Generic-Group Delay Functions Require Hidden-Order Groups

Lior Rotem, Gil Segev(✉), and Ido Shahaf(✉)

School of Computer Science and Engineering, Hebrew University of Jerusalem, 91904 Jerusalem, Israel
{lior.rotem,segev,ido.shahaf}@cs.huji.ac.il

Abstract. Despite the fundamental importance of delay functions, underlying both the classic notion of a time-lock puzzle and the more recent notion of a verifiable delay function, the only known delay function that offers both sufficient structure for realizing these two notions and a realistic level of practicality is the "iterated squaring" construction of Rivest, Shamir and Wagner. This construction, however, is based on rather strong assumptions in groups of hidden orders, such as the RSA group (which requires a trusted setup) or the class group of an imaginary quadratic number field (which is still somewhat insufficiently explored from the cryptographic perspective). For more than two decades, the challenge of constructing delay functions in groups of known orders, admitting a variety of well-studied instantiations, has eluded the cryptography community.

In this work we prove that there are no constructions of generic-group delay functions in cyclic groups of known orders: We show that for any delay function that does not exploit any particular property of the representation of the underlying group, there exists an attacker that completely breaks the function's sequentiality when given the group's order. As any time-lock puzzle and verifiable delay function give rise to a delay function, our result holds for these two notions we well, and explains the lack of success in resolving the above-mentioned long-standing challenge. Moreover, our result holds even if the underlying group is equipped with a d-linear map, for any constant $d \geq 2$ (and even for super-constant values of d under certain conditions).

1 Introduction

The classic notion of a time-lock puzzle, introduced by Rivest, Shamir and Wagner [RSW96], and the recent notion of a verifiable delay function, introduced by Boneh et al. [BBB+18], are instrumental to a wide variety of exciting

L. Rotem, G. Segev and I. Shahaf—Supported by the European Union's Horizon 2020 Framework Program (H2020) via an ERC Grant (Grant No. 714253).
L. Rotem—Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.
I. Shahaf—Supported by the Clore Israel Foundation via the Clore Scholars Programme.

A. Canteaut and Y. Ishai (Eds.): EUROCRYPT 2020, LNCS 12107, pp. 155–180, 2020.
https://doi.org/10.1007/978-3-030-45727-3_6

applications, such as randomness beacons, resource-efficient blockchains, proofs of replication and computational timestamping. Underlying both notions is the basic notion of a cryptographic delay function: For a delay parameter T, evaluating a delay function on a randomly-chosen input should require at least T sequential steps (even with a polynomial number of parallel processors and with a preprocessing stage), yet the function can be evaluated on any input in time polynomial in T (e.g., $2T$ or T^4).[1]

A delay function can be easily constructed by iterating a cryptographic hash function, when modeled as a random oracle for proving its sequentiality. However, the complete lack of structure that is offered by this construction renders its suitability for realizing time-lock puzzles or verifiable delay functions rather unclear. Specifically, for time-lock puzzles, iterating a cryptographic hash function in general does not enable sufficiently fast generation of input-output pairs. Similarly, for verifiable delay functions, iterating a cryptographic hash function in general does not able sufficiently fast verification (although, asymptotically, such verification can be based on succinct non-interactive arguments for NP languages [Kil92, Mic94, GW11], as suggested by Döttling et al. [DGM+19] and Boneh et al. [BBB+18]).

The only known construction of a delay function that offers both a useful structure for realizing time-lock puzzles or verifiable delay functions and a realistic level of practicality is the "iterated squaring" construction underlying the time-lock puzzle of Rivest et al. [RSW96], which was recently elegantly extended by Pietrzak [Pie19] and Wesolowski [Wes19] to additionally yield a verifiable delay function. The iterated squaring construction, however, is based on rather strong assumptions in groups of hidden orders such as the RSA group or the class group of an imaginary quadratic number field. Unfortunately, RSA groups require a trusted setup stage as the factorization of the RSA modulus serves as a trapdoor enabling fast sequential evaluation [RSW96, Pie19, Wes19], and the class group of an imaginary quadratic number field is not as well-studied from the cryptographic perspective as other, more standard, cryptographic groups [BBF18, Sec. 6].

Thus, a fundamental goal is to construct delay functions in groups of known orders, giving rise to a variety of well-studied instantiations. In such groups, the security of delay functions can potentially be proved either based on long-standing cryptographic assumptions or within the generic-group model as a practical heuristic.

1.1 Our Contributions

In this work we prove that there are no constructions of generic-group delay functions in cyclic groups of known orders: Roughly speaking, we show that for any delay function that does not exploit any particular property of the representation of the underlying group, there exists an attacker that breaks the

[1] We refer the reader to Sect. 2 for a formal definition of a delay function, obtained as a natural relaxation of both a time-lock puzzle and a verifiable delay function.

function's sequentiality when given the group's order. As any time-lock puzzle and verifiable delay function give rise to a delay function, our result holds for these two notions as well. Moreover, our impossibility result holds even if the underlying group is equipped with a d-linear map, for any constant $d \geq 2$ and even for super-constant values of d under certain conditions as discussed below.

Our result: Attacking delay functions in known-order groups. Generic-group algorithms have access to an oracle for performing the group operation and for testing whether two group elements are equal, and the efficiency of such algorithms is measured mainly by the number of oracle queries that they issue [Nec94, Sho97, BL96, MW98, Mau05]. In the context of generic-group delay functions, we view generic-group algorithms as consisting of parallel processors, and we measure the number of such processors together with the number of sequential queries that are issued by each such processor. In addition, we measure the amount of any internal computation that is performed by our attacker, and this enables us to prove an impossibility result that is not only of theoretical significance in the generic-group model, but is also of practical significance.

