^*), until a random-oracle query for g^{uv} is made (this can be detected using the $\mathsf{O_v}$ oracle). The challenge is to simulate both oracles consistently: As the reduction cannot compute discrete logarithms, it uses the oracle $\mathsf{O_v}$ to detect whether a random-oracle query X and a decapsulation query C_i satisfy $\mathsf{O_v}(C_i, X) = \mathsf{true}$, and, if this is the case, answers them with the same value.

This reduction requires memory to store all prior decapsulation and random-oracle queries. Unlike other reductions, the problem here is not to store the random-oracle output values (which could be compressed using a PRF), but the actual *inputs* to these queries, which are under adversarial control. This motivates the conjecture that a reduction using little memory does not exist, but the main challenge is of course to prove this is indeed the case.

[2] Abdalla et al. [1] do not phrase their paper in terms of the KEM/DEM paradigm [6,12], which was introduced concurrently – instead, they prove that an intermediate assumption, called Oracle Diffie-Hellman (ODH), follows from SDH in the ROM. However, the ODH assumption is structurally equivalent to the CCA security of Hashed ElGamal KEM for one challenge ciphertext.

Our result, in summary. We provide a *memory* lower bound for reductions that are *generic* with respect to the underlying group G. Specifically, we show the existence of an (inefficient) adversary $\mathcal{A}$ in *the generic group model (GGM)* which breaks the CCA security of Hashed ElGamal via $O(k)$ random oracle/decapsulation queries, but such that no reduction using less than $k \cdot \lambda$ bits of memory can break the SDH assumption *even* with access to $\mathcal{A}$, where λ is the bit-size of the underlying group elements.

Our lower bound is strong in that it shows we do not even have a trade-off between advantage and memory, i.e., if the memory is smaller than $k \cdot \lambda$, then the advantage is very small, as long as the reduction makes a polynomial number of queries to O_v and to the generic group oracle. It is however also important to discuss two limitations of our lower bound. The first one is that the reduction – which receives g, g^v in the SDH game – uses $\mathsf{pk} = g^v$ as the public key to the Hashed ElGamal adversary. The second one is that the reduction is straightline, i.e., it does not perform any rewinding.

We believe that our impossibility result would extend even when the reduction is not straightline. However, allowing for rewinding appears to be out of reach of our techniques. Nonetheless, we *do* conjecture a lower bound on the memory of $\Omega(k \log k)$ bits, and discuss the reasoning behind our conjecture in detail in the full version.

We stress that our result applies to reductions in the GGM, but treats the adversary as a black box. This captures reductions which are black-box in their usage of the group and the adversary. (In particular, the reduction cannot see generic group queries made by the adversary, as in a GGM security proofs.) Looking at the GGM reduces the scope of our result. However, it is uncommon for reductions to depend on the specifics of the group, although our result can be bypassed for specific groups, e.g., if the group has a pairing.

Concurrent related work. Concurrently to our work, Bhattacharyya [4] provides memory-tight reductions of KEM-CCA security for variants of Hashed ElGamal. At first glance, the results seem to contradict ours. However, they are entirely complementary – for example, a first result shows a memory tight reduction for the KEM-CCA security of the "Cramer-Shoup" variant of Hashed ElGamal – this variant differs from the (classical) Hashed ElGamal we consider here and is less efficient. The second result shows a memory-tight reduction for the version considered in this paper, but assumes that the underlying group has a pairing. This is a good example showing our result can be bypassed for specific groups i.e. groups with pairings, but we also note that typical instantiations of the scheme are on elliptic curves for which no pairing exists.

1.1 Our Techniques

We give a high-level overview of our techniques here. We believe some of these to be novel and of broader interest in providing other impossibility results.

The shuffling game. Our adversary against Hashed ElGamal[3] $\mathcal{A}$ first attempts to detect whether the reduction is using a sufficient amount of memory. The adversary $\mathcal{A}$ is given as input the public key g^v, as well as g^u, as well as a string $C^* \in \{0,1\}^\ell$, which is either a real encapsulation or a random string. It first samples k values $i_1, \ldots, i_k$ from $\mathbb{Z}_p$. It then:

(1) Asks for decapsulation queries for $C_j \leftarrow g^{i_j}$, obtaining values K_j, for $j \in [k]$
(2) Picks a random permutation $\pi : [k] \rightarrow [k]$.
(3) Asks for RO queries for $H_j \leftarrow \mathsf{H}(V_j)$ for $j \in [k]$, where $V_j \leftarrow g^{v \cdot i_{\pi(j)}}$.

After this, the adversary checks whether $K_j = H_{\pi(j)}$ for all $j \in [k]$, and if so, it continues its execution, breaking the ODH assumption (inefficiently). If not, it just outputs a random guess.

The intuition here is that no reduction using substantially less than $k \cdot \log p$ bits succeeds in passing the above test – in particular, because the inputs C_j and V_j are (pseudo-)random, and thus incompressible. If the test does not pass, the adversary $\mathcal{A}$ is rendered useless, and thus not helpful to break SDH.

Remark 1. The adversary here is described in a way that requires secret randomness, not known to the reduction, and it is easier to think of $\mathcal{A}$ in this way. We will address in the body how to generically make the adversary deterministic.

Remark 2. We stress that this adversary requires memory – it needs to remember the answers $C_1, \ldots, C_k$. However, recall that we adopt a black-box approach to memory-tightness, where our requirement is that the reduction itself uses little memory, regardless of the memory used by the adversary. We also argue this is somewhat necessary – it is not clear how to design a reduction which adapts its memory usage to the adversary, even if given this information in a non-black-box manner. Also, we conjecture different (and much harder to analyze) memory-less adversaries exist enabling a separation. An example is multi-round variant, where each round omits **(2)**, and **(3)** only asks a single query $\mathsf{H}(V_j)$ for a random $j \leftarrow_{\$} [k]$, and checks consistency. Intuitively, the chance of passing each round is roughly $k \log p/s$, but we do not know how to make this formal.

Introducing the GGM. Our intuition is however *false* for an arbitrary group. For instance, if the discrete logarithm (DL) problem is easy in the group, then the reduction can simply simulate the random oracle via a PRF, as suggested in [2]. Ideally, we could prove that if the DL problem is hard in G, then any PPT reduction given access to $\mathcal{A}$ and with less than $k \cdot \log p$ bits of memory fails to break SDH.[4] Unfortunately, it will be hard to capture a single hardness property of G sufficient for our proof to go through. Instead, we will model the group via the *generic group model* (GGM) [9,11]: We model a group of prime

[3] The paper will in fact use the cleaner formalization of the ODH assumption, so we stick to Hashed ElGamal only in the introduction.

[4] This statement is somewhat confusing, so note that in general, the existence of a reduction is *not* a contradiction with the hardness of DL, as the reduction is meant to break SDH only given access to an adversary breaking the scheme, and this does not imply the ability to break SDH *without* access to the adversary.

order p defined via a random injection $\sigma : \mathbb{Z}_p \rightarrow \mathcal{L}$. An algorithm in the model typically has access to $\sigma(1)$ (in lieu of g) and an evaluation oracle which on input $\mathbf{a}, \mathbf{b} \in \mathcal{L}$ returns $\sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$. (We will keep writing g^i instead of $\sigma(i)$ in the introduction, for better legibility.)

The permutation game. In order to fool $\mathcal{A}$, the reduction can learn information about π via the $\mathsf{O_v}$ oracle. For example, it can try to input $C_j = g^{i_j}$ and $V_{j'} = g^{v i_{\pi(j')}}$ (both obtained from $\mathcal{A}$'s queries), and $\mathsf{O_v}(C_j, V_{j'}) = \mathsf{true}$ if and only if $\pi(j') = j$. More generally, the reduction can compute, for any $\vec{a} = (a_1, \ldots, a_k)$ and $\vec{b} = (b_1, \ldots, b_k)$,

$$C^* = g^{\sum_{j=1}^{k} a_j i_j} = \prod_{j=1}^{k} C_j^{a_j} \ , \ \ V^* = g^{\sum_{j=1}^{k} b_j v \cdot i_{\pi(j)}} = \prod_{j=1}^{k} V_j^{b_j},$$

and the query $\mathsf{O_v}(C^*, V^*)$ returns true iff $b_j = a_{\pi(j)}$ for all $j \in [k]$, which we write as $\vec{b} = \pi(\vec{a})$. We abstract this specific strategy in terms of an information-theoretic game – which we refer to as the *permutation game* – which gives the adversary access to an oracle O which takes as inputs pairs of vectors $(\vec{a}, \vec{b})$ from $\mathbb{Z}_p^k$, and returns true iff $\vec{b} = \pi(\vec{a})$ for a secret permutation π. The goal of the adversary is to recover π.

Clearly, a strategy can win with $O(k^2)$ oracle queries $(\vec{e}_i, \vec{e}_j)$ for all i, j, where $\vec{e}_i \in \mathbb{Z}_p^k$ is the unit vector with a 1 in the i-th coordinate, and 0 elsewhere. This strategy requires in particular querying, in its first component, vectors which have rank k. Our first result will prove that this is necessary – namely, assume that an adversary makes a sequence of q queries $(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_q, \vec{y}_q)$ such that the rank of $\vec{x}_1, \ldots, \vec{x}_p$ is at most ℓ, then the probability to win the permutation game is of the order $O(q^\ell / k!)$. We will prove this via a compression argument.

Note that roughly, this bound tells us that to win with probability ϵ and q queries to the oracle, the attacker needs

$$\ell = \Omega\left(\frac{k \log k - \log(1/\epsilon)}{\log(q)}\right).$$

A reduction to the permutation game. We will think of the execution of the reduction against our adversary as consisting of two stages – we refer to them as $\mathcal{R}_1$ and $\mathcal{R}_2$. The former learns the decapsulation queries $g^{i_1}, \ldots, g^{i_k}$, whereas the latter learns the RO queries $g^{i_{\pi(1)} v}, \ldots, g^{i_{\pi(k)} v}$, and (without loss of generality) attempts to guess the permutation π. We will lower bound the size of the state ϕ that $\mathcal{R}_1$ passes on to $\mathcal{R}_2$. Both stages can issue $\mathsf{O_v}$ and Eval queries.

Note that non-trivial $\mathsf{O_v}$ queries (i.e., those revealing some information about the permutation), are (except with very small probability) issued by $\mathcal{R}_2$, since no information about π is ever revealed to $\mathcal{R}_1$. As one of our two key steps, we will provide a reduction from the execution of $\mathcal{R}_1, \mathcal{R}_2$ against $\mathcal{A}$ in the GGM to the permutation game – i.e., we build an adversary $\mathcal{D}$ for the latter game simulating the interaction between $\mathcal{R}_1, \mathcal{R}_2$ and $\mathcal{A}$, and such that $\mathcal{R}_1, \mathcal{R}_2$ "fooling" $\mathcal{A}$ results in $\mathcal{D}$ guessing the permutation.

Memory vs. rank. The main question, however, is to understand the complexity of $\mathcal{D}$ in the permutation game, and in particular, the *rank* ℓ of the first component of its queries – as we have seen above, this affects its chance of winning the game.

To do this, we will take a slight detour, and specifically consider a set $\mathcal{Z} \subseteq \mathcal{L}$ of labels (i.e., outputs of σ) that the reduction $\mathcal{R}_2$ comes up with (as inputs to either of Eval or $\mathsf{O_v}$) on its own (in the original execution), i.e., no earlier Eval query of $\mathcal{R}_2$ returned them, and that have been previously learnt by $\mathcal{R}_1$ as an output of its Eval queries. (The actual definition of $\mathcal{Z}$ is more subtle, and this is due to the ability of the adversary to come up with labels *without* knowing the corresponding pre-image.)

Then, we will show two statements about $\mathcal{Z}$:

(i) On the one hand, we show that the rank ℓ of the oracle queries of the adversary $\mathcal{D}$ is upper bound by $|\mathcal{Z}|$ in its own simulation of the execution of $\mathcal{R}_1, \mathcal{R}_2$ with $\mathcal{A}$.

(ii) On the other hand, via a compression argument, we prove that the size of $\mathcal{Z}$ is related to the length of ϕ, and this will give us our final upper bound.

This latter statement is by itself not very surprising – one can look at the execution of $\mathcal{R}_2$, and clearly every label in $\mathcal{Z}$ that appears "magically" in the execution must be the result of storing them into the state ϕ. What makes this different from more standard compression arguments is the handling of the generic group model oracle (which admits non-trivial operations). In particular, our compression argument will compress the underlying map σ, and we will need to be able to figure out the pre-images of these labels in $\mathcal{Z}$. We give a very detailed technical overview in the body explaining the main ideas.

Memory-Tight AGM Reduction. The Algebraic Group Model (AGM) was introduced in [8]. AGM reductions make strong extractability assumptions, and the question of their memory-tightness is an interesting one. In the full version we construct a reduction to the discrete logarithm problem that runs an adversary against the KEM-CCA security of Hashed ElGamal in the AGM such that the reduction is memory-tight but not tight with respect to advantage. We note that the model of our reduction is different than a (full-fledged) GGM reduction which is not black-box, in that it can observe the GGM queries made by the adversary. Our result does not imply any impossibility for these. In turn, AGM reductions are weaker, but our results do not imply anything about them, either.

2 Preliminaries

In this section, we review the formal definition of the generic group model. We also state ODH and SDH as introduced in [1] in the generic group model.

Notation. Let $\mathbb{N} = \{0, 1, 2, \cdots\}$ and, for $k \in \mathbb{N}$, let $[k] = \{1, 2, \cdots, k\}$. We denote by $\mathsf{InjFunc}(S_1, S_2)$ the set of all injective function from S_1 to S_2.

We also let $*$ denote a wildcard element. For example $\exists t : (t, *) \in T$ is true if the set T contains an ordered pair whose first element is t (the type of the wildcard element shall be clear from the context). Let $\mathcal{S}_k$ denote the set of all permutations on $[k]$. We use $f : \mathsf{D} \rightarrow \mathsf{R} \cup \{\bot\}$ to denote a partial function, where $f(x) = \bot$ indicates the value of $f(x)$ is undefined. Define in particular $D(f) = \{d \in \mathsf{D} \ : \ f(d) \neq \bot\}$ and $R(f) = \{r \in \mathsf{R} \ : \ \exists d \in \mathsf{D} \ : \ \sigma(d) = r\}$. Moreover, we let $\overline{D(f)} = \mathsf{D} \setminus D(f)$ and $\overline{R(f)} = \mathsf{R} \setminus R(f)$.