The following theorem presents our main result in an informal and simplified manner that focuses on prime-order groups without d-linear maps, and on delay functions whose public parameters, inputs and outputs consist only of group elements[2]:

Theorem (informal & simplified). *Let* DF *be a generic-group delay function whose public parameters, inputs and outputs consist of* $k_{\mathsf{pp}}(\lambda, T)$, $k_{\mathsf{in}}(\lambda, T)$ *and* $k_{\mathsf{out}}(\lambda, T)$ *group elements, respectively, where* $\lambda \in \mathbb{N}$ *is the security parameter and* $T \in \mathbb{N}$ *is the delay parameter. Let* $Q_{\mathsf{eqEval}}(\lambda, T)$ *denote the number of equality queries issued by the function's honest evaluation algorithm. Then, there exists a generic-group attacker* $\mathcal{A}$ *that takes as input the* λ*-bit order* p *of the group such that:*

- $\mathcal{A}$ *correctly computes the function on any input.*
- $\mathcal{A}$ *consists of* $(k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot \max\{k_{\mathsf{out}}, Q_{\mathsf{eqEval}}\}$ *parallel processors, each of which issues at most* $O((k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot \log p)$ *sequential oracle queries.*

For interpreting our theorem, first note that our attacker does not require a preprocessing stage, and is able to correctly compute the function on any input (these rule out even an extremely weak notion of sequentiality).

Second, note that the number $(k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot \max\{k_{\mathsf{out}}, Q_{\mathsf{eqEval}}\}$ of parallel processors used by our attacker is at most polynomial in the security parameter λ and in the delay parameter T, and that the number $O((k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot \log p)$ of sequential queries issued by each processor is polynomial in λ and essentially independent of the delay parameter T. Specifically, for delay functions underlying time-lock puzzles and verifiable delay functions, the parameters k_{pp}, k_{in} and

[2] As discussed in Sect. 1.3, we prove our result also to groups of composite order, to groups equipped with a d-linear map, and to delay functions whose public parameters, inputs and outputs consist of both group elements and arbitrary additional values.

k_{out} are all polynomials in λ and $\log T$ (for the iterated squaring delay function, for example, it holds that $k_{\mathsf{pp}} = Q_{\mathsf{eqEval}} = 0$ and $k_{\mathsf{in}} = k_{\mathsf{out}} = 1$).[3] Therefore, in these cases the number of sequential queries issued by each processor is at most polynomial in λ and $\log T$.

An additional interpretation of our result is as follows. The term $\max\{k_{\mathsf{out}}, Q_{\mathsf{eqEval}}\}$ lower bounds the time to compute Eval without parallelism (even though it could be much smaller – as for the iterated squaring function). Optimally, an α speedup, that is, computing the function α times faster than without parallelism, is obtained by using α parallel processors. We show that an (at least) α speedup can be obtained by using $O(\alpha \cdot (k_{\mathsf{pp}} + k_{\mathsf{in}})^2 \cdot \log p)$ parallel processors.

1.2 Related Work

Various cryptographic notions that share a somewhat similar motivation with delay functions have been proposed over the years, such as the above-discussed notions of time-lock puzzles and verifiable delay functions (e.g., [RSW96,BGJ+16,BBB+18,BBF18,Pie19,Wes19,EFK+19,DMP+19]), as well as other notions such as sequential functions and proofs of sequential work (e.g., [MMV11,MMV13,CP18]). It is far beyond the scope of this work to provide an overview of these notions, and we refer the reader to the work of Boneh et al. [BBB+18] for an in-depth discussion of these notions and of the relations among them.

A generic-group candidate for a function that requires more time to evaluate than to verify was proposed by Dwork and Naor [DN92] based on extracting square roots modulo a prime number p (see also the work of Lenstra and Wesolowski [LW15] on composing several such functions). However, the time required to sequentially evaluate this function, as well as the gap between the function's sequential evaluation time and its verification time, both seem limited to $O(\log p)$, and thus cannot be flexibly adjusted via a significantly larger delay parameter T. As noted by Boneh et al. [BBB+18], this does not meet the notion of a verifiable delay function (or our less-strict notion of a delay function).

In the random-oracle model, Döttling, Garg, Malavolta and Vasudevan [DGM+19], and Mahmoody, Smith and Wu [MSW19] proved impossibility results for certain classes of verifiable delay functions (and, thus, in particular, for certain classes of delay functions). Before describing their results, we note that whereas Döttling et al. and Mahmoody et al. captured restricted classes of verifiable delay functions within the random-oracle model, our work captures all constructions of delay functions (a more relaxed notion) within the incomparable generic-group model. Most importantly, in the random-oracle model a delay function can be easily constructed by iterating the random oracle (however, as discussed above, this does not seem practically useful for realizing time-lock puzzles or verifiable delay functions).

[3] For time-lock puzzles this follows from the requirement that an input-output pair can be generated in time polynomial in λ and $\log T$, and for verifiable delay functions this follows from the requirement that the verification algorithm runs in time polynomial in λ and $\log T$.

The work of Döttling et al. rules out constructions of verifiable delay functions with a *tight* gap between the assumed lower bound on their sequential evaluation time and their actual sequential evaluation time. Specifically, they proved that there is no construction that cannot be evaluated using less than T sequential oracle queries (even with parallel processors), but can be evaluated using $T + O(T^{\delta})$ sequential oracle queries (for any constant $\delta > 0$ where T is the delay parameter). Note, however, that this does not rule out constructions that cannot be evaluated using less than T sequential oracle queries but can be evaluated, say, using $4T$ or $T \log T$ sequential oracle queries. In addition to their impossibility result, Döttling et al. showed that any verifiable delay function with a prover that runs in time $O(T)$ and has a natural self-composability property can be generically transformed into a verifiable delay function with a prover that runs in time $T + O(1)$ based on succinct non-interactive arguments for NP languages [Kil92, Mic94, GW11].

The work of Mahmoody et al. rules out constructions of verifiable delay functions that are *statistically* sound with respect to *any* oracle[4]. That is, they consider verifiable delay functions whose soundness property holds for *unbounded* adversaries and holds completely independently of the oracle. As noted by Mahmoody et al. this suffices, for example, for ruling out verifiable delay functions that are permutations. However, for such functions that are not permutations, this strong soundness property does not necessarily hold – as the security of constructions in the random-oracle model is on based the randomness of the oracle (and does not hold with respect to any oracle).

1.3 Overview of Our Approach

In this section we give an informal technical overview of our approach. We start by reviewing the generic-group model in which our lower bound is proven, and then move on to describe our attack, first in simplified settings and then gradually building towards our full-fledged attack. Finally, we illustrate how this attack can be extended to rule out generic-group delay functions in groups equipped with multilinear maps.

The Framework. We prove our impossibility result within the generic-group model introduced by Maurer [Mau05], which together with the incomparable model introduced by Shoup [Sho97], seem to be the most commonly-used approaches for capturing generic group computations. At a high level, in both models algorithms have access to an oracle for performing the group operation and for testing whether two group elements are equal. The difference between the two models is in the way that algorithms specify their queries to the oracle. In Maurer's model algorithms specify their queries by pointing to two group elements that have appeared in the computation so far (e.g., the 4th and the 7th group elements), whereas in Shoup's model group elements have an explicit

[4] In fact, as pointed out by Mahmoody et al. their impossibility result holds also for proofs of sequential work.

representation (sampled uniformly at random from the set of all injective mappings from the group to sufficiently long strings) and algorithms specify their queries by providing two strings that have appeared in the computation so far as encoding of group elements.

Jager and Schwenk [JS08] proved that the complexity of any computational problem that is defined in a manner that is independent of the representation of the underlying group (e.g., computing discrete logarithms) in one model is essentially equivalent to its complexity in the other model. However, not all generic cryptographic constructions are independent of the underlying representation.

More generally, these two generic-group models are rather incomparable. On one hand, the class of cryptographic schemes that are captured by Maurer's model is a subclass of that of Shoup's model – although as demonstrated by Maurer his model still captures all schemes that only use the abstract group operation and test whether two group elements are equal. On the other hand, the same holds also for the class of adversaries, and thus in Maurer's model we have to break the security of a given scheme using an adversary that is more restricted when compared to adversaries in Shoup's model. In fact, Shoup's model is "sufficiently non-generic" to accommodate delay functions such as the iterated-hashing construction. Delay functions of such flavor, however, rely on the randomness of the representation of group elements, which may or may not be sufficient in specific implementations of concrete groups, and are not based solely on the underlying algebraic hardness as in Maurer's model. Furthermore, as discussed earlier, delay functions that exploit such randomness are somewhat unstructured, and thus seem limited in their applicability to the design of time-lock puzzles and VDFs (for time-lock puzzles insufficient structure may not enable sufficiently fast generation of input-output pairs, and for VDFs insufficient structure may not enable sufficiently fast verification). We refer the reader to Sect. 2.1 for a formal description of Maurer's generic-group model.

Generic-group delay functions. A generic-group delay function in a cyclic group of order N is defined by an evaluation algorithm Eval, which receives the public parameters pp and an input $\mathbf{x}$, and returns an output $\mathbf{y}$. For the sake of this overview, we assume that pp, $\mathbf{x}$ and $\mathbf{y}$ consist of k_{pp}, k_{in} and k_{out} group elements, respectively (we refer the reader to Sect. 5 for a detailed account of how we handle additional explicit bit-strings as part of the public parameters, input and output). As a generic algorithm, Eval's access to these group elements is implicit and is provided via oracle access as follows. At the beginning of its execution, a table $\mathbf{B}$ is initialized with $\mathbb{Z}_N$ elements which correspond to the elements in pp and in $\mathbf{x}$. Eval can then access the table via two types of queries: (1) group operation queries, which place the sum of the two $\mathbb{Z}_N$ elements in the entries pointed to by Eval in the next vacant entry of the table; and (2) equality queries, which return 1 if and only if the two $\mathbb{Z}_N$ elements in the entries pointed to by Eval are equal. At the end of its execution, in order to implicitly output group elements, Eval outputs the indices of entries in the table in which the output group elements are positioned. We refer the reader to Sect. 2.2 for a more formal presentation of generic-group delay functions.

A simplified warm-up. Our goal is to construct an attacker, which (implicitly) receives the public parameters pp and an input $\mathbf{x}$, and computes the corresponding output $\mathbf{y}$ in a sequentially-fast manner. As a starting point, consider an oversimplified and hypothetical scenario in which the attacker is provided not only with oracle access to the table $\mathbf{B}$, but also with the explicit $\mathbb{Z}_N$ elements which are in the table and that correspond to pp and to $\mathbf{x}$. In this case, an attacker can simply emulate the execution of Eval locally without any queries to the oracle, where instead of the oracle table $\mathbf{B}$, the attacker keeps a local table of $\mathbb{Z}_N$ elements: Group oracle queries are emulated via integer addition modulo N, and equality queries are answered in accordance with integer equality. At the end of this emulation, the attacker holds the $\mathbb{Z}_N$ elements that correspond to the output elements of Eval. A key observation is that translating each of these $\mathbb{Z}_N$ elements into the appropriate group element – i.e., placing this $\mathbb{Z}_N$ element in the table $\mathbf{B}$ – requires only $O(\log N) = O(\lambda)$ oracle queries (e.g., via the standard square-then-multiply method).[5] Moreover, for any number k_{out} of group elements in the function's output, the number of sequential oracle queries remains only $O(\lambda)$ when using k_{out} parallel processors – one per each output element.