We shall use pseudocode descriptions of games inspired by the code-based framework of [3]. The output of a game is denoted using the symbol $\Rightarrow$. In all games we assume the flag bad is set to false initially. In pseudocode, we denote random sampling using $\leftarrow_{\$}$, assignment using $\leftarrow$ and equality check using $=$. In games that output boolean values, we use the term"winning" the game to mean that the output of the game is true.

We also introduce some linear-algebra notation. Let S be a set vectors with equal number of coordinates. We denote the rank and the linear span of the vectors by $\mathsf{rank}(S)$ and $\mathsf{span}(S)$ respectively. Let $\vec{x}, \vec{y}$ be vectors of dimension k. We denote $\vec{z}$ of dimension $2k$ which is the concatenation of $\vec{x}, \vec{y}$ as $\vec{z} = (\vec{x}, \vec{y})$. We denote the element at index i of a vector $\vec{x}$ as $\vec{x}[i]$.

2.1 Generic Group Model

The *generic group model* [11] captures algorithms that do not use any special property of the encoding of the group elements, other than assuming every element of the group has a unique representation, and that the basic group operations are allowed. This model is useful in proving lower bounds for some problems, but we use it here to capture reductions that are not specific to the underlying group.

More formally, let the order of the group be a large prime p. Let $\mathbb{Z}_p = \{0, 1, 2, \cdots, p-1\}$. Let $\mathcal{L} \subset \{0,1\}^*$ be a set of size p, called the set of *labels*. Let σ be a random injective mapping from $\mathbb{Z}_p$ to $\mathcal{L}$. The idea is that now every group element in $\mathbb{Z}_p$ is represented by a label in $\mathcal{L}$. An algorithm in this model takes as input $\sigma(1), \sigma(x_1), \sigma(x_2), \cdots, \sigma(x_n)$ for some $x_1, \cdots, x_n \in \mathbb{Z}_p$ (and possibly other inputs which are not group elements). The algorithm also has access to an oracle named Eval which takes as input two labels $\mathbf{a}, \mathbf{b} \in \mathcal{L}$ and returns $\mathbf{c} = \sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$. Note that for any d, given $\sigma(x_i)$, $\sigma(d \cdot x_i)$ can be computed using $O(\log d)$ queries to Eval. We denote this operation as $\mathsf{Exp}(\sigma(x_i), d)$. We assume that all labels queried by algorithms in the generic group model are valid i.e. all labels queried by algorithms in the generic group model are in $\mathcal{L}$.[5]

Oracle Diffie-Hellman assumption (ODH). We first formalize the Oracle Diffie-Hellman Assumption (ODH) [1], which we are going to use in lieu of the CCA security of Hashed ElGamal. Suppose, a group has generator g and order p. The domain of hash function H is all finite strings and range is $\{0,1\}^{\mathsf{hLen}}$. The assumption roughly states for $u, v \leftarrow_{\$} \mathbb{Z}_p, W \leftarrow_{\$} \{0,1\}^{\mathsf{hLen}}$, the distributions $(g^u, g^v, \mathsf{H}(g^{uv}))$ and (g^u, g^v, W) are indistinguishable to an adversary who has access to the oracle H_{v} where $\mathsf{H}_{\mathsf{v}}(g^x)$ returns $\mathsf{H}(g^{xv})$ with the restriction that it is not queried on g^u.

[5] We stress that we assume a strong version of the model where the adversary *knows* $\mathcal{L}$.

```
Game $\mathbb{G}^{\text{ODH-REAL-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A})$ :
1 : $\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p \to \mathcal{L})$
2 : $u \leftarrow\$ \mathbb{Z}_p; U \leftarrow \sigma(u)$
3 : $v \leftarrow\$ \mathbb{Z}_p; V \leftarrow \sigma(v)$
4 : $\mathsf{H} \leftarrow\$ \Omega_{\mathsf{hLen}}$
5 : $W \leftarrow \mathsf{H}(\sigma(uv))$
6 : $b \leftarrow \mathcal{A}^{\mathsf{H_v}(.),\mathsf{H}(.),\mathsf{Eval}(.,.)}(U, V, W, \sigma(1))$
7 : return $b$
```

```
Game $\mathbb{G}^{\text{ODH-RAND-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A})$ :
1 : $\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p \to \mathcal{L})$
2 : $u \leftarrow\$ \mathbb{Z}_p; U \leftarrow \sigma(u)$
3 : $v \leftarrow\$ \mathbb{Z}_p; V \leftarrow \sigma(v)$
4 : $\mathsf{H} \leftarrow\$ \Omega_{\mathsf{hLen}}$
5 : $W \leftarrow \{0,1\}^{\mathsf{hLen}}$
6 : $b \leftarrow \mathcal{A}^{\mathsf{H_v}(.),\mathsf{H}(.),\mathsf{Eval}(.,.)}(U, V, W, \sigma(1))$
7 : return $b$
```

```
Oracle $\mathsf{Eval}(\mathbf{a}, \mathbf{b})$ :
1 : return $\sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$
```

```
Oracle $\mathsf{H_v}(\mathbf{a})$ :
1 : if $\mathbf{a} = U$ thenreturn $\perp$
2 : else return $\mathsf{H}(\sigma(\sigma^{-1}(\mathbf{a}) \cdot v))$
```

```
Game $\mathbb{G}^{\text{SDH-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A})$ :
1 : $\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p \to \mathcal{L})$
2 : $u \leftarrow\$ \mathbb{Z}_p; U \leftarrow \sigma(u)$
3 : $v \leftarrow\$ \mathbb{Z}_p; V \leftarrow \sigma(v)$
4 : $z \leftarrow \mathcal{A}^{\mathsf{Eval}(.,.),\mathsf{O_v}(.,.)}(U, V, \sigma(1))$
5 : return $(z = \sigma(uv))$
```

```
Oracle $\mathsf{O_v}(\mathbf{a}, \mathbf{b})$ :
1 : return $(\sigma^{-1}(\mathbf{a}) \cdot v = \sigma^{-1}(\mathbf{b}))$
```

Fig. 1. Games for ODH and SDH assumptions

We give a formalization of this assumption in the random-oracle and generic group models. For a fixed $\mathsf{hLen} \in \mathbb{N}$, let Ω_{hLen} be the set of hash functions mapping $\{0,1\}^*$ to $\{0,1\}^{\mathsf{hLen}}$. In Fig. 1, we formally define the Games $\mathbb{G}^{\text{ODH-REAL-GG}}_{\mathcal{L},p,\mathsf{hLen}}$, $\mathbb{G}^{\text{ODH-RAND-GG}}_{\mathcal{L},p,\mathsf{hLen}}$. The advantage of violating ODH is defined as

$$\mathsf{Adv}^{\text{ODH-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A}) = \left|\Pr\left[\mathbb{G}^{\text{ODH-REAL-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbb{G}^{\text{ODH-RAND-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A}) \Rightarrow 1\right]\right|.$$

Strong Diffie-Hellman Assumption (SDH). This is a stronger version of the classical CDH assumption. This assumption roughly states that CDH is hard even in the presence of a DDH-oracle $\mathsf{O_v}$ where $\mathsf{O_v}(g^x, g^y)$ is true if and only if $x \cdot v = y$.

We formally define the game $\mathbb{G}^{\text{SDH-GG}}$ in the generic group model in Fig. 1. The advantage of violating SDH is defined as

$$\mathsf{Adv}^{\text{SDH-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A}) = \left|\Pr\left[\mathbb{G}^{\text{SDH-GG}}_{\mathcal{L},p,\mathsf{hLen}}(\mathcal{A}) \Rightarrow \mathsf{true}\right]\right|.$$

Note in particular that one *can* upper bound this advantage unconditionally. We shall drop the $\mathcal{L}$ from the subscript of advantages and games henceforth since the set of labels $\mathcal{L}$ remains the same throughout our paper.

Black Box reductions in the GGM. We consider black-box reductions in the generic group model. We will limit ourselves to an informal description, but this can easily be formalized within existing formal frameworks for reductions (see e.g. [10]). We let the reduction $\mathcal{R}$ access an adversary $\mathcal{A}$, and denote by $\mathcal{R}^{\mathcal{A}}$ the resulting algorithm – understood here is that $\mathcal{R}$ supplies inputs, answers queries, etc. In addition, we let $\mathcal{R}$ and $\mathcal{A}$ access the Eval oracle available in the GGM. We stress that the GGM oracle is not under the reduction's control here – typically, the reduction itself will break a (hard) problem in the GGM with help of $\mathcal{A}$. We will allow (for simplicity) $\mathcal{A}$ to be run depending on some secret private coins[6] not accessible by $\mathcal{R}$. Reductions can run $\mathcal{A}$ several times (with fresh private coins). We call a reduction *straigthline* if it only runs $\mathcal{A}$ *once.*

In our case, the reduction $\mathcal{R}$ will be playing $\mathbb{G}_{p,\mathsf{hLen}}^{\mathsf{SDH\text{-}GG}}$. It receives as inputs $\sigma(1)$, $U = \sigma(u)$, $V = \sigma(v)$, and has access to the Eval, $\mathsf{O_v}$ oracles, as well as an adversary $\mathcal{A}$ for $\mathbb{G}_{p,\mathsf{hLen}}^{\mathsf{ODH\text{-}REAL\text{-}GG}}$ or $\mathbb{G}_{p,\mathsf{hLen}}^{\mathsf{ODH\text{-}RAND\text{-}GG}}$. The reduction needs therefore to supply inputs $(\sigma(1), U', V', W)$ to $\mathcal{A}$, and to answer its queries to the oracles $\mathsf{H_v}$, as well as queries to H. We will call such a reduction *restricted* if it is straightline *and* $V' = V$.

2.2 Compression Lemma

In our lower bound proof we use the compression lemma that was formalized in [7] which roughly means that it is impossible to compress every element in a set with cardinality c to a string less than $\log c$ bits long, even relative to a random string. We state the compression lemma here as a proposition.

Proposition 1. *Suppose, there is a (not necessarily efficient) procedure* $\mathsf{Encode} : \mathcal{X} \times \mathcal{R} \to \mathcal{Y}$ *and a (not necessarily efficient) decoding procedure* $\mathsf{Decode} : \mathcal{Y} \times \mathcal{R} \to \mathcal{X}$ *such that*

$$\Pr_{x \in \mathcal{X}, r \in \mathcal{R}} [\mathsf{Decode}(\mathsf{Encode}(x, r), r) = x] \geqslant \epsilon ,$$

then $\log |\mathcal{Y}| \geqslant \log |\mathcal{X}| - \log(1/\epsilon)$*.*

2.3 Polynomials

Let $\mathsf{p}(X_1, \cdots, X_n)$ be a n variate polynomial. We denote by $\mathsf{p}(x_1, \cdots, x_n)$ the evaluation of p at the point $(x_1, \cdots, x_n)$ throughout the paper. The polynomial ring in variables $X_1, \cdots, X_n$ over the field $\mathbb{Z}_p$ is denoted by $\mathbb{Z}_p[X_1, \cdots, X_n]$.

2.4 Key Encapsulation Mechanism (KEM)

A key-encapsulation mechanism (KEM) consists of three probabilistic polynomial time (PPT) algorithms $\mathsf{Gen}, \mathsf{Encap}, \mathsf{Decap}$. The key generation algorithm Gen is probabilistic and outputs a key-pair $(\mathsf{pk}, \mathsf{sk})$. The encapsulation algorithm Encap is a probabilistic algorithm that takes pk as input and outputs a ciphertext c and a key K where $K \in \mathcal{K}$ for some non-empty set $\mathcal{K}$. The decapsulation algorithm Decap is a deterministic algorithm that takes as input the secret key sk and a ciphertext c outputs a key $K \in \mathcal{K}$ if (sk, c) is a valid secret key-ciphertext pair and $\perp$ otherwise. For correctness, it is required that for all pairs

[6] If we want to allow the reduction to control random bits, we model them explicitly as an additional input.

$(\mathsf{pk}, \mathsf{sk})$ output by Gen, if (K, c) is output by $\mathsf{Encap}(\mathsf{pk})$ then K is the output of $\mathsf{Decap}(\mathsf{sk}, c)$.

Single challenge KEM-CCA security. The single challenge CCA security of a KEM is defined by a pair of games called $\mathbb{G}^{\mathsf{KEM\text{-}CCA\text{-}REAL}}, \mathbb{G}^{\mathsf{KEM\text{-}CCA\text{-}RAND}}$. In both games a $(\mathsf{pk}, \mathsf{sk})$ pair is generated by Gen, and (c, K) is output by the encapsulation algorithm Encap on input pk. The adversary is provided with (pk, c, K) in $\mathbb{G}^{\mathsf{KEM\text{-}CCA\text{-}REAL}}$ and with (pk, c, K') in $\mathbb{G}^{\mathsf{KEM\text{-}CCA\text{-}RAND}}$ where K' is a randomly sampled element of $\mathcal{K}$. The adversary has access to the decapsulation oracle with sk as the secret key and it can make decapsulation queries on any ciphertext except the ciphertext c and has to output a bit. We define the advantage of violating single challenge KEM-CCA security is defined as the absolute value of the difference of probabilities of the adversary outputting 1 in the two games. A KEM is single challenge CCA-secure if for all non-uniform PPT adversaries the advantage of violating single challenge KEM-CCA security is negligible.

Single challenge KEM-CCA of Hashed ElGamal. We describe the KEM for Hashed ElGamal in a group with order p and generator g and a hash function H. The function Gen samples v at random from $\mathbb{Z}_p$, and returns (g^v, v) as the $(\mathsf{pk}, \mathsf{sk})$ pair. The function Encap on input v, samples u at random from $\mathbb{Z}_p$ and returns g^u as the ciphertext and $\mathsf{H}(g^{uv})$ as the key K. The function Decap on input c, returns $\mathsf{H}(c^v)$. Note that Decap in KEM of Hashed ElGamal is identical to the $\mathsf{H_v}$ function as defined in the ODH assumption. It follows that the single challenge KEM-CCA security of Hashed ElGamal is equivalent to the ODH assumption. In particular, in the generic group model when H is modeled as a random oracle, the single challenge KEM-CCA security of Hashed ElGamal is equivalent to the ODH assumption in the random oracle and generic group model.