As an intermediate step towards our full-fledged attack, consider a somewhat less hypothetical scenario, in which the attacker only gets implicit access to the group elements in pp and in $\mathbf{x}$, but Eval does not issue any equality queries. Observe that this setting already captures the widely-used iterated squaring delay function discussed above. The main idea behind our attack in this setting is to replace each of the input group elements to Eval with a formal variable, and then to symbolically compute each output element as a polynomial in these variables. Note that in general, these are not fixed polynomials, but rather depend on the equality pattern resulting from Eval's equality queries. Here, however, we are assuming that Eval does not issue any such queries. Concretely, when there are no equality queries, computing the output polynomials does not require any oracle queries by a similar emulation to the one described above, where values in the local table are stored as polynomials, and the group operation is replaced with polynomial addition. Once we have each of the output elements expressed as a polynomial, we can implicitly evaluate it at $(\mathsf{pp}, \mathbf{x})$, starting with implicit access to the elements in $(\mathsf{pp}, \mathbf{x})$, using $k_{\mathsf{pp}} + k_{\mathsf{in}}$ parallel processors each of which performing $O(\log N + \log(k_{\mathsf{pp}} + k_{\mathsf{in}})) = O(\lambda)$ sequential group operations.[6]

Handling equality queries. On the face of it, the attack described in the previous paragraph is not applicable when Eval does issue equality queries, since it is unclear how to answer such queries in the polynomial-based emulation of

[5] We assume that the first entry of the table $\mathbf{B}$ is always occupied with the number 1, which is always a generator for $\mathbb{Z}_N$.

[6] Note that implicitly evaluating each monomial using roughly $\log N$ sequential group operations requires knowing the precise order N of the group. Without knowing N, this polynomial may have coefficients which are exponentially large in the delay parameter T, and evaluating each monomial can take up to $\mathsf{poly}(T)$ sequential group operations.

Eval. One possibility is to answer each equality query in the affirmative if and only if the two elements pointed to by Eval are identical as polynomials in the formal variables replacing the input elements. Indeed, if the two polynomials are identical, it is necessarily the case that the two elements are equal. Unfortunately, the opposite is incorrect, and it is possible (and indeed to be expected) that the two elements will be equal even though their corresponding polynomials are not identical, resulting in a false negative answer and thus the emulation will deviate from the true execution of Eval.

The main observation underlying our attack is that even though the number Q_{eqEval} of equality queries that Eval issues might be quite large (potentially as large as the delay parameter T), at most $|\mathsf{factors}(N)|\cdot(k_{\mathsf{pp}}+k_{\mathsf{in}})$ of the non-trivial queries can be affirmatively answered, where $\mathsf{factors}(N)$ denotes the multi-set of prime factors of N (where the number of appearances of each primes factor is its multiplicity – e.g., $\mathsf{factors}(100) = \{2, 2, 5, 5\}$), and by trivial queries we mean queries for which equality or inequality follows from the previous query/answer pattern. This is the case because at each point during the execution of Eval, the set of possible values for $(\mathsf{pp}, \mathbf{x})$, given the equality pattern so far, is a coset of some subgroup $H \leq \mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}$ relative to $(\mathsf{pp}, \mathbf{x})$: The possible values for $(\mathsf{pp}, \mathbf{x})$ are a set of the form $\{(\mathsf{pp}, \mathbf{x}) + (\mathsf{pp}', \mathbf{x}') | (\mathsf{pp}', \mathbf{x}') \in H\}$, where initially $H = \mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}$. Moreover, if q is a non-trivial equality query answered affirmatively, H is the said subgroup before q is issued and H' is the subgroup after q is answered, then due to the non-triviality of q, it is necessarily the case that $H' < H$ (i.e., H' is a proper subgroup of H). In particular, the order of H' is smaller than the order of H and divides it. Hence, since $|\mathsf{factors}(|\mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}|)| = (k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)|$, the observation follows by induction.

Utilizing the Power of Parallelism. We translate this observation into an attack on the sequentiality of any generic-group delay function by carefully utilizing the power of parallelism in the following manner. Our attacker keeps track of an initially empty set $\mathcal{L}$ of linear equations in the formal variables that replace pp and $\mathbf{x}$, and runs for $(k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| + 1$ iterations.[7] In each iteration, the attacker runs the polynomial-based emulation described above, with the exception that now equality queries are answered affirmatively if and only if equality follows from the equations in $\mathcal{L}$. The attacker then checks, by querying the oracle, if any of the negatively-answered queries in the emulation should have been answered affirmatively, and if so, the equality that follows from this query is added to $\mathcal{L}$ – this step can be executed using $Q_{\mathsf{eqEval}} \cdot (k_{\mathsf{pp}} + k_{\mathsf{in}})$ parallel processors, each of which issuing $O(\log N + \log(k_{\mathsf{pp}} + k_{\mathsf{in}})) = O(\lambda)$ sequential queries.

Since we make sure that the true $(\mathsf{pp}, \mathbf{x})$ is always in the solution set of $\mathcal{L}$, there will be no false positive answers, and in each iteration there are only two possibilities: Either there exists a false negative answer (which we will then add

[7] We emphasize that our attack does not require knowing the factorization of N. Since $|\mathsf{factors}(N)| \leq \log N$, one can replace $|\mathsf{factors}(N)|$ with $\log N$ when determining the number of iterations.

to $\mathcal{L}$ as an equality) or all queries are answered correctly. On the one hand, if all queries are answered correctly, then the emulation in this iteration is accurate and we are done – all that is left is to translate the output polynomials of this emulation into implicit group elements, which we already discussed how to do. On the other hand, if there exists a false negative answer, then we learn a new equation that does not follow from the equations already in $\mathcal{L}$. By our observation, we can learn at most $|\mathsf{factors}(N)| \cdot (k_{\mathsf{pp}} + k_{\mathsf{in}})$ new such equations, so there must be an iteration in which we successfully emulate the execution of Eval and compute the correct output of the function.

Attacking generic delay functions in multilinear groups. We extend our attack so that it computes the output of any generic delay function in groups that are equipped with a d-linear map and on any input, while issuing at most $O((k_{\mathsf{pp}} + k_{\mathsf{in}} + 1)^d \cdot |\mathsf{factors}(N)| \cdot \lambda)$ sequential queries. In such groups, in addition to the group operation and equality queries, generic algorithms can also issue d-linear map queries, supplying (implicitly) d elements in the source group and receiving as a reply implicit access to the resulting element of the target group. In our polynomial-based emulation of Eval described above, we replace such queries with polynomial multiplication, resulting in polynomials of degree at most d. Since these polynomials may be non-linear, and the analysis of our attack heavily relied on the fact that the learned equations are linear, this analysis no longer applies.