3 Memory Lower Bound on the ODH-SDH Reduction

3.1 Result and Proof Outline

In this section, we prove a memory lower bound for restricted black-box reductions from ODH to SDH. We stress that the restricted reduction has access only to the $\mathsf{H}, \mathsf{H_v}$ queries of the adversary. As discussed above, the ODH assumption is equivalent to the single-challenge KEM-CCA security of Hashed ElGamal, this proves a memory lower-bound for (restricted) black-box reductions of single challenge KEM-CCA security of Hashed ElGamal to the SDH assumption.

Theorem 1 (Main Theorem). *In the generic group model, with group order p, there exists an ODH adversary $\mathcal{A}$ that makes k H queries and k $\mathsf{H_v}$ queries (where k is a polynomial in $\log p$), a function $\epsilon_1(p, \mathsf{hLen})$ which is negligible in $\log p, \mathsf{hLen}$, and a function $\epsilon_2(p)$ which is negligible in $\log p$, such that,*

1. $\mathsf{Adv}^{\mathsf{ODH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{A}) = 1 - \epsilon_1(p, \mathsf{hLen})$.

2. *For all restricted black-box reductions $\mathcal{R}$, with s bits of memory and making a total of q (assuming $q \geqslant k$) queries to* $\mathsf{O_v}$, Eval*,*

$$\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}}) \leqslant 2 \cdot 2^{\frac{s}{2}} \left(\frac{48q^3}{p}\right)^{\frac{k}{8c}} \left(1 + \frac{6q}{p}\right)^q + \frac{4q^2 \log p + 13q^2 + 5q}{p} + \epsilon_2(p) \ ,$$

where $c = 4\lceil \frac{\log q}{\log k} \rceil$.

This result implies that if $\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}})$ is non-negligible for a reduction $\mathcal{R}$ making q queries where q is a polynomial in $\log p$, then $s = \Omega(k \log p)$ i.e. the memory required by any restricted black-box reduction grows with the number of queries by $\mathcal{A}$. Hence, there does not exist any efficient restricted black-box reduction from ODH to SDH that is memory-tight.

In the full version, we discuss how rewinding can slightly improve the memory complexity to (roughly) $O(k \log k)$, with heavy computational cost (essentially, one rewinding per oracle query of the adversary). We conjecture this to be optimal, but a proof seems to evade current techniques.

De-randomization. Before we turn to the proof – which also connects several technical lemmas presented across the next sections, let us discuss some aspects of the results. As explained above, our model allows for the adversary $\mathcal{A}$ to be run with randomness unknown to $\mathcal{R}$. This aspect *may* be controversial, but we note that there is a generic way for $\mathcal{A}$ to be made deterministic. Recall that $\mathcal{A}$ must be inefficient for the separation to even hold true. For example, $\mathcal{A}$ can use the injection σ from the generic group model to generate its random coin – say, using $\sigma^{-1}(\mathbf{a}_i)$ as coins a priori fixed labels $\mathbf{a}_1, \mathbf{a}_2, \ldots$. It is a standard – albeit tedious and omitted – argument to show that unless the reduction ends up querying the pre-images (which happens with negligible probability only), the $\sigma^{-1}(\mathbf{a}_i)$'s are good random coins.

Strengthening beyond SDH. We would like to note that our result can be strengthened without much effort to a reduction between ODH and a more general version of SDH. Informally, we can extend our result to every problem which is hard in the generic group model in presence of an $\mathsf{O_v}$ oracle. For example, this could be a problem where given g, g^u, and g^v, the attacker needs to output $g^{f(u,v)}$, where f is (a fixed) two-variate polynomial with degree at least 2. We do not include the proof for the strengthened version for simplicity. However, it appears much harder to extend our result to different types of oracles than $\mathsf{O_v}$, as our proof is tailored at this oracle.

Proof. Here, we give the overall structure, the key lemmas, and how they are combined – quantitatively – to obtain the final result.

First off, Lemma 1 establishes that there exists an adversary $\mathcal{A}$ such that $\mathsf{Adv}^{\mathsf{ODH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{A})$ is close to 1, which we will fix (i.e., when we refer to $\mathcal{A}$, we refer to the one guaranteed to exist by the lemma). The proof of Lemma 1 is in Sect. 4.1 and the proof of Lemma 2 is in Sect. 4.2.

Lemma 1. *There exists an adversary $\mathcal{A}$ and a function $\epsilon_1(p, \mathsf{hLen})$ such that is negligible in $\log p, \mathsf{hLen}$, and*

$$\mathsf{Adv}^{\mathsf{ODH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{A}) = 1 - \epsilon_1(p, \mathsf{hLen}).$$

After that, we introduce a game, called $\mathbb{G}_1$ and described in Fig. 3 in Sect. 4.2. Very informally, this is a game played by a two-stage adversary $\mathcal{R}_1, \mathcal{R}_2$ which can pass a state to each other of size s bits and have access to the $\mathsf{Eval}, \mathsf{O_v}$ oracles. The game captures the essence of the reduction $\mathcal{R}$ the adversary $\mathcal{A}$ of having a sufficient amount of memory. This is made formal in Lemma 2, where we show that the probability of the reduction $\mathcal{R}$ winning the $\mathsf{SDH\text{-}GG}$ game while running $\mathcal{A}$ is bounded by the probability of winning $\mathbb{G}_1$.

Lemma 2. *For every restricted black box reduction $\mathcal{R}$ to $\mathsf{SDH\text{-}GG}$ that runs $\mathcal{A}$, there exist adversaries $\mathcal{R}_1, \mathcal{R}_2$ playing $\mathbb{G}_1$, such that the number of queries made by $\mathcal{R}_1, \mathcal{R}_2$ to $\mathsf{Eval}, \mathsf{O_v}$ is same as the number of queries made by $\mathcal{R}$ to $\mathsf{Eval}, \mathsf{O_v}$, the state passed from $\mathcal{R}_1$ to $\mathcal{R}_2$ is upper bounded by the memory used by $\mathcal{R}$ and,*

$$\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}}) \leqslant \Pr[\mathbb{G}_1 \Rightarrow \mathsf{true}] + \frac{4k^2(\log p)^2}{p} + \frac{4qk\log p + q^2}{p}\ .$$

We introduce Games $\mathbb{G}_2, \mathbb{G}_3$ in Fig. 4 in Sect. 4.2. These games are identical to $\mathbb{G}_1$ except for the condition to output true. The condition to output true in these games are disjoint and the disjunction of the two conditions is equivalent to the condition to output true in $\mathbb{G}_1$. A little more specifically, both games depend on a parameter l, which can be set arbitrarily, and in $\mathbb{G}_3$ and $\mathbb{G}_2$ the winning condition of $\mathbb{G}_1$ is strengthened by additional ensuring that a certain set defined during the execution of the game is smaller or larger than l, respectively. Therefore, tautologically,

$$\Pr[\mathbb{G}_1 \Rightarrow \mathsf{true}] = \Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] + \Pr[\mathbb{G}_3 \Rightarrow \mathsf{true}]\ . \tag{1}$$

We now prove the following two lemmas below, in Sects. 4.3 and 4.4,

Lemma 3. *For the game $\mathbb{G}_2$,*

$$\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] \leqslant \frac{q^l}{k!} + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p} + \frac{k^2+k+2}{p}.$$

Lemma 4. *If the size of the state ϕ output by $\mathcal{R}_1$ is s bits and $(\mathcal{R}_1, \mathcal{R}_2)$ make q queries in total in $\mathbb{G}_3$, then*

$$\Pr[\mathbb{G}_3 \Rightarrow \mathsf{true}] \leqslant 2 \cdot 2^{\frac{s}{2}} \left(\frac{8q^2(2k+2+3q)}{p}\right)^{\frac{l}{2}} \left(1 + \frac{6q}{p}\right)^{\frac{2q-l}{2}} + \frac{k^2+k+2}{p}\ .$$

Combining (1) and the result of Lemmas 3 and 4 we get,

$$\Pr[\mathbb{G}_1 \Rightarrow \mathsf{true}] \leqslant 2 \cdot 2^{\frac{s}{2}} \left(\frac{8q^2(2k+2+3q)}{p}\right)^{\frac{l}{2}} \left(1 + \frac{6q}{p}\right)^{\frac{2q-l}{2}} + \frac{2(k^2+k+2)}{p} + \frac{q^l}{k!} + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p}\ . \tag{2}$$

Since $\left(1+\frac{6q}{p}\right)^{\frac{2q-l}{2}} \leqslant \left(1+\frac{6q}{p}\right)^{q}$, combining Lemma 2, (2) we get,

$$\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}}) \leqslant 2\cdot 2^{\frac{s}{2}}\left(\frac{8q^2(2k+2+3q)}{p}\right)^{\frac{l}{2}}\left(1+\frac{6q}{p}\right)^{q}+\frac{2(k^2+k+2)}{p}+ \\ \frac{2q(2k+3q+2)}{p}+\frac{5q}{p}+\frac{4k^2(\log p)^2}{p}+\frac{4qk\log p+q^2}{p}+\frac{q^l}{k!}.$$

We let,

$$\epsilon_2(p)=\frac{q^l}{k!}+\frac{2(k^2+k+2)}{p}+\frac{4k^2(\log p)^2}{p}.$$

Setting $c=\lceil\frac{\log q}{\log k}\rceil$ and $l=\frac{k}{4c}$, $\frac{q^l}{k!}\leqslant\frac{k^{k/4}}{k!}$. By Sterling's approximation $k!\geqslant k^{k+1/2}e^{-k}$. Therefore,

$$\frac{k^{k/4}}{k!}=\frac{k^{k/4}}{k^{k/4}}\frac{e^k}{k^{k/4}}\frac{1}{k^{k/2+1/2}}.$$

For $k>e^4$ (we can set $k>e^4$), $\frac{q^l}{k!}\leqslant\frac{1}{k^{k/2+1/2}}$ i.e. $\frac{q^l}{k!}$ is negligible in $\log p$ for k polynomial in $\log p$. Also, $\frac{2(k^2+k+2)}{p}+\frac{4k^2(\log p)^2}{p}$ is negligible in $\log p$ (since k is a polynomial in $\log p$). So, $\epsilon_2(p)$ is negligible in $\log p$. We have that,

$$\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}}) \leqslant 2\cdot 2^{\frac{s}{2}}\left(\frac{8q^2(2k+2+3q)}{p}\right)^{\frac{k}{8c}}\left(1+\frac{6q}{p}\right)^{q}+ \\ \frac{2q(2k+3q+2)}{p}+\frac{5q}{p}+\frac{4qk\log p+q^2}{p}+\epsilon_2(p).$$

where $c=4\lceil\frac{\log q}{\log k}\rceil$. Assuming $q\geqslant k$ (and thus $q>e^4>2$), we get,

$$\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}}) \leqslant 2\cdot 2^{\frac{s}{2}}\left(\frac{48q^3}{p}\right)^{\frac{k}{8c}}\left(1+\frac{6q}{p}\right)^{q}+\frac{4q^2\log p+13q^2+5q}{p}+\epsilon_2(p).$$

□

4 Proof of Theorem

4.1 Adversary $\mathcal{A}$ Against ODH

In this section, we construct the ODH adversary $\mathcal{A}$ needed for the proof.

Lemma 1. *There exists an adversary $\mathcal{A}$ and a function $\epsilon_1(p,\mathsf{hLen})$ such that is negligible in $\log p,\mathsf{hLen}$, and*

$$\mathsf{Adv}^{\mathsf{ODH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{A})=1-\epsilon_1(p,\mathsf{hLen}).$$

```
Adversary $\mathcal{A}^{\mathsf{H_v}(.),\mathsf{H}(.),\mathsf{Eval}(.,.)}(U, V, W, \sigma(1))$ :
1 : $i_1, \cdots, i_k \leftarrow\$ \mathbb{Z}_p$
2 : foreach $j \in [k]$ do
3 :     $Q_1[j] \leftarrow \mathsf{Exp}(\sigma(1), i_j); Q_2[j] \leftarrow \mathsf{Exp}(V, i_j)$
4 : $\mathsf{honest} \leftarrow 1$
5 : foreach $j \in [k]$ do
6 :     $\mathsf{ans}_1[j] \leftarrow \mathsf{H_v}(Q_1[j])$
7 : $\pi \leftarrow\$ \mathcal{S}_k$
8 : foreach $j \in [k]$ do
9 :     $\mathsf{ans}_2[\pi(j)] \leftarrow \mathsf{H}(Q_2[\pi(j)])$
10 : if $\exists j, l \in [k], j \neq l : (\mathsf{ans}_1[j] = \mathsf{ans}_1[l] \vee \mathsf{ans}_2[j] = \mathsf{ans}_2[l])$ then $\mathsf{honest} \leftarrow 0$
11 : if $\exists j \in [k] : \mathsf{ans}_1[j] \neq \mathsf{ans}_2[j]$ then $\mathsf{honest} \leftarrow 0$
12 : if $\mathsf{honest} = 1$ then
13 :     $\mathsf{temp} \leftarrow \sigma(1); v \leftarrow 1$
14 :     while $(\mathsf{temp} \neq V)$
15 :         $\mathsf{temp} \leftarrow \mathsf{Eval}(\mathsf{temp}, \sigma(1)); v \leftarrow v + 1$
16 :     $\mathsf{inp} \leftarrow \mathsf{Exp}(U, v); W' \leftarrow \mathsf{H}(\mathsf{inp}); b \leftarrow (W' = W)$
17 : else $b \leftarrow\$ \{0, 1\}$
18 : return $b$
```

Fig. 2. The adversary $\mathcal{A}$

The adversary $\mathcal{A}$ is formally defined in Fig. 2. The proof of Lemma 1 itself is deferred to the full version. Adversary $\mathcal{A}$ samples $i_1, \cdots, i_k$ from $\mathbb{Z}_p$, and computes $\sigma(i_j), \sigma(i_j \cdot v)$ for all j in $[k]$. It then makes $\mathsf{H_v}$ queries on $\sigma(i_j)$'s for all j in $[k]$. Adversary $\mathcal{A}$ then samples a permutation π on $[k] \rightarrow [k]$, and then makes H queries on $\sigma(i_{\pi(j)} \cdot v)$'s for all j in $[k]$. If answers of all the H queries are distinct and the answers of all the $\mathsf{H_v}$ queries are distinct and for all j in $[k]$, $\mathsf{H_v}(\sigma(i_j)) = \mathsf{H}(\sigma(i_j \cdot v))$, $\mathcal{A}$ computes the discrete logarithm of V outputs the correct answer. Otherwise it returns a bit uniformly at random. Note that $\mathcal{A}$ is inefficient, but only if it is satisfied from the responses it gets from the reduction using it.