We address this situation by carefully employing a linearization procedure. Roughly speaking, in our polynomial-based emulation of Eval, the attacker now replaces each possible product of at most d formal variables (out of the formal variables that replace the group elements in pp and in $\mathbf{x}$) with a single new formal variable. After applying this linearization procedure, the learned equations are once again linear (in the new formal variables), but by applying it, we lose information about the possible set of assignments to the elements in $(\mathsf{pp}, \mathbf{x})$, given the learned equations in $\mathcal{L}$. As a result, it might be that a certain equality which follows from the equations in $\mathcal{L}$, no longer follows from them after applying the linearization procedure (to both the equality and the equations in $\mathcal{L}$). The main observation that makes our attack successful nevertheless is that if a certain equality follows from $\mathcal{L}$ after applying the linearization procedure, it necessarily followed from $\mathcal{L}$ before applying the procedure as well. Hence, it is still the case that there are no false positive answers in the emulation, and that in each iteration we either add a new equation to $\mathcal{L}$ or compute the correct output.

This linearization procedure comes at a cost. After applying it, we have $(k_{\mathsf{pp}} + k_{\mathsf{in}} + 1)^d$ different formal variables instead of just $k_{\mathsf{pp}} + k_{\mathsf{in}}$ as before. Thus, in order for our analysis from the linear setting to apply, our attacker needs to run for roughly $(k_{\mathsf{pp}} + k_{\mathsf{in}} + 1)^d \cdot |\mathsf{factors}(N)|$ iterations, explaining the exponential dependency on d in its sequential query complexity. Note however that the attack still computes the output with less than T sequential queries as long as $d \leq O(\log T / (\log \lambda \cdot \log(k_{\mathsf{pp}} + k_{\mathsf{in}})))$, and in particular whenever d is constant.

Our attacker's internal computation. In order to rule out constructions of delay functions whose sequentiality is proven within the generic-group model, it suffices to present an attacker which is efficient relative to the security parameter and the delay parameter in terms of its number of parallel processors and generic-group operations, regardless of the amount of additional internal computation required by the attacker. Nevertheless, we show that our attacker requires an overhead which is only polynomial in terms of its internal computation. Consequently, when our attack is applied to any "heuristically secure" construction in any cyclic group of known order, the number of sequential group operations it performs is essentially independent of T, and the additional computation – which is independent of the specific group in use – is at most $\mathsf{poly}(\lambda, T)$. Put differently, either this additional computation can be sped-up using parallelism and then the construction is insecure; or it cannot be sped-up and thus yields an inherently-sequential computation that does not rely on the underlying group.

Specifically, the most significant operation that is performed by our attacker which is non-trivial in terms of its computational cost is checking in each iteration whether or not a given linear equation over $\mathbb{Z}_N$ follows from the linear equations already in the set $\mathcal{L}$. When considering groups of prime order, this can be done simply by testing for linear independence among the vectors of coefficients corresponding to these equations. When considering groups of composite order this is a bit more subtle, and can be done for example by relying on fast algorithms for computing the Smith normal form of integer matrices (e.g., [Sto96]) and without knowing the factorization of the order of the group – see Appendix A for more details.

1.4 Paper Organization

The remainder of this paper is organized as follows. First, in Sect. 2 we present the basic notation used throughout the paper, and formally describe the framework we consider for generic-group delay functions. In Sect. 3 we prove our main impossibility result for generic delay functions, and in Sect. 4 we extend it to generic multilinear groups. Finally, in Sect. 5 we discuss several additional extensions, and in Appendix A we show that our attacker is efficient not only with respect to its number of parallel processors and generic group operations, but also in its additional internal computation.

2 Preliminaries

In this section we present the basic notions and standard cryptographic tools that are used in this work. For a distribution X we denote by $x \leftarrow X$ the process of sampling a value x from the distribution X. Similarly, for a set $\mathcal{X}$ we denote by $x \leftarrow \mathcal{X}$ the process of sampling a value x from the uniform distribution over $\mathcal{X}$. For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, \ldots, n\}$. A function $\nu : \mathbb{N} \rightarrow \mathbb{R}^+$ is *negligible* if for any polynomial $p(\cdot)$ there exists an integer N such that for all $n > N$ it holds that $\nu(n) \leq 1/p(n)$.

2.1 Generic Groups and Algorithms

As discussed in Sect. 1.1, we prove our results within the generic-group model introduced by Maurer [Mau05]. We consider computations in cyclic groups of order N (all of which are isomorphic to $\mathbb{Z}_N$ with respect to addition modulo N), for a λ-bit integer N that is generated by a order generation algorithm $\mathsf{OrderGen}(1^\lambda)$, where $\lambda \in \mathbb{N}$ is the security parameter (and N may or may not be prime).

When considering such groups, each computation Maurer's model is associated with a table $\mathbf{B}$. Each entry of this table stores an element of $\mathbb{Z}_N$, and we denote by V_i the group element that is stored in the ith entry. Generic algorithms access this table via an oracle $\mathcal{O}$, providing black-box access to $\mathbf{B}$ as follows. A generic algorithm $\mathcal{A}$ that takes d group elements as input (along with an optional bit-string) does not receive an explicit representation of these group elements, but instead, has oracle access to the table $\mathbf{B}$, whose first d entries store the $\mathbb{Z}_N$ elements corresponding to the d group element in $\mathcal{A}$'s input. That is, if the input of an algorithm A is a tuple $(g_1, \ldots, g_d, x)$, where $g_1, \ldots, g_d$ are group elements and x is an arbitrary string, then from A's point of view the input is the tuple $(\widehat{g_1}, \ldots, \widehat{g_d}, x)$, where $\widehat{g_1}, \ldots, \widehat{g_d}$ are pointers to the group elements $g_1, \ldots, g_d$ (these group elements are stored in the table $\mathbf{B}$), and x is given explicitly. All generic algorithms in this paper will receive as their first input a generator of the group; we capture this fact by always assuming that the first entry of $\mathbf{B}$ is occupied by $1 \in \mathbb{Z}_N$, and we will sometimes forgo noting this explicitly. The oracle $\mathcal{O}$ allows for two types of queries:

- **Group operation queries:** On input $(i, j, +)$ for $i, j \in \mathbb{N}$, the oracle checks that the ith and jth entries of the table $\mathbf{B}$ are not empty, computes $V_i + V_j \bmod N$ and stores the result in the next available entry. If either the ith or the jth entries are empty, the oracle ignores the query.
- **Equality queries:** On input $(i, j, =)$ for $i, j \in \mathbb{N}$, the oracle checks that the ith and jth entries in $\mathbf{B}$ are not empty, and then returns 1 if $V_i = V_j$ and 0 otherwise. If either the ith or the jth entries are empty, the oracle ignores the query.

In this paper we consider interactive computations in which multiple algorithms pass group elements (as well as non-group elements) as inputs to one another. This is naturally supported by the model as follows: When a generic algorithm $\mathcal{A}$ outputs k group elements (along with a potential bit-string σ), it outputs the indices of k (non-empty) entries in the table $\mathbf{B}$ (together with σ). When these outputs (or some of them) are passed on as inputs to a generic algorithm $\mathcal{C}$, the table $\mathbf{B}$ is re-initialized, and these values (and possibly additional group elements that $\mathcal{C}$ receives as input) are placed in the first entries of the table. Additionally, we rely on the following conventions:

1. Throughout the paper we refer to values as either "explicit" ones or "inexplicit" ones. Explicit values are all values whose representation (e.g., binary strings of a certain length) is explicitly provided to the generic algorithms under consideration. Inexplicit values are all values that correspond to group elements and that are stored in the table $\mathbf{B}$ – thus generic algorithms can

access them only via oracle queries. We will sometimes interchange between providing group elements as input to generic algorithms inexplicitly, and providing them explicitly. Note that moving from the former to the latter is well defined, since a generic algorithm $\mathcal{A}$ that receives some of its input group elements explicitly can always simulate the computation as if they were received as part of the table $\mathbf{B}$.
2. For a group element g, we will differentiate between the case where g is provided explicitly and the case where it is provided implicitly via the table $\mathbf{B}$, using the notation g in the former case, and the notation $\widehat{g}$ in the latter.
3. As is common in the generic group model, we identify group elements that are given as input to a generic algorithm with formal variables, the results of addition queries (i.e., the content of the entries in the table $\mathbf{B}$) with linear polynomials in these variables, and positively-answered equality queries between distinct polynomials with linear equations.

2.2 Generic-Group Delay Functions

A generic-group delay function is a triplet $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Sample}, \mathsf{Eval})$ of oracle-aided algorithms satisfying the following properties:

- Setup is a randomized algorithm that has oracle access to the group oracle $\mathcal{O}$, receives as input the group order $N \in \mathbb{N}$ and a sequentiality parameter $T \in \mathbb{N}$, and outputs public parameters $\mathsf{pp} = (\mathsf{pp}_G, \mathsf{pp}_s)$ where pp_G is an ordered list of group elements and $\mathsf{pp}_s \in \{0,1\}^*$ is an explicit string.
- Sample is a randomized algorithm that has oracle access to the group oracle $\mathcal{O}$, receives as input N and T as above, as well as the public parameters pp, and outputs $x = (x_G, x_s) \in \mathcal{X}_{\mathsf{pp}}$ (the domain $\mathcal{X}_{\mathsf{pp}}$ may be a function of the public parameters pp), where x_G is an ordered list of group elements and $x_s \in \{0,1\}^*$ is an explicit string.
- Eval is a deterministic algorithm that has oracle access to the group oracle $\mathcal{O}$, receives as input N,T and pp as above, as well as an input $x \in \mathcal{X}_{\mathsf{pp}}$, and outputs $y = (y_G, y_s)$, where y_G is an ordered list of group elements and $y_s \in \{0,1\}^*$ in an explicit string.

Motivated by notions of time-lock puzzles and verifiable delay functions, we consider delay functions where the lengths of the public parameters, inputs, and outputs are polynomial in λ and $\log T$. For time-lock puzzles this follows from the requirement that an input-output pair can be generated in time polynomial in λ and $\log T$, and for verifiable delay functions this follows from the requirement that the verification algorithm runs in time polynomial in λ and $\log T$.

In terms of security, we require that for a delay parameter T, no algorithm should be successful with a non-negligible probability in evaluating a delay function on a randomly-chosen input – even with any polynomial number of parallel processors and with a preprocessing stage.

Definition 2.1 (Sequentiality). *Let $T = T(\lambda)$ and $p = p(\lambda)$ be functions of the security parameter $\lambda \in \mathbb{N}$. A delay function $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Sample}, \mathsf{Eval})$ is (T,p)-sequential if for every polynomial $q = q(\cdot,\cdot)$ and for every pair of*

oracle-aided algorithms $(\mathcal{A}_0, \mathcal{A}_1)$*, where* $\mathcal{A}_0$ *issues at most* $q(\lambda, T)$ *oracle queries, and* $\mathcal{A}_1$ *consists of at most* $p(\lambda)$ *parallel processors, each of which issues at most* T *oracle queries, there exists a negligible function* $\nu(\cdot)$ *such that*

$$\Pr\left[y' = y \;\middle|\; \begin{array}{c} N \leftarrow \mathsf{OrderGen}(1^\lambda), \mathsf{pp} \leftarrow \mathsf{Setup}^{\mathcal{O}}(N, T) \\ \mathsf{st} \leftarrow \mathcal{A}_0^{\mathcal{O}}(N, T, \mathsf{pp}), x \leftarrow \mathsf{Sample}^{\mathcal{O}}(N, T, \mathsf{pp}) \\ y \leftarrow \mathsf{Eval}^{\mathcal{O}}(N, T, \mathsf{pp}, x) \\ y' \leftarrow \mathcal{A}_1^{\mathcal{O}}(\mathsf{st}, N, T, \mathsf{pp}, x) \end{array}\right] \leq \nu(\lambda)$$

for all sufficiently large $\lambda \in \mathbb{N}$*.*

3 Our Impossibility Result

In this section we prove our impossibility result for generic-group delay functions in cyclic groups of known orders. For ease of presentation, here we consider functions whose public parameters, inputs and outputs consist of group elements and do not additionally contain any explicit bit-strings (see Sect. 5 for extending our approach to this case).