4.2 The Shuffling Games

The Game $\mathbb{G}_1$. We first introduce the two-stage game $\mathbb{G}_1$ played by a pair of adversaries $\mathcal{R}_1$ and $\mathcal{R}_2$. (With some foresight, these are going to be two stages of the reduction.) It is formally described in Fig. 3. Game $\mathbb{G}_1$ involves sampling $\sigma, i_1, \cdots, i_k, v$ from $\mathbb{Z}_p$, then running $\mathcal{R}_1$, followed by sampling permutation π from $\mathcal{S}_k$ and then running $\mathcal{R}_2$. The first stage $\mathcal{R}_1$ has inputs $\sigma(i_1), \cdots, \sigma(i_k)$ and it outputs a state ϕ of s bits along with k strings in $\{0,1\}^{\mathsf{hLen}}$. The second stage $\mathcal{R}_2$ has inputs $\phi, \sigma(i_{\pi(1)} \cdot v), \cdots, \sigma(i_{\pi(k)} \cdot v)$ and it outputs k strings in $\{0,1\}^{\mathsf{hLen}}$. Both the stages $\mathcal{R}_1, \mathcal{R}_2$ have access to oracles $\mathsf{Eval}, \mathsf{O_v}$. Game $\mathbb{G}_1$ outputs true if all the k strings output by $\mathcal{R}_1$ are distinct, and if all the k strings output by $\mathcal{R}_2$ are distinct, and if for all $j \in [k]$, the jth string output by $\mathcal{R}_2$ is identical to the $\pi(j)$th string output by $\mathcal{R}_1$. Additionally, $\mathbb{G}_1$ involves some bookkeeping. The $\mathsf{Eval}, \mathsf{O_v}$ oracles in $\mathbb{G}_1$ take an extra parameter named from as input which indicates whether the query was from $\mathcal{R}_1$ or $\mathcal{R}_2$.

```
Game $\mathbb{G}_1$ :
1 : $\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p, \mathcal{L}); i_1, \cdots, i_k, v \leftarrow\$ \mathbb{Z}_p$
2 : $\mathcal{X} \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}; \mathcal{Y}_1 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}$
3 : $\phi, s_1, \cdots, s_k \leftarrow \mathcal{R}_1^{\mathsf{Eval}(.,.,1), \mathsf{O_v}(.,.,1)}(\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k))$
4 : $\pi \leftarrow\$ \mathcal{S}_k; \mathcal{Y}_2 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1 \cdot v), \cdots, \sigma(i_k \cdot v)\}; \mathcal{Z} \leftarrow \emptyset$
5 : $s'_1, s'_2, \cdots, s'_k \leftarrow \mathcal{R}_2^{\mathsf{Eval}(.,.,2), \mathsf{O_v}(.,.,2)}(\phi, \sigma(1), \sigma(v), \sigma(i_{\pi(1)} \cdot v), \cdots, \sigma(i_{\pi(k)} \cdot v))$
6 : $\mathsf{win} \leftarrow (\forall j \in [k] : s_{\pi(j)} = s'_j) \wedge (\forall j, l \in [k] : j \neq l \implies s_j \neq s_l \wedge s'_j \neq s'_l)$
7 : return $\mathsf{win}$

Oracle $\mathsf{Eval}(\mathbf{a}, \mathbf{b}, \mathsf{from})$ :
1 : $\mathbf{c} \leftarrow \sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$
2 : if $\mathsf{from} = 1$ then
3 :     if $\mathbf{c} \notin \mathcal{Y}_1$ then $\mathcal{X} \xleftarrow{\cup} \{\mathbf{c}\}$
4 :     $\mathcal{Y}_1 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$
5 : if $\mathsf{from} = 2$ then
6 :     if $\mathbf{a} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{a}\}$
7 :     if $\mathbf{b} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{b}\}$
8 :     $\mathcal{Y}_2 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$
9 : return $\mathbf{c}$

Oracle $\mathsf{O_v}(\mathbf{a}, \mathbf{b}, \mathsf{from})$ :
1 : if $\mathsf{from} = 1$ then $\mathcal{Y}_1 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}\}$
2 : if $\mathsf{from} = 2$ then
3 :     if $\mathbf{a} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{a}\}$
4 :     if $\mathbf{b} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{b}\}$
5 :     $\mathcal{Y}_2 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}\}$
6 : return $(v \cdot \sigma^{-1}(\mathbf{a}) = \sigma^{-1}(\mathbf{b}))$
```

Fig. 3. Game $\mathbb{G}_1$. We use the phrase $\mathcal{R}_1, \mathcal{R}_2$ win $\mathbb{G}_1$ to mean $\mathbb{G}_1 \Rightarrow \mathsf{true}$. We shall use this convention for all games in the paper that output boolean values.

We introduce the phrase "seen by" before describing the bookkeeping. A label has been "seen by" $\mathcal{R}_1$ if it was an input to $\mathcal{R}_1$, queried by $\mathcal{R}_1$ or an answer to a previously made $\mathsf{Eval}(.,.,1)$ query. A label has been "seen by" $\mathcal{R}_2$ if it was an input to $\mathcal{R}_2$, queried by $\mathcal{R}_2$ or an answer to a previously made $\mathsf{Eval}(.,.,2)$ query. We describe the sets $\mathcal{X}, \mathcal{Y}_1, \mathcal{Y}_2, \mathcal{Z}$ which are used for bookkeeping in $\mathbb{G}_1$.

- The labels in $\mathcal{X}$ are answers to $\mathsf{Eval}(.,.,1)$ queries such that it has not yet been "seen by" $\mathcal{R}_1$ before the query.
- $\mathcal{Y}_1$ contains all the labels that are input to $\mathcal{R}_1$, queried by $\mathcal{R}_1$ or answers to $\mathsf{Eval}(.,.,1)$ queries i.e. it is the set of labels "seen by" $\mathcal{R}_1$.
- $\mathcal{Y}_2$ contains all the labels that are input to $\mathcal{R}_2$, queried by $\mathcal{R}_1$ or answers to $\mathsf{Eval}(.,.,2)$ queries i.e. it is the set of labels "seen by" $\mathcal{R}_2$.
- All labels in $\mathcal{Z}$ are queried by $\mathcal{R}_2$ and have not been "seen by" $\mathcal{R}_2$ before the query and are in $\mathcal{X}$

The following lemma tells us that we can (somewhat straightforwardly) take a reduction as in the theorem statement, and transform it into an equivalent pair $\mathcal{R}_1, \mathcal{R}_2$ of adversaries for $\mathbb{G}_1$. The point here is that the reduction is very unlikely to succeed in breaking the SDH assumption without doing an effort equivalent to winning $\mathbb{G}_1$ to get $\mathcal{A}$'s help – otherwise, it is left with breaking SDH directly in the generic group model, which is hard. The proof is deferred to the full version.

```
Game $\boxed{\mathbb{G}_2}$, $\boxed{\mathbb{G}_3}$ :
$\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p, \mathcal{L}); i_1, \cdots, i_k, v \leftarrow\$ \mathbb{Z}_p$
$\mathcal{X} \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}; \mathcal{Y}_1 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}$
$\phi, s_1, \cdots, s_k \leftarrow \mathcal{R}_1^{\mathsf{Eval}(.,.,1), \mathsf{O_v}(.,.,1)}(\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k))$
$\pi \leftarrow\$ \mathcal{S}_k; \mathcal{Y}_2 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1 \cdot v), \cdots, \sigma(i_k \cdot v)\}; \mathcal{Z} \leftarrow \varnothing$
$s'_1, s'_2, \cdots, s'_k \leftarrow \mathcal{R}_2^{\mathsf{Eval}(.,.,2), \mathsf{O_v}(.,.,2)}(\phi, \sigma(1), \sigma(v), \sigma(i_{\pi(1)} \cdot v), \cdots, \sigma(i_{\pi(k)} \cdot v))$
$\mathsf{win} \leftarrow (\forall j \in [k] : s_{\pi(j)} = s'_j) \wedge (\forall j, l \in [k] : j \neq l \implies s_j \neq s_l \wedge s'_j \neq s'_l)$
$\boxed{\textbf{return } (\mathsf{win} \wedge |\mathcal{Z}| < l)}$   $\boxed{\textbf{return } (\mathsf{win} \wedge |\mathcal{Z}| \geqslant l)}$
```

Fig. 4. Games $\mathbb{G}_2, \mathbb{G}_3$. The $\mathsf{Eval}, \mathsf{O_v}$ oracles in $\mathbb{G}_2, \mathbb{G}_3$ are identical to those in $\mathbb{G}_1$ and hence we do not rewrite it here. The newly introduced changes compared to $\mathbb{G}_1$ are highlighted. The statement within the thinner box is present only in $\mathbb{G}_3$ and the statement within the thicker box is present only in $\mathbb{G}_2$.

Lemma 2. *For every restricted black box reduction $\mathcal{R}$ to* SDH-GG *that runs $\mathcal{A}$, there exist adversaries $\mathcal{R}_1, \mathcal{R}_2$ playing $\mathbb{G}_1$, such that the number of queries made by $\mathcal{R}_1, \mathcal{R}_2$ to* $\mathsf{Eval}, \mathsf{O_v}$ *is same as the number of queries made by $\mathcal{R}$ to* $\mathsf{Eval}, \mathsf{O_v}$*, the state passed from $\mathcal{R}_1$ to $\mathcal{R}_2$ is upper bounded by the memory used by $\mathcal{R}$ and,*

$$\mathsf{Adv}^{\mathsf{SDH\text{-}GG}}_{p,\mathsf{hLen}}(\mathcal{R}^{\mathcal{A}}) \leqslant \Pr[\mathbb{G}_1 \Rightarrow \mathsf{true}] + \frac{4k^2(\log p)^2}{p} + \frac{4qk\log p + q^2}{p} .$$

The Games $\mathbb{G}_2$ and $\mathbb{G}_3$. In Fig. 4 we define $\mathbb{G}_2, \mathbb{G}_3$ which have an added check on the cardinality of $\mathcal{Z}$ to output true. Everything else remains unchanged (in particular $\mathsf{Eval}, \mathsf{O_v}$ are unchanged from $\mathbb{G}_1$ and we do not specify them again here). The statement within the thinner box is present only in $\mathbb{G}_3$ and statement within the thicker box is present only in $\mathbb{G}_2$. The changes from $\mathbb{G}_1$ have been highlighted. We shall follow these conventions of using boxes and highlighting throughout the paper.

The games $\mathbb{G}_2, \mathbb{G}_3$ are identical to $\mathbb{G}_1$ except for the condition to output true. Since this disjunction of the conditions to output true in $\mathbb{G}_2, \mathbb{G}_3$ is equivalent to the condition to output true in $\mathbb{G}_1$, and the conditions to output true in $\mathbb{G}_2, \mathbb{G}_3$ are disjoint, we have,

$$\Pr[\mathbb{G}_1 \Rightarrow \mathsf{true}] = \Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] + \Pr[\mathbb{G}_3 \Rightarrow \mathsf{true}] .$$

4.3 Proof of Lemma 3

Recall we are going to prove the following lemma.

Lemma 3. *For the game $\mathbb{G}_2$,*

$$\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] \leqslant \frac{q^l}{k!} + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p} + \frac{k^2+k+2}{p}.$$

```
Game $\mathbb{PG}(\mathcal{A})$ :
1 :  $\pi \leftarrow\!\!\$\ \mathcal{S}_k$
2 :  $\pi' \leftarrow \mathcal{A}^{O(\cdot,\cdot)}$
3 :  return $(\pi = \pi')$

Oracle $O(\vec{x}, \vec{y})$ : // $\vec{x} \in \mathbb{Z}_p^k, \vec{y} \in \mathbb{Z}_p^k$
1 :  return $(\forall i \in [k] : \vec{x}[\pi(i)] = \vec{y}[i])$
```

Fig. 5. The permutation game $\mathbb{PG}$ being played by adversary $\mathcal{A}$ is denoted by $\mathbb{PG}(\mathcal{A})$

We introduce a new game – called the *permutation game* and denoted $\mathbb{PG}$ – in order to upper bound $\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}]$. In the rest of this proof, we are going to first define the game, and upper bound the winning probability of an adversary. Then, we are going to reduce an adversary for $\mathbb{G}_2$ to one for $\mathbb{PG}$.

The Permutation Game. In Game $\mathbb{PG}$, an adversary has to guess a randomly sampled permutation π over $[k]$. The adversary has access to an oracle that takes as input two vectors of length k and returns true if the elements of the first vector, when permuted using π, results in the second vector and false otherwise. Figure 5 formally describes the game $\mathbb{PG}$.

In the following, we say an adversary playing $\mathbb{PG}$ is a (q, l)-query adversary if it makes at most q queries to O, and the rank of the vectors that were the first argument to the O queries returning true is at most l.

The following lemma – which we prove via a compression argument – yields an upper bound on the probability of winning the game for a (q, l)-query adversary.

Lemma 5. *For a (q, l)-query adversary $\mathcal{A}$ playing $\mathbb{PG}$ the following is true.*

$$\Pr[\mathbb{PG}(\mathcal{A}) \Rightarrow \mathsf{true}] \leqslant \frac{q^l}{k!}.$$

Proof. We construct an encoding of π by running adversary $\mathcal{A}$. In order to run $\mathcal{A}$, all the O queries need to be correctly answered. This can be naively done by storing the sequence number of queries whose answers are true. In fact, of all such queries, we need to just store the sequence number of just those whose first argument is not in the linear span of vectors which were the first argument of previous such queries i.e. we store the sequence number of only those O queries returning true whose first argument form a basis of the first argument of all O queries returning true. This approach works because for every vector $\vec{x}$, there is only a unique vector $\vec{y}$ such that $O(\vec{x}, \vec{y}) = 1$. The random tape of the adversary can be derived using the common randomness of Encode, Decode and hence the adversary produces identical queries and output. For simplicity, we do not specify this explicitly in the algorithms and treat $\mathcal{A}$ as deterministic. The formal description of the algorithms Encode, Decode are in Fig. 6.

Observe that S is a basis of vectors $\vec{x}$ such that $O(\vec{x}, \vec{y}) = \mathsf{true}$. Note that for an $O(\vec{x}, \vec{y})$ query returning true, if $\vec{x} \in S$ then the sequence number of the query is stored in enc. Therefore, $(\vec{x}, \vec{y}) \in S'$ in Decode. Again, for an $O(\vec{x}, \vec{y})$ query returning true, if $\vec{x} \notin S$ then the sequence number of the query is not stored in enc and therefore $(\vec{x}, \vec{y}) \notin S'$. So, for an $O(\vec{x}, \vec{y})$ query returning true,

```
Procedure Encode($\pi$) :
  $c \leftarrow 0$
  $S \leftarrow \varnothing$
  enc $\leftarrow \varnothing$
  $\pi' \leftarrow \mathcal{A}^{O(\cdot,\cdot)}$
  return enc

Oracle $O(\vec{x}, \vec{y})$ :
  $c \leftarrow c + 1$
  if $(\exists i \in [k] : \vec{x}[\pi(i)] \neq \vec{y}[i])$ then
    return false
  else
    if $\vec{x} \notin \mathsf{span}(S)$ then
      $S \leftarrow S \cup \{\vec{x}\}$
      enc $\leftarrow$ enc $\cup \{c\}$
    return true

Procedure Decode(enc) :
  $c \leftarrow 0$
  $S' \leftarrow \varnothing$
  $\pi' \leftarrow \mathcal{A}^{O(\cdot,\cdot)}$
  return $\pi'$

Oracle $O(\vec{x}, \vec{y})$ :
  $c \leftarrow c + 1$
  if $c \in$ enc then
    $S' \leftarrow S' \cup \{(\vec{x}, \vec{y})\}$
    return true
  return $((\vec{x}, \vec{y}) \in \mathsf{span}(S'))$
```

Fig. 6. Encoding and decoding π using $\mathcal{A}$

$(\vec{x}, \vec{y}) \in S'$ iff $\vec{x} \in S$. Since, for all $(\vec{x}, \vec{y})$ such that $O(\vec{x}, \vec{y}) = \mathsf{true}$ we have that for all $i \in [k]$, $\vec{y}[i] = \vec{x}[\pi^{-1}(i)]$, it follows that S' forms a basis of vectors $(\vec{x}, \vec{y})$ such that $O(\vec{x}, \vec{y}) = \mathsf{true}$.

In Decode(enc), the simulation of $O(\vec{x}, \vec{y})$ is perfect because

– If c is in enc, then $\vec{x} \in S$ in Encode. From the definition of S in Encode, it follows that $O(\vec{x}, \vec{y})$ should return true.
– Otherwise we check if $(\vec{x}, \vec{y}) \in \mathsf{span}(S')$ and return true if the check succeeds, false otherwise. This is correct since in S' is a basis of vectors $(\vec{x}, \vec{y})$ such that $O(\vec{x}, \vec{y}) = \mathsf{true}$.

The encoding is a set of $|S|$ query sequence numbers. Since there are at most q queries, the encoding space is at most $\binom{q}{|S|}$. Using $\mathcal{X}$ to be the set S_k, $\mathcal{Y}$ to be the set of all possible encodings, $\mathcal{R}$ to be the set of random tapes of $\mathcal{A}$, it follows from Proposition 1 that,

$$\Pr\left[\text{Decoding is sucessful}\right] \leqslant \frac{\binom{q}{|S|}}{k!}.$$

Since the simulation of $O(\vec{x}, \vec{y})$ is perfect in Decode, decoding is successful if $\mathbb{PG}(\mathcal{A}) \Rightarrow \mathsf{true}$. Therefore,

$$\Pr\left[\mathbb{PG}(\mathcal{A}) \Rightarrow \mathsf{true}\right] \leqslant \frac{\binom{q}{|S|}}{k!} \leqslant \frac{q^{|S|}}{k!} .$$

Since $\mathcal{A}$ is a (q, l)-query adversary, $|S| \leqslant l$. Thus, we have,

$$\Pr\left[\mathbb{PG}(\mathcal{A}) \Rightarrow \mathsf{true}\right] \leqslant \frac{q^{l}}{k!} \tag{3}$$

□

```
Procedure PopulateSetsEval(a, b, c, from) :
1 : if from = 1 then
2 :     if $\mathbf{c} \notin \mathcal{Y}_1$ then $\mathcal{X} \xleftarrow{\cup} \{\mathbf{c}\}$
3 :     $\mathcal{Y}_1 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$
4 : if from = 2 then
5 :     if $\mathbf{a} \in \mathcal{X} \setminus \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{a}\}$
6 :     if $\mathbf{b} \in \mathcal{X} \setminus \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{b}\}$
7 :     $\mathcal{Y}_2 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$

Procedure PopulateSetsO_v(a, b, from) :
1 : if from = 1 then $\mathcal{Y}_1 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}\}$
2 : if from = 2 then
3 :     if $\mathbf{a} \in \mathcal{X} \setminus \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{a}\}$
4 :     if $\mathbf{b} \in \mathcal{X} \setminus \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{b}\}$
5 :     $\mathcal{Y}_2 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}\}$
```

Fig. 7. Subroutines PopulateSetsEval, PopulateSetsO$_\mathsf{v}$

Reduction to $\mathbb{PG}$. We next show that the $\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}]$ is upper bounded in terms of the probability of a (q,l)-query adversary winning the game $\mathbb{PG}$.

Lemma 6. *There exists a (q,l)-query adversary $\mathcal{D}$ against the permutation game $\mathbb{PG}$ such that*

$$\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] \leqslant \Pr[\mathbb{PG}(\mathcal{D}) \Rightarrow \mathsf{true}] + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p} + \frac{k^2+k+2}{p}.$$

Proof. We transform $\mathcal{R}_1, \mathcal{R}_2$ playing $\mathbb{G}_2$ to an adversary $\mathcal{D}$ playing the game $\mathbb{PG}$ through a sequence of intermediate games and use the upper bound on the probability of winning the game $\mathbb{PG}$ established previously to prove an upper bound on $\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}]$. In order to make the pseudocode for subsequent games compact we define the two subroutines PopulateSetsEval, PopulateSetsO$_\mathsf{v}$ and invoke them from Eval, O$_\mathsf{v}$. The subroutines PopulateSetsEval, PopulateSetsO$_\mathsf{v}$ are formally described in Fig. 7.

The game $\mathbb{G}_4$. We next describe game $\mathbb{G}_4$ where we introduce some additional bookkeeping. In $\mathbb{G}_4$, every valid label that is an input to $\mathcal{R}_1, \mathcal{R}_2$ or queried by $\mathcal{R}_1, \mathcal{R}_2$ or an answer to a query of $\mathcal{R}_1, \mathcal{R}_2$, is mapped to a polynomial in $\mathbb{Z}_p[I_1, \cdots, I_k, V, T_1, \cdots, T_{2q}]$ where q is the total number of Eval, O$_\mathsf{v}$ queries made by $\mathcal{R}_1, \mathcal{R}_2$. The polynomial associated with label $\mathbf{a}$ is denoted by $\mathsf{p}_\mathbf{a}$. Similarly, we define Λ to be a mapping from polynomials to labels. For all labels $\mathbf{a} \in \mathcal{L}$, $\Lambda(\mathsf{p}_\mathbf{a}) = \mathbf{a}$. The mapping from labels to polynomials is done such that for every label $\mathbf{a}$ mapped to $\mathsf{p}_\mathbf{a}$,

$$\sigma^{-1}(\mathbf{a}) = \mathsf{p}_\mathbf{a}(i_1, \cdots, i_k, v, t_1, \cdots, t_{2q}).$$

For compactness, let us denote $(i_1, \cdots, i_k, v, t_1, \cdots, t_{2q})$ by $\vec{i}$. Before running $\mathcal{R}_1$, $\mathsf{p}_{\sigma(1)}, \mathsf{p}_{\sigma(v)}, \mathsf{p}_{\sigma(i_1)}, \cdots, \mathsf{p}_{\sigma(i_k)}, \mathsf{p}_{\sigma(i_1 \cdot v)}, \cdots, \mathsf{p}_{\sigma(i_k \cdot v)}$ are assigned polynomials $1, V, I_1, \cdots, I_k, I_1V, \cdots, I_kV$ respectively and for all other labels $\mathbf{a} \in \mathcal{L}$, $\mathsf{p}_\mathbf{a} = \perp$. The function Λ is defined accordingly. For labels $\mathbf{a}$ queried by $\mathcal{R}_1, \mathcal{R}_2$ that have not been previously mapped to any polynomial (i.e. $\mathsf{p}_\mathbf{a} = \perp$), $\mathsf{p}_\mathbf{a}$ is assigned T_{new} (new starting from 1 and being incremented for every such label queried), the

Game $\mathbb{G}_4$:

```
1 :  $\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p, \mathcal{L})$; foreach $\mathbf{a} \in \mathcal{L}$ do $\mathsf{p}_{\mathbf{a}} \leftarrow \perp$
2 :  foreach $\mathsf{p}' \in \mathbb{Z}_p[I_1, \cdots, I_k, V, T_1, \cdots, T_{2q}]$ do $\Lambda(\mathsf{p}') \leftarrow \perp$
3 :  $i_1, \cdots, i_k, v \leftarrow\$ \mathbb{Z}_p$; $\mathsf{p}_{\sigma(1)} \leftarrow 1$; $\Lambda(1) \leftarrow \sigma(1)$
4 :  if $\mathsf{p}_{\sigma(v)} = \perp$ then $\mathsf{p}_{\sigma(v)} \leftarrow V$
5 :  $\Lambda(V) \leftarrow \sigma(v)$
6 :  foreach $j \in [k]$ do
7 :     if $\mathsf{p}_{\sigma(i_j)} = \perp$ then $\mathsf{p}_{\sigma(i_j)} \leftarrow I_j$
8 :     $\Lambda(I_j) \leftarrow \sigma(i_j)$
9 :     if $\mathsf{p}_{\sigma(v \cdot i_j)} = \perp$ then $\mathsf{p}_{\sigma(v \cdot i_j)} \leftarrow VI_j$
10 :    $\Lambda(VI_j) \leftarrow \sigma(v \cdot i_j)$
11 : $\mathsf{new} \leftarrow 0$; $\mathcal{X} \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}$; $\mathcal{Y}_1 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}$
12 : $\phi, s_1, \cdots, s_k \leftarrow \mathcal{R}_1^{\mathsf{Eval}(.,.,1),\mathsf{O_v}(.,.,1)}(\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k))$
13 : $\pi \leftarrow\$ \mathcal{S}_k$; $\mathcal{Y}_2 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1 \cdot v), \cdots, \sigma(i_k \cdot v)\}$; $\mathcal{Z} \leftarrow \emptyset$
14 : $s'_1, s'_2, \cdots, s'_k \leftarrow \mathcal{R}_2^{\mathsf{Eval}(.,.,2),\mathsf{O_v}(.,.,2)}(\phi, \sigma(1), \sigma(v), \sigma(i_{\pi(1)} \cdot v), \cdots, \sigma(i_{\pi(k)} \cdot v))$
15 : $\mathsf{win} \leftarrow (\forall j \in [k] : s_{\pi(j)} = s'_j) \wedge (\forall j, l \in [k] : j \neq l \implies s_j \neq s_l \wedge s'_j \neq s'_l)$
16 : return $(\mathsf{win} \wedge |\mathcal{Z}| < l)$
```

Oracle $\mathsf{Eval}(\mathbf{a}, \mathbf{b}, \mathsf{from})$:

```
1 :  if $\mathsf{p}_{\mathbf{a}} = \perp$ then
2 :     $\mathsf{AssignPoly}(\mathbf{a})$
3 :  if $\mathsf{p}_{\mathbf{b}} = \perp$ then
4 :     $\mathsf{AssignPoly}(\mathbf{b})$
5 :  $\mathsf{p}' \leftarrow \mathsf{p}_{\mathbf{a}} + \mathsf{p}_{\mathbf{b}}$
6 :  if $\Lambda(\mathsf{p}') = \perp$ then
7 :     if $\exists \mathbf{c}' \in \mathcal{L} : \mathsf{p}_{\mathbf{c}'}(\vec{i}) = \mathsf{p}'(\vec{i})$ then
8 :        $\Lambda(\mathsf{p}') \leftarrow \mathbf{c}'$
9 :     else
10 :       $\Lambda(\mathsf{p}') \leftarrow \sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$;
11 :       $\mathsf{p}_{\Lambda(\mathsf{p}')} \leftarrow \mathsf{p}'$
12 : $\mathsf{PopulateSetsEval}(\mathbf{a}, \mathbf{b}, \Lambda(\mathsf{p}'), \mathsf{from})$
13 : return $\Lambda(\mathsf{p}')$
```

Oracle $\mathsf{O_v}(\mathbf{a}, \mathbf{b}, \mathsf{from})$:

```
1 :  if $\mathsf{p}_{\mathbf{a}} = \perp$ then
2 :     $\mathsf{AssignPoly}(\mathbf{a})$
3 :  if $\mathsf{p}_{\mathbf{b}} = \perp$ then
4 :     $\mathsf{AssignPoly}(\mathbf{b})$
5 :  $\mathsf{ans} \leftarrow (V\mathsf{p}_{\mathbf{a}} = \mathsf{p}_{\mathbf{b}})$
6 :  if $(v\mathsf{p}_{\mathbf{a}}(\vec{i}) = \mathsf{p}_{\mathbf{b}}(\vec{i})) \neq \mathsf{ans}$ then
7 :     $\mathsf{ans} \leftarrow (v\mathsf{p}_{\mathbf{a}}(\vec{i}) = \mathsf{p}_{\mathbf{b}}(\vec{i}))$
8 :  $\mathsf{PopulateSetsO_v}(\mathbf{a}, \mathbf{b}, \mathsf{from})$
9 :  return $\mathsf{ans}$
```

Procedure $\mathsf{AssignPoly}(\mathbf{l})$:

```
1 :  $\mathsf{new} \leftarrow \mathsf{new} + 1$; $t_{\mathsf{new}} \leftarrow \sigma^{-1}(\mathbf{l})$; $\mathsf{p}_{\mathbf{l}} \leftarrow T_{\mathsf{new}}$; $\Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{l}$
```

Fig. 8. $\mathbb{G}_4$ introduces additional bookkeeping. The newly introduced changes compared to $\mathbb{G}_2$ are highlighted.

variable t_{new} is assigned the pre-image of the label and $\Lambda(T_{\mathsf{new}})$ is assigned $\mathbf{a}$. Since there are q queries (each with two inputs), there can be at most $2q$ labels that had not previously been mapped to any polynomial. Hence, the polynomials have variables $I_1, \cdots, I_k, V, T_1, \cdots, T_{2q}$.