In what follows we denote by $\mathsf{factors}(N)$ the multi-set of prime factors of the λ-bit group order N (where the number of appearances of each prime factor is its multiplicity – e.g., $\mathsf{factors}(100) = \{2, 2, 5, 5\}$). We prove the following theorem:

Theorem 3.1. *Let* $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Sample}, \mathsf{Eval})$ *be a generic-group delay function whose public parameters, inputs and outputs consist of* $k_{\mathsf{pp}}(\lambda, T)$*,* $k_{\mathsf{in}}(\lambda, T)$ *and* $k_{\mathsf{out}}(\lambda, T)$ *group elements, respectively, where* $\lambda \in \mathbb{N}$ *is the security parameter and* $T \in \mathbb{N}$ *is the delay parameter. Let* $Q_{\mathsf{eqEval}}(\lambda, T)$ *denote the number of equality queries issued by the algorithm* Eval*. Then, there exists a generic-group algorithm* $\mathcal{A}$ *that consists of* $(k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot \max\{k_{\mathsf{out}}, Q_{\mathsf{eqEval}}\}$ *parallel processors, each of which issues at most* $O((k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| \cdot \lambda)$ *sequential oracle queries, such that*

$$\Pr\left[\mathbf{y}' = \mathbf{y} \;\middle|\; \begin{array}{c} N \leftarrow \mathsf{OrderGen}(1^\lambda), \widehat{\mathsf{pp}} \leftarrow \mathsf{Setup}^{\mathcal{O}}(N, T) \\ \widehat{\mathbf{x}} \leftarrow \mathsf{Sample}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}) \\ \widehat{\mathbf{y}} \leftarrow \mathsf{Eval}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}}) \\ \widehat{\mathbf{y}'} \leftarrow \mathcal{A}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}}) \end{array}\right] = 1$$

for all $\lambda \in \mathbb{N}$ *and* $T \in \mathbb{N}$*, where the probability is taken over the internal randomness of* $\mathsf{OrderGen}$*,* Setup *and* Sample*.*

The proof of Theorem 3.1 relies on the following notation. We will at times substitute the group elements $\widehat{\mathsf{pp}} = (\widehat{\mathsf{pp}_1}, \ldots, \widehat{\mathsf{pp}_{k_{\mathsf{pp}}}})$ and $\widehat{\mathbf{x}} = (\widehat{x_1}, \ldots, \widehat{x_k})$ that are given as input to Eval, with formal variables $\mathsf{PP} = (\mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}})$ and $\mathbf{X} = (X_1, \ldots, X_{k_{\mathsf{in}}})$. When this is the case, instead of writing $\mathsf{Eval}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$ we will write $\mathsf{Eval}^{\mathbb{Z}_N[\mathsf{PP}, \mathbf{X}] | \mathcal{L}}(N, T, \mathsf{PP}, \mathbf{X})$, where $\mathcal{L}$ is a set of linear equations in PP and in $\mathbf{X}$. This latter computation is obtained from the original one by the following emulation:

- Group elements are represented via polynomials in the formal variables PP and $\mathbf{X}$. The computation keeps track of the elements via a local table, which replaces the table $\mathbf{B}$ of the oracle $\mathcal{O}$ (recall Sect. 2). This table isinitialized

so that its first $1 + k_{\mathsf{pp}} + k_{\mathsf{in}}$ entries are inhabited with the monomials $1, \mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}}, X_1, \ldots, X_{k_{\mathsf{in}}}$.

- Group operations are simulated via polynomial addition; i.e., when Eval issues a group operation query with two elements that are represented in the local table by two polynomials $p_1(\mathsf{PP}, \mathbf{X})$ and $p_2(\mathsf{PP}, \mathbf{X})$, the result is the polynomial $p_1(\mathsf{PP}, \mathbf{X}) + p_2(\mathsf{PP}, \mathbf{X})$, which is then placed in the next vacant entry of the table.
- Each equality query is answered affirmatively if and only if equality follows from the equations in $\mathcal{L}$ (in particular, when $\mathcal{L} = \emptyset$, equality queries are answered affirmatively if and only if the two polynomials at hand are identical).
- The output $\mathbf{y}(\mathsf{PP}, \mathbf{X}) = (y_1(\mathsf{PP}, \mathbf{X}), \ldots, y_{k_{\mathsf{out}}}(\mathsf{PP}, \mathbf{X}))$ of this computation is a vector of polynomials in PP and in $\mathbf{X}$. We denote by $\mathbf{y}(\mathsf{pp}, \mathbf{x}) = (y_1(\mathsf{pp}, \mathbf{x}), \ldots, y_{k_{\mathsf{out}}}(\mathsf{pp}, \mathbf{x}))$ the vector obtained by evaluating each entry of $\mathbf{y}(\mathsf{PP}, \mathbf{X})$ at the point $(\mathsf{pp}, \mathbf{x}) \in \mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}$.

We now turn to present the proof of Theorem 3.1.