For an $\mathsf{Eval}(\mathbf{a}, \mathbf{b}, .)$ query where $\mathbf{c} = \sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$, let $\mathsf{p}' = \mathsf{p}_{\mathbf{a}} + \mathsf{p}_{\mathbf{b}}$. From the definition of p, we have that $\mathsf{p}'(\vec{i}) = \sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b})$. If $\Lambda(\mathsf{p}') \neq \perp$, then by definition of Λ, we have $\Lambda(\mathsf{p}') = \mathbf{c}$. If $\Lambda(\mathsf{p}') = \perp$, then exactly one of the following two must be true.

1. The label $\mathbf{c}$ has been mapped to a polynomial which is different from p'. In this case $\mathsf{p}_\mathbf{c}(\vec{i}) = \mathsf{p}'(\vec{i})$ and $\Lambda(\mathsf{p}')$ is assigned $\mathbf{c}$.
2. The label $\mathbf{c}$ has not been mapped to any polynomial. In this case, $\mathsf{p}_\mathbf{c}$ is assigned p' and $\Lambda(\mathsf{p}')$ is assigned $\mathbf{c}$.

The label $\Lambda(\mathsf{p}')$ is returned as the answer of the Eval query. Note that the output of Eval is $\mathbf{c} = \sigma(\sigma^{-1}(\mathbf{a}) + \sigma^{-1}(\mathbf{b}))$ in all cases, i.e. it is the same as the output of Eval in $\mathbb{G}_2$.

For an $\mathsf{O}_\mathsf{v}(\mathbf{a}, \mathbf{b}, .)$ query, we first assign the boolean value $V\mathsf{p}_\mathbf{a} = \mathsf{p}_\mathbf{b}$ to ans. Note that if ans is true, then $v \cdot \sigma^{-1}(\mathbf{a}) = \sigma^{-1}(\mathbf{b})$. However, we might have that $v \cdot \sigma^{-1}(\mathbf{a}) = \sigma^{-1}(\mathbf{b})$ and $V\mathsf{p}_\mathbf{a} \neq \mathsf{p}_\mathbf{b}$. When this happens, the boolean value $v(\mathsf{p}_\mathbf{a}(\vec{i}) = \mathsf{p}_\mathbf{b}(\vec{i}))$ is assigned to ans. Oracle O_v returns ans. From the definition of p, it follows that the value returned by O_v in $\mathbb{G}_4$ is $(v \cdot \sigma^{-1}(\mathbf{a}) = \sigma^{-1}(\mathbf{b}))$ i.e. it is the same as the output of O_v in $\mathbb{G}_2$.

Figure 8 formally describes $\mathbb{G}_4$. The changes in $\mathbb{G}_4$ compared to $\mathbb{G}_2$ have been highlighted. We have already pointed out that the outputs of $\mathsf{O}_\mathsf{v}, \mathsf{Eval}$ in $\mathbb{G}_4$ are identical to those in $\mathbb{G}_2$. Since the other changes involve only additional bookkeeping, the outputs of $\mathbb{G}_2, \mathbb{G}_4$ are identical. Therefore

$$\Pr[\mathbb{G}_4 \Rightarrow \mathsf{true}] = \Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}]. \tag{4}$$

The game $\mathbb{G}_{11}$. We introduce a new game named $\mathbb{G}_{11}$ in Fig. 9. Initially, for all polynomials p, $\Lambda(\mathsf{p}) = \perp$. In this game $\Lambda(1), \Lambda(V)$, $\Lambda(I_j)$'s, and $\Lambda(VI_j)$'s are assigned distinct labels sampled from $\mathcal{L}$. Adversary $\mathcal{R}_1$ is run with input labels $\Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k)$ and $\mathcal{R}_2$ has input labels $\Lambda(1), \Lambda(V), \Lambda(I_{\pi(1)} \cdot V), \cdots, \Lambda(I_{\pi(k)} \cdot V)$. The bookkeeping is identical to that in $\mathbb{G}_4$. Observe from the pseudocode that the mapping Λ is injective in this game and hence Λ^{-1} is well defined.

For every Eval or O_v query, if for the input label $\mathbf{l}$, $\Lambda^{-1}(\mathbf{l})$ is $\perp$, then $\mathbf{l}$ is assigned to $\Lambda(T_\mathsf{new})$. For every such input label, new is incremented. For an $\mathsf{Eval}(\mathbf{a}, \mathbf{b}, .)$ query, if $\Lambda(\Lambda^{-1}(\mathbf{a}) + \Lambda^{-1}(\mathbf{b}))$ is not defined, then it is assigned a random label in $\overline{R(\Lambda)}$. The label $\Lambda(\Lambda^{-1}(\mathbf{a}) + \Lambda^{-1}(\mathbf{b}))$ is returned as answer. For $\mathsf{O}_\mathsf{v}(\mathbf{a}, \mathbf{b}, .)$, query true is returned iff $V\Lambda^{-1}(\mathbf{a})$ and $\Lambda^{-1}(\mathbf{b})$ are the same polynomials.

We next upper bound $\Pr[\mathbb{G}_4 \Rightarrow \mathsf{true}]$ in terms of $\Pr[\mathbb{G}_{11} \Rightarrow \mathsf{true}]$ in Lemma 7.

Lemma 7. *For the games $\mathbb{G}_4, \mathbb{G}_{11}$, we have,*

$$\Pr[\mathbb{G}_4 \Rightarrow \mathsf{true}] \leqslant \Pr[\mathbb{G}_{11} \Rightarrow \mathsf{true}] + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p} + \frac{k^2+k+2}{p}.$$

The proof of Lemma 7 has been deferred to the full version.

The Adversary $\mathcal{D}$. Next, we construct the adversary $\mathcal{D}$ that plays $\mathbb{PG}$ by simulating $\mathbb{G}_{11}$ to $\mathcal{R}_1, \mathcal{R}_2$, where the permutation π is the secret permutation

```
Game $\mathbb{G}_{11}$ :
1 : foreach $\mathsf{p} \in \mathbb{Z}_p[I_1, \cdots, I_k, V, T_1, \cdots, T_{2q}]$ do $\Lambda(\mathsf{p}) \leftarrow \perp; \Lambda(1) \leftarrow\$ \mathcal{L}$
2 : $\Lambda(V) \leftarrow\$ \overline{R(\Lambda)}$
3 : foreach $j \in [k]$ do
4 :   $\Lambda(I_j) \leftarrow\$ \overline{R(\Lambda)}; \Lambda(VI_j) \leftarrow\$ \overline{R(\Lambda)}$
5 : $\mathsf{new} \leftarrow 0; \mathcal{X} \leftarrow \Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k)\}; \mathcal{Y}_1 \leftarrow \{\Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k)\}$
6 : $\phi, s_1, \cdots, s_k \leftarrow \mathcal{R}_1^{\mathsf{Eval}(.,.,1),\mathsf{O_v}(.,.,1)}(\Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k))$
7 : $\pi \leftarrow\$ \mathcal{S}_k; \mathcal{Y}_2 \leftarrow \{\Lambda(1), \Lambda(V), \Lambda(VI_1), \cdots, \Lambda(VI_k)\}; \mathcal{Z} \leftarrow \varnothing$
8 : $s'_1, s'_2, \cdots, s'_k \leftarrow \mathcal{R}_2^{\mathsf{Eval}(.,.,2),\mathsf{O_v}(.,.,2)}(\phi, \Lambda(1), \Lambda(V), \Lambda(I_{\pi(1)} \cdot V), \cdots, \Lambda(I_{\pi(k)} \cdot V))$
9 : $\mathsf{win} \leftarrow (\forall j \in [k] : s_{\pi(j)} = s'_j) \wedge (\forall j, l \in [k] : j \neq l \implies s_j \neq s_l \wedge s'_j \neq s'_l)$
10 : return $(\mathsf{win} \wedge |\mathcal{Z}| < l)$
```

```
Oracle Eval(a, b, from) :
1 : if $\Lambda^{-1}(\mathbf{a}) = \perp$ then
2 :   $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{a}$
3 : if $\Lambda^{-1}(\mathbf{b}) = \perp$ then
4 :   $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{b}$
5 : $\mathsf{p} \leftarrow \Lambda^{-1}(\mathbf{a}) + \Lambda^{-1}(\mathbf{b})$
6 : if $\Lambda(\mathsf{p}) = \perp$ then
7 :   $\Lambda(\mathsf{p}) \leftarrow\$ \overline{R(\Lambda)}$
8 : PopulateSetsEval($\mathbf{a}, \mathbf{b}, \Lambda(\mathsf{p})$, from)
9 : return $\Lambda(\mathsf{p})$
```

```
Oracle O_v(a, b, from) :
1 : if $\Lambda^{-1}(\mathbf{a}) = \perp$ then
2 :   $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{a}$
3 : if $\Lambda^{-1}(\mathbf{b}) = \perp$ then
4 :   $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{b}$
5 : PopulateSetsO_v($\mathbf{a}, \mathbf{b}$, from)
6 : return $(V\Lambda^{-1}(\mathbf{a}) = \Lambda^{-1}(\mathbf{b}))$
```

Fig. 9. Game $\mathbb{G}_{11}$

```
Procedure PolyMultCheck($\mathsf{p_a}, \mathsf{p_b}$) :
1 : if $\exists j$ : (coefficient($\mathsf{p_a}, T_j$) $\neq 0 \vee$ coefficient($\mathsf{p_a}, VI_j$) $\neq 0$) thenreturn false
2 : if $\exists j$ : (coefficient($\mathsf{p_b}, T_j$) $\neq 0 \vee$ coefficient($\mathsf{p_b}, I_j$) $\neq 0$) thenreturn false
3 : if coefficient($\mathsf{p_b}, V$) $\neq$ coefficient($\mathsf{p_a}, 1$) thenreturn false
4 : foreach $j \in [k]$ do $\vec{x}[j] \leftarrow$ coefficient($\mathsf{p_a}, I_j$); $\vec{y}[j] \leftarrow$ coefficient($\mathsf{p_b}, VI_j$)
5 : if $O(\vec{x}, \vec{y})$ = true then
6 :   if $\vec{x} \notin$ span($S$) then $S \overset{\cup}{\leftarrow} \{\vec{x}\}; \mathcal{Z}' \overset{\cup}{\leftarrow} \{\mathbf{a}\}$
7 :   if $|S| = l$ then ABORT
8 :   return true
9 : else return false
```

Fig. 10. Subroutine PolyMultCheck **for simulating** $\mathsf{O_v}$**.** In particular, coefficient(p, M) returns the coefficient of the monomial M in the polynomial p. The sets S and $\mathcal{Z}'$ have no effect on the behavior, and are only used in the analysis of $\mathcal{D}$. The symbol ABORT indicates that $\mathcal{D}$ aborts and outputs $\perp$.

from $\mathbb{PG}$. As we will discuss below, the core of the adversary $\mathcal{D}$ will boil down to properly simulating the $\mathsf{O_v}$ oracle using the O oracle from $\mathbb{PG}$ *and* simulating the labels $\sigma(i_{\pi(j)})$ (and the associated polynomials) correctly without knowing π. After a correct simulation, $\mathcal{D}$ will simply extract the permutation π.

```
Adversary $\mathcal{D}$ :
1 : foreach $\mathsf{p} \in \mathbb{Z}_p[I_1, \cdots, I_k, V, K_1, \cdots, K_k, T_1, \cdots, T_{2q}]$ do $\Lambda(\mathsf{p}) \leftarrow \perp$
2 : $\Lambda(1) \leftarrow\$ \mathcal{L}; \Lambda(V) \leftarrow\$ \overline{R(\Lambda)}$
3 : foreach $j \in [k]$ do
4 :     $\Lambda(I_j) \leftarrow\$ \overline{R(\Lambda)}; \Lambda(K_j) \leftarrow\$ \overline{R(\Lambda)}$
5 : $\mathsf{new} \leftarrow 0; \mathcal{X} \leftarrow \{\Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k)\}; \mathcal{Y}_1 \leftarrow \{\Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k)\}$
6 : $\phi, s_1, \cdots, s_k \leftarrow \mathcal{R}_1^{\mathsf{Eval}(.,.,1),\mathsf{O_v}(.,.,1)}(\Lambda(1), \Lambda(V), \Lambda(I_1), \cdots, \Lambda(I_k))$
7 : $\mathcal{Y}_2 \leftarrow \{\Lambda(1), \Lambda(V), \Lambda(K_1), \cdots, \Lambda(K_k)\}; \mathcal{Z} \leftarrow \varnothing; \mathcal{Z}' \leftarrow \varnothing; S \leftarrow \varnothing$
8 : $s'_1, s'_2, \cdots, s'_k \leftarrow \mathcal{R}_2^{\mathsf{Eval}(.,.,2),\mathsf{O_v}(.,.,2)}(\phi, \Lambda(1), \Lambda(V), \Lambda(K_1), \cdots, \Lambda(K_k))$
9 : $\mathsf{win}' \leftarrow (\{s_1, \cdots, s_l\} = \{s'_1, \cdots, s'_l\}) \wedge (\forall j, l \in [k] : j \neq l \implies s_j \neq s_l \wedge s'_j \neq s'_l)$
10 : if $\mathsf{win}' = \mathsf{true}$ then
11 :     foreach $i, j \in [k]$ do if $s_i = s'_j$ then $\pi(i) = j$
12 :     return $\pi$
13 : else return $\perp$
```

```
Oracle Eval(a, b, from) :
1 : if $\Lambda^{-1}(\mathbf{a}) = \perp$ then
2 :     $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{a}$
3 : if $\Lambda^{-1}(\mathbf{b}) = \perp$ then
4 :     $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{b}$
5 : $\mathsf{p} \leftarrow \mathsf{p_a} + \mathsf{p_b}$
6 : if $\Lambda(\mathsf{p}) = \perp$ then $\Lambda(\mathsf{p}) \leftarrow\$ \overline{R(\Lambda)}$
7 : PopulateSetsEval$(\mathbf{a}, \mathbf{b}, \Lambda(\mathsf{p}), \mathsf{from})$
8 : return $\Lambda(\mathsf{p})$
```

```
Oracle O_v(a, b, from) :
1 : if $\Lambda^{-1}(\mathbf{a}) = \perp$ then
2 :     $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{a}$
3 : if $\Lambda^{-1}(\mathbf{b}) = \perp$ then
4 :     $\mathsf{new} \leftarrow \mathsf{new} + 1; \Lambda(T_{\mathsf{new}}) \leftarrow \mathbf{b}$
5 : PopulateSetsO$_\mathsf{v}(\mathbf{a}, \mathbf{b}, \mathsf{from})$
6 : PolyMultCheck$(\mathsf{p_a}, \mathsf{p_b})$
```

```
Procedure PopulateSetsEval(a, b, c, from) :
1 : if $\mathsf{from} = 1$ then
2 :     if $\mathbf{c} \notin \mathcal{Y}_1$ then $\mathcal{X} \xleftarrow{\cup} \{\mathbf{c}\}$
3 :     $\mathcal{Y}_1 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$
4 : if $\mathsf{from} = 2$ then
5 :     if $\mathbf{a} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{a}\}$
6 :         if $\mathbf{b} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{b}\}$
7 :         $\mathcal{Y}_2 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$
```

```
Procedure PopulateSetsO_v(a, b, from) :
1 : if $\mathsf{from} = 1$ then $\mathcal{Y}_1 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}\}$
2 : if $\mathsf{from} = 2$ then
3 :     if $\mathbf{a} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{a}\}$
4 :     if $\mathbf{b} \in \mathcal{X} \backslash \mathcal{Y}_2$ then $\mathcal{Z} \xleftarrow{\cup} \{\mathbf{b}\}$
5 :     $\mathcal{Y}_2 \xleftarrow{\cup} \{\mathbf{a}, \mathbf{b}\}$
```

Fig. 11. Adversary $\mathcal{D}$ which plays the permutation game $\mathbb{PG}$. The changes in $\mathcal{D}$ compared to $\mathbb{G}_{11}$ have been highlighted.

To see how this can be done, let us first have a closer look at $\mathbb{G}_{11}$. Let us introduce the shorthand $K_j = VI_{\pi(j)}$ for $j \in [k]$. With this notation, every polynomial input to or output from Eval is a linear combination of the monomials $1, I_1, \ldots, I_k, V, K_1, \ldots, K_k, T_1, T_2, \ldots$. Now, it is convenient to slightly rethink the check of whether $V\mathsf{p_a} = \mathsf{p_b}$ within $\mathsf{O_v}$ with this notation. First off, we observe that if either of the polynomial contains a monomial of the form T_i, the check fails. In fact, it is immediately clear that the check can only possibly succeed is if $\mathsf{p}_{\mathbf{a}}$ is a linear combination of 1 and the I_j's and $\mathsf{p}_{\mathbf{b}}$ is a linear combination of V and the K_j's. Now, assume that

$$\mathsf{p}_{\mathbf{a}}(I_1, \ldots, I_k) = a_0 + \sum_{j=1}^{k} \vec{x}[j] \cdot I_j,$$

$$\mathsf{p}_{\mathbf{b}}(V, K_1, \ldots K_k) = b_0 \cdot V + \sum_{j=1}^{k} \vec{y}[j] \cdot K_j.$$

Then, $V \cdot \mathsf{p}_a = \mathsf{p}_b$ if and only if $a_0 = b_0$ *and* $\vec{y}[j] = \vec{x}[\pi(j)]$ for all $j \in [k]$. If we are now in Game $\mathbb{PG}$, and π is the chosen permutation, then this is equivalent to $O(\vec{x}, \vec{y}) = \mathsf{true}$ and $a_0 = b_0$.

This leads naturally to the adversary $\mathcal{D}$, which we formally describe in Fig. 11. The adversary will simply sample labels $\mathbf{f}_1, \ldots, \mathbf{f}_k$ for $\sigma(v \cdot i_{\pi(1)}), \ldots, \sigma(v \cdot i_{\pi(k)})$, and associate with them polynomials in the variables $K_1, \ldots, K_j$. Other than that, it simulates the game $\mathbb{G}_{11}$, with the exception that the check $V \cdot \mathsf{p}_{\mathbf{a}} = \mathsf{p}_{\mathbf{b}}$ is not implemented using the above approach – summarized in Fig. 10. Note that $\mathcal{D}$ aborts when $|S| = l$ and makes at most q queries to O. Thus $\mathcal{D}$ is a-query adversary against $\mathbb{PG}$. If $\mathcal{D}$ does not abort, then its simulation of $\mathbb{G}_{11}$ is perfect. If $\mathbb{G}_{11} \Rightarrow \mathsf{true}$ and $\mathcal{D}$ does not abort, then win' shall be true and $\mathcal{D}$ will output the correct π.

The rest of the proof will now require proving that whenever $\mathbb{G}_{11}$ outputs true our adversary $\mathcal{D}$ will never abort due to the check $|S| = l$. Since $\mathbb{G}_{11} \Rightarrow \mathsf{true}$ only if $|\mathcal{Z}| < l$, the following lemma implies that $\mathcal{D}$ does not abort if $\mathbb{G}_{11} \Rightarrow \mathsf{true}$.

Lemma 8. *Let $(\vec{x}_1, \vec{y}_1), \cdots, (\vec{x}_u, \vec{y}_u)$ be the queries made by $\mathcal{D}$ to O which return* true. *Then,*

$$\mathsf{rank}(\vec{x}_1, \cdots, \vec{x}_u) \leqslant |\mathcal{Z}|.$$

The proof of Lemma 8 has been deferred to the full version.

We have established that if $\mathbb{G}_{11}$ outputs true, then $\mathcal{D}$ will not abort and hence $\mathcal{D}$ simulates $\mathbb{G}_{11}$ to $\mathcal{R}_1, \mathcal{R}_2$ perfectly. If $\mathsf{win} = \mathsf{true}$ in $\mathbb{G}_{11}$, the checks by $\mathcal{D}$ succeed and $\mathcal{D}$ outputs the correct permutation and wins $\mathbb{PG}$. Therefore, $\mathcal{D}$ is a (q, l)-query adversary such that $\mathbb{PG}(\mathcal{D}) \Rightarrow \mathsf{true}$ if $\mathbb{G}_{11} \Rightarrow \mathsf{true}$. Hence,

$$\Pr[\mathbb{G}_{11} \Rightarrow \mathsf{true}] \leqslant \Pr[\mathbb{PG}(\mathcal{D}) \Rightarrow \mathsf{true}]. \tag{5}$$

Combining Lemma 7 and (4), (5) we get,

$$\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] \leqslant \Pr[\mathbb{PG}(\mathcal{D}) \Rightarrow \mathsf{true}] + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p} + \frac{k^2+k+2}{p}. \tag{6}$$

□

Combining (3) and (6), we get,

$$\Pr[\mathbb{G}_2 \Rightarrow \mathsf{true}] \leqslant \frac{q^l}{k!} + \frac{2q(2k+3q+2)}{p} + \frac{5q}{p} + \frac{k^2+k+2}{p}.$$

```
Game $\boxed{\mathbb{G}_{12}}$, $\boxed{\mathbb{G}_{13}}$ :
1 : $\sigma \leftarrow\$ \mathsf{InjFunc}(\mathbb{Z}_p, \mathcal{L}); i_1, \cdots, i_k, v \leftarrow \mathsf{RestrictedSample}()$
2 : $\mathcal{X} \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}; \mathcal{Y}_1 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k)\}$
3 : $\phi, s_1, \cdots, s_k \leftarrow \mathcal{R}_1^{\mathsf{Eval}(.,.,1), \mathsf{O_v}(.,.,1)}(\sigma(1), \sigma(v), \sigma(i_1), \cdots, \sigma(i_k))$
4 : $\pi \leftarrow\$ \mathcal{S}_k; \mathcal{Y}_2 \leftarrow \{\sigma(1), \sigma(v), \sigma(i_1 \cdot v), \cdots, \sigma(i_k \cdot v)\}; \mathcal{Z} \leftarrow \varnothing$
5 : $s'_1, s'_2, \cdots, s'_k \leftarrow \mathcal{R}_2^{\mathsf{Eval}(.,.,2), \mathsf{O_v}(.,.,2)}(\phi, \sigma(1), \sigma(v), \sigma(i_{\pi(1)} \cdot v), \cdots, \sigma(i_{\pi(k)} \cdot v))$
6 : $\mathsf{win} \leftarrow (\forall j \in [k] : s_{\pi(j)} = s'_j) \wedge (\forall j, l \in [k] : j \neq l \implies s_j \neq s_l \wedge s'_j \neq s'_l)$
7 : return $(\boxed{\mathsf{win} \wedge} |\mathcal{Z}| \geqslant l)$

    Procedure RestrictedSample() :
    1 : $v \leftarrow\$ \mathbb{Z}_p$; if $v \in \{0, 1\}$ then $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{v \leftarrow\$ \mathbb{Z}_p \backslash \{0, 1\}}$
    2 : $\mathcal{S} \leftarrow \{1\}; \mathcal{S}' \leftarrow \{v^{-1}\}$
    3 : foreach $j \in [k]$ do
    4 :     $i_j \leftarrow\$ \mathbb{Z}_p$; if $i_j \in \mathcal{S} \cup \mathcal{S}'$ then $\mathsf{bad} \leftarrow \mathsf{true}$; $\boxed{i_j \leftarrow\$ \mathbb{Z}_p \backslash (\mathcal{S} \cup \mathcal{S}')}$
    5 :     $\mathcal{S} \xleftarrow{\cup} \{i_j\}; \mathcal{S}' \xleftarrow{\cup} \{v^{-1} \cdot i_j\}$
    6 : return $i_1, \cdots, i_k, v$
```

Fig. 12. Games $\mathbb{G}_{12}, \mathbb{G}_{13}$. The $\mathsf{Eval}, \mathsf{O_v}$ oracles in $\mathbb{G}_{12}, \mathbb{G}_{13}$ are identical to those in $\mathbb{G}_3$ and hence we do not rewrite it here. The statement within the thinner box is present only in $\mathbb{G}_{12}$ and the statement within the thicker box is present only in $\mathbb{G}_{13}$. The newly introduced changes compared to $\mathbb{G}_3$ are highlighted.

4.4 Memory Lower Bound When $|\mathcal{Z}| \geqslant l$ (Proof of Lemma 4)

Recall that we need to prove the following lemma, which we do by using a compression argument.

Lemma 4. *If the size of the state ϕ output by $\mathcal{R}_1$ is s bits and $(\mathcal{R}_1, \mathcal{R}_2)$ make q queries in total in $\mathbb{G}_3$, then*

$$\Pr[\mathbb{G}_3 \Rightarrow \mathsf{true}] \leqslant 2 \cdot 2^{\frac{s}{2}} \left(\frac{8q^2(2k+2+3q)}{p}\right)^{\frac{l}{2}} \left(1 + \frac{6q}{p}\right)^{\frac{2q-l}{2}} + \frac{k^2+k+2}{p}.$$

Proof. Our proof does initial game hopping, with easy transitions. It first introduces a new game, $\mathbb{G}_{12}$ (Fig. 12) whose minor difference from game $\mathbb{G}_3$ is that it samples $i_1, \cdots, i_k, v$ using $\mathsf{RestrictedSample}$ which was previously used in game $\mathbb{G}_{11}$. It adds a bad flag while sampling $i_1, \cdots, i_k, v$ which is set to true if v is in $\{0, 1\}$ or if $|1, v, i_1, \cdots, i_k, i_1 \cdot v, \cdots, i_k \cdot v| < 2k + 2$. The bad event does not affect the output of $\mathbb{G}_{12}$ in any way. Observe that even though the sampling of $i_1, \cdots, i_k, v$ is written in a different manner in $\mathbb{G}_{12}$, it is identical to that in $\mathbb{G}_3$. In all other respects these two games are identical.

$$\Pr[\mathbb{G}_3 \Rightarrow \mathsf{true}] = \Pr[\mathbb{G}_{12} \Rightarrow \mathsf{true}]. \tag{7}$$

Games $\mathbb{G}_{12}, \mathbb{G}_{13}$ differ in the procedure $\mathsf{RestrictedSample}$ and the condition to return true. Note that the conditions of bad being set to true is identical in

$\mathbb{G}_{12}, \mathbb{G}_{13}$ and given that bad is not set to true, $\mathbb{G}_{13}$ returns true whenever $\mathbb{G}_{12}$ returns true. Therefore,

$$\Pr[\mathbb{G}_{12} \Rightarrow \mathsf{true}] \leqslant \Pr[\mathbb{G}_{13} \Rightarrow \mathsf{true}] + \Pr[\mathsf{bad} = \mathsf{true} \text{ in } \mathbb{G}_{13}] \ .$$

It is not hard to show (details in the full version) that the probability of bad being set to true in $\mathsf{RestrictedSample}$ is at most $\frac{k^2+k+2}{p}$. Since in $\mathbb{G}_{13}$ bad is set only in $\mathsf{RestrictedSample}$, the probability of bad being set to true is the same. Hence, we get,

$$\Pr[\mathbb{G}_{12} \Rightarrow \mathsf{true}] \leqslant \Pr[\mathbb{G}_{13} \Rightarrow \mathsf{true}] + \frac{k^2+k+2}{p} \ . \tag{8}$$

The compression argument. We assume $\Pr[\mathbb{G}_{13} \Rightarrow \mathsf{true}] = 2\epsilon$. We say a σ is "good" in $\mathbb{G}_{13}$ if

$$\Pr\left[\mathbb{G}_{13} \Rightarrow \mathsf{true} \,\middle|\, \sigma \text{ was sampled in } \mathbb{G}_{13}\right] \geqslant \epsilon.$$

It follows from Markov's inequality that at least ϵ fraction of σ's are "good".

The following lemma captures the essence of our compression argument.

Lemma 9. *If the state output by $\mathcal{R}_1$ has size s bits, all the "good" σ's can be encoded in an encoding space of size at most*

$$2^s p! \left(1 + \frac{6q}{p}\right)^{(2q-l)} \left(\frac{p}{8q^2(2k+2+3q)}\right)^{-l},$$

and decoded correctly with probability ϵ.

We next give some intuition regarding how we achieve compression and defer the formal proof of Lemma 9 to the full version.

Intuition regarding compression. Observe in $\mathbb{G}_{13}$, the labels in $\mathcal{Z}$ were queried by $\mathcal{R}_2$ (these labels were not seen by $\mathcal{R}_2$ before they were queried) and were answers to $\mathcal{R}_1$ and were not seen by $\mathcal{R}_1$ before the query. The core idea is that for all $\mathbf{a} \in \mathcal{L} \setminus \mathcal{Z}$, we store exactly one of $\mathbf{a}$ or its pre-image in the encoding and for all labels in $\mathcal{Z}$, we store neither the label nor its pre-image. Since $\mathcal{R}_2$ queries all the labels in $\mathcal{Z}$, these labels can be found by running $\mathcal{R}_2$ while decoding. Since all the labels in $\mathcal{Z}$ are answers to queries of $\mathcal{R}_1$ and were not seen by $\mathcal{R}_1$ before the query, their pre-images can be figured out while running $\mathcal{R}_1$.

High level outlines of Encode, Decode. In Encode, we simulate the steps of $\mathbb{G}_{13}$ to $\mathcal{R}_1, \mathcal{R}_2$, including bookkeeping and then run $\mathcal{R}_1$ again assuming the particular σ we are compressing is sampled in $\mathbb{G}_{13}$. In Decode, we run $\mathcal{R}_2$ and then $\mathcal{R}_1$ to recover σ. We treat the values $i_1, \cdots, i_k, v, \pi$ as part of the common randomness provided to $\mathsf{Encode}, \mathsf{Decode}$ (we assume they are sampled from the same distribution they are sampled from in $\mathbb{G}_{13}$). The random tapes of $\mathcal{R}_1, \mathcal{R}_2$

can also be derived from the common randomness of Encode, Decode. For simplicity, we do not specify this explicitly in the algorithms and treat $\mathcal{R}_1, \mathcal{R}_2$ as deterministic.

RUNNING $\mathcal{R}_2$. First off, we assume that $\mathcal{R}_1$ queries labels that it has "seen" before and $\mathcal{R}_2$ queries labels that $\mathcal{R}_1$ has "seen" or it has "seen" before. We shall relax this assumption later. Ideally, we would want to just store only ϕ, the inputs labels to $\mathcal{R}_2$ and the labels that are answers to $\mathcal{R}_2$'s queries. We append the input labels of $\mathcal{R}_2$ and labels that are answers to its Eval queries that it has not "seen" before to a list named Labels. However, it is easy to see that this information is not enough to answer O_v queries during decoding, as answering O_v queries inherently requires knowledge about pre-images of $\mathcal{R}_2$. This naturally leads to the idea of maintaining a mapping of all the labels "seen by" $\mathcal{R}_2$ to their pre-images.

THE MAPPING T OF LABELS TO PRE-IMAGE EXPRESSIONS. The pre-images of input labels and the labels that were results of sequence of Eval queries on its input labels by $\mathcal{R}_2$, are known. However, $\mathcal{R}_2$ might query labels which were neither an input to it nor an answer to one of its Eval queries. Such a label is in $\mathcal{Z}$ since we have assumed that all labels queried by $\mathcal{R}_2$ were "seen by" $\mathcal{R}_1$ or "seen by" $\mathcal{R}_2$ before. We represent the pre-images of labels in $\mathcal{Z}$ using a placeholder variable X_n where n is incremented for every such label. Note that the pre-image of every label seen by $\mathcal{R}_2$ can be expressed as a linear polynomial in the X_n's (these linear polynomials are referred to as pre-image expressions from hereon). Therefore we maintain a mapping of all labels "seen by" and their pre-image expressions in a list of tuples named T. Our approach is inspired by a similar technique used by Corrigan-Gibbs and Kogan in [5]. Like in [5], we *stress* that the mapping T is not a part of the encoding.

For Eval queries, we can check if there is a tuple in T whose pre-image expression is the sum of the pre-image expressions of the input labels. If that is the case, we return the label of such a tuple. Otherwise, we append the answer label to Labels. For O_v queries, we can return true if the pre-image expression of the first input label multiplied by v gives the pre-image expression of the second input label. Otherwise we return false.

SURPRISES.There is a caveat, however. There might arise a situation that the label which is the answer to the Eval query is present in T but its pre-image expression is not the sum of the pre-image expressions of the input labels. We call such a situation a "surprise" and we call the answer label in that case a "surprise label". For O_v queries, there might be a surprise when the answer of the O_v query is true but the pre-image expression of the first input label multiplied by v is different pre-image expression of the second input label. In this case we call the second input label the surprise label. We assign a sequence number to each query made by $\mathcal{R}_2$, starting from 1 and an index to each tuple in T, with the indices being assigned to tuples in the order they were appended to T. To detect the query where the surprise happens, we maintain a set named Srps_1 that contains tuples of query sequence numbers and indices of the surprise label in T. This set Srps_1 is a part of the encoding. Note that whenever there is a surprise, it means that two different pre-image expressions evaluate to the same value. Since these two pre-image expressions are linear polynomials, at least one variable can be eliminated from T by equating the two pre-image expressions.

Running $\mathcal{R}_1$. Now that we have enough information in the encoding to run $\mathcal{R}_2$, we consider the information we need to add to the encoding to run $\mathcal{R}_1$ after $\mathcal{R}_2$ is run. First, we need to provide $\mathcal{R}_1$ its input labels. Our initial attempt would be to append the input labels of $\mathcal{R}_1$ (except $\sigma(1), \sigma(v)$, which are already present) to Labels. However, some of these input labels to $\mathcal{R}_1$ might have already been "seen by" $\mathcal{R}_2$. Since all labels "seen by" $\mathcal{R}_2$ are in T, we need a way to figure out which of $\sigma(i_j)$'s are in T. Note that such a label was either queried by $\mathcal{R}_2$ or an answer to a query of $\mathcal{R}_2$ (cannot have been an input to $\mathcal{R}_2$ given the restrictions on $i_1, \cdots, i_k, v$). Suppose q was the sequence number of the query in which $\sigma(i_j)$ was queried or an answer. The tuple (q, b, j) is added to the set Inputs where b can take values $\{1, 2, 3\}$ depending on whether $\sigma(i_j)$ was the first input label, the second input label or the answer label respectively. This set Inputs is a part of the encoding. The rest of the labels $\sigma(i_j)$, which do not appear in T, are added to T with their pre-images and the labels are appended to Labels. Note that for all queries of $\mathcal{R}_1$, it follows from our assumption that the input labels will be in T. For every surprise, we add a tuple of sequence number and an index in T to the set Srps_2.

Relaxing the assumption. When we allow $\mathcal{R}_2$ to query labels it has not seen before or $\mathcal{R}_1$ has not seen, there are two issues. First, we need to add a tuple for the label in T (since T, by definition contains a tuple for all labels queried by $\mathcal{R}_2$). We solve this issue by adding the tuple made of the label and its pre-image. We have no hope of recovering the pre-image later, hence, we append the pre-image to a list named Vals. This list needs to be a part of the encoding since the pre-image of the label needs to be figured out to be added to T during decoding. For queries of $\mathcal{R}_1$, if the input label is not present in T, we do the same thing. The second issue that comes up when we relax the assumption is that we need to distinguish whether an input label was in $\mathcal{Z}$ or not. We solve this issue by maintaining a set of tuples named Free. For all labels in $\mathcal{Z}$ that are not an input label to $\mathcal{R}_1$, we add the tuple consisting of the sequence number of the query of $\mathcal{R}_2$ and b to Free where b set to 1 indicates it was the first input label and b set to 2 indicates it was the second input label.

The final steps. The labels the are absent in T are appended to a list named $\mathsf{RLabels}$. If $|\mathcal{Z}| < l$, a fixed encoding D (the output of Encode for some fixed σ when $|\mathcal{Z}| \geqslant l$) is returned. Otherwise the encoding of σ consisting of Labels, $\mathsf{RLabels}$, Vals, Inputs, Srps_1, Srps_2, Free, ϕ is returned.

Wrapping up.The set of all "good" σ's has size at least $\epsilon p!$ (where we have used that the total number of injective functions from $\mathbb{Z}_p \to \mathcal{L}$ is $p!$). Using $\mathcal{X}$ to be the set of the "good" σ's, $\mathcal{Y}$ to be the set of encodings, $\mathcal{R}$ to be the set of cartesian product of the domains of $i_1, \cdots, i_k, v, \pi$, the set of all random tapes of $\mathcal{R}_1$ the set of all random tapes of $\mathcal{R}_2$ and $\mathcal{L}$, it follows from Lemma 9 and

Proposition 1 that

$$\log(\Pr[\text{Decoding is correct}]) \leqslant s + (2q-l)\log\left(1+\frac{6q}{p}\right) - l\log\left(\frac{p}{8q^2(2k+2+3q)}\right) - \log\epsilon\,.$$

We have from Lemma 9 that $\Pr[\text{Decoding is correct}] \leqslant \epsilon$. Therefore,

$$2\log\epsilon \leqslant s + (2q-l)\log\left(1+\frac{6q}{p}\right) - l\log\left(\frac{p}{8q^2(2k+2+3q)}\right).$$

Since $\Pr[\mathbb{G}_{13}] = 2\epsilon$, using (7) and (8) we have,

$$\Pr[\mathbb{G}_3 \Rightarrow \mathsf{true}] \leqslant 2\cdot 2^{\frac{s}{2}}\left(\frac{8q^2(2k+2+3q)}{p}\right)^{\frac{l}{2}}\left(1+\frac{6q}{p}\right)^{\frac{2q-l}{2}} + \frac{k^2+k+2}{p}\,.$$

□

5 Conclusions

Despite a clear restriction of our result to straightline reductions, we believe the main contribution of this work is the introduction of novel techniques for proving lower bounds on the memory of reductions that will find wider applicability. In particular, we clearly departed from the framework of prior works [2,13] tailored at the usage of lower bounds for streaming algorithms, and provided the first lower bound for "algebraic" proofs in the public-key domain. The idea of a problem-specific proof of memory could be helpful elsewhere.

Of course, there are several open problems. It seems very hard to study the role of rewinding for such reductions. In particular, the natural approach is to resort to techniques from communication complexity (and their incarnation as streaming lower bounds), as they are amenable to the multi-pass case. The simple combinatorial nature of these lower bounds however is at odds with the heavily structured oracles we encounter in the generic group model. Another problem we failed to solve is to give an adversary $\mathcal{A}$ in our proof which uses little memory – we discuss a candidate in the body, but analyzing it seems to give us difficulties similar to those of rewinding.

This latter point makes a clear distinction, not discussed by prior works, between the *way* in which we prove memory-tightness (via reductions using small memory), and its most general interpretation, as defined in [2], which would allow the reduction to adapt its memory usage to that of $\mathcal{A}$.

Acknowledgements. We thank the anonymous reviewers of EUROCRYPT 2020 for helpful comments. This work was partially supported by NSF grants CNS-1553758 (CAREER), CNS-1719146, and by a Sloan Research Fellowship.

References

1. Abdalla, M., Bellare, M., Rogaway, P.: The Oracle Diffie-Hellman assumptions and an analysis of DHIES. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 143–158. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45353-9_12
2. Auerbach, B., Cash, D., Fersch, M., Kiltz, E.: Memory-tight reductions. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017. LNCS, vol. 10401, pp. 101–132. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63688-7_4
3. Bellare, M., Rogaway, P.: The security of triple encryption and a framework for code-based game-playing proofs. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 409–426. Springer, Heidelberg (2006). https://doi.org/10.1007/11761679_25
4. Bhattacharyya, R.: Memory-tight reductions for practical key encapsulation mechanisms. In: PKC (2020)
5. Corrigan-Gibbs, H., Kogan, D.: The discrete-logarithm problem with preprocessing. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part II. LNCS, vol. 10821, pp. 415–447. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_14
6. Cramer, R., Shoup, V.: Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. SIAM J. Comput. **33**(1), 167–226 (2003)
7. De, A., Trevisan, L., Tulsiani, M.: Time space tradeoffs for attacks against one-way functions and PRGs. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 649–665. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_35
8. Fuchsbauer, G., Kiltz, E., Loss, J.: The algebraic group model and its applications. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018, Part II. LNCS, vol. 10992, pp. 33–62. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96881-0_2
9. Maurer, U.: Abstract models of computation in cryptography. In: Smart, N.P. (ed.) Cryptography and Coding 2005. LNCS, vol. 3796, pp. 1–12. Springer, Heidelberg (2005). https://doi.org/10.1007/11586821_1
10. Reingold, O., Trevisan, L., Vadhan, S.: Notions of reducibility between cryptographic primitives. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 1–20. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24638-1_1
11. Shoup, V.: Lower bounds for discrete logarithms and related problems. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 256–266. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-69053-0_18
12. Shoup, V.: A proposal for an ISO standard for public key encryption. Cryptology ePrint Archive, Report 2001/112 (2001). http://eprint.iacr.org/2001/112
13. Wang, Y., Matsuda, T., Hanaoka, G., Tanaka, K.: Memory lower bounds of reductions revisited. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 61–90. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78381-9_3