Proof. Let $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Sample}, \mathsf{Eval})$ be a generic-group delay function, and consider the following adversary $\mathcal{A}$:

The adversary $\mathcal{A}$

The adversary $\mathcal{A}$ on input $(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$ and oracle access to $\mathcal{O}$ is defined as follows:

1. Initialize a set $\mathcal{L} = \emptyset$ of linear equations in the formal variables $\mathsf{PP} = (\mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}})$ and $\mathbf{X} = (X_1, \ldots, X_{k_{\mathsf{in}}})$.
2. Repeat the following steps for $t = (k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| + 1$ iterations:
   (a) Compute $\mathbf{y}'(\mathsf{PP}, \mathbf{X}) = \mathsf{Eval}^{\mathbb{Z}_N[\mathsf{PP}, \mathbf{X}]|\mathcal{L}}(N, T, \mathsf{PP}, \mathbf{X})$. Let m denote the number of equality queries that are negatively answered in the computation, and let $\ell_1(\mathsf{PP}, \mathbf{X}), \ldots, \ell_m(\mathsf{PP}, \mathbf{X})$ be the linear equations that would have followed from each of these queries had it been affirmatively answered.
   (b) For each $i \in [m]$, if $\ell_i(\mathsf{pp}, \mathbf{x})$ holds then add $\ell_i(\mathsf{PP}, \mathbf{X})$ to $\mathcal{L}$. If at least one linear equation was added to $\mathcal{L}$ then skip step 2(c) and continue to the next iteration.
   (c) Compute and output $\widehat{\mathbf{y}'(\mathsf{pp}, \mathbf{x})}$, then terminate.
3. Output $\perp$.

Query completed. Steps 1 and 2(a) require no oracle queries. Step 2(b) requires $m \cdot (k_{\mathsf{pp}} + k_{\mathsf{in}})$ parallel processors, each issuing $O(\log N)$ sequential queries for checking whether $\ell_i(\mathsf{pp}, \mathbf{x})$ hold for any $i \in [m]$ (and it holds that $m \leq Q_{\mathsf{eqEval}}$). Step 2(c) is executed at most once and requires $k_{\mathsf{out}} \cdot (k_{\mathsf{pp}} + k_{\mathsf{in}})$ parallel processors, each issuing $O(\log N)$ queries.

Finally, note that for a composite order N, the attacker $\mathcal{A}$ is not required to compute the factorization of N in order to determine the number of iterations. Specifically, for a λ-bit modulos N it always holds that $|\mathsf{factors}(N)| < \lambda$, and $\mathcal{A}$ can use this upper bound for determining an upper bound on the number of iterations.

Fix an iteration $j \in [t]$ where $t = (k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| + 1$, let $\mathcal{L}_j$ denote the state of the set $\mathcal{L}$ of linear equations at the beginning of the jth iteration, and consider the two computations $\mathbf{y} = \mathsf{Eval}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$ and $\mathbf{y}'_j(\mathsf{PP}, \mathbf{X}) = \mathsf{Eval}^{\mathbb{Z}_N[\mathsf{PP},\mathbf{X}]|\mathcal{L}_j}(N, T, \mathsf{PP}, \mathbf{X})$. By the condition specified in step 2(b) for adding a linear equation ℓ to $\mathcal{L}$, any $\ell \in \mathcal{L}_j$ is satisfied by $(\mathsf{pp}, \mathbf{x})$ (i.e., $\ell(\mathsf{pp}, \mathbf{x})$ holds). Therefore, every equality query that is negatively answered in the computation of $\mathbf{y}$ is also negatively answered in the computation of $\mathbf{y}'_j(\mathsf{PP}, \mathbf{X})$. Hence, one of the following two cases must happen:

- Case I: All equality queries in both computations are answered the same way. In this case, the output of both computations is the same vector of linear polynomials in terms of the inputs, and it holds that $\mathbf{y} = \mathbf{y}'_j(\mathsf{pp}, \mathbf{x})$. Furthermore, since all negatively answered queries in the computation of $\mathbf{y}'_j(\mathsf{PP}, \mathbf{X})$ are also negatively answered in the computation of $\mathbf{y}$, then for all $i \in [m]$ the linear equation $\ell_i(\mathsf{pp}, \mathbf{x})$ is not satisfied. Therefore, step 2(c) is reached in this case and $\mathcal{A}$ succeeds in outputting $\mathbf{y}$.
- Case II: There exists an equality query that is positively answered in the computation of $\mathbf{y}$ but is negatively answered in the computation of $\mathbf{y}'_j(\mathsf{PP}, \mathbf{X})$. This means that there exists an $i \in [m]$ for which $\ell_i(\mathsf{pp}, \mathbf{x})$ holds, but $\ell_i(\mathsf{PP}, \mathbf{X})$ is not implied by the linear equations in $\mathcal{L}_j$. Thus, ℓ_i is added to $\mathcal{L}$ and the algorithm skips to the next iteration.

So far we have shown that $\mathcal{A}$ outputs $\mathbf{y}$ (i.e., the correct output) whenever step 2(c) is reached. We now complete the proof by showing that step 3 is never reached (i.e., that step 2(c) is always reached). Suppose towards contradiction that $t = (k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| + 1$ iterations are performed, but none of them reaches step 2(c). For every $j \in [t]$ recall that $\mathcal{L}_j$ denotes the state of the set $\mathcal{L}$ at the beginning of the jth iteration, and denote by $\mathcal{L}_{t+1}$ the state of $\mathcal{L}$ when reaching step 3. Then, it holds that $\mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \cdots \subsetneq \mathcal{L}_{t+1}$, since for every $j \in [t]$ the set $\mathcal{L}_{j+1}$ contains at least one linear equation that is not implied by $\mathcal{L}_j$. Also, as already mentioned, for every $j \in [t+1]$ and $\ell \in \mathcal{L}_j$ the linear equation $\ell(\mathsf{pp}, \mathbf{x})$ is satisfied. For a system of linear equations $\mathcal{M}$ with k variables over $\mathbb{Z}_N$, if there exists a solution $\mathbf{z} \in \mathbb{Z}_N^k$ to the system $\mathcal{M}$ then the set of solutions forms a coset of a subgroup of $\mathbb{Z}_N^k$. That is, there exists a subgroup H of $\mathbb{Z}_N^k$ such the the set of solutions to $\mathcal{M}$ is $\mathbf{z} + H$. Therefore, there exist subgroups $H_1, \ldots, H_{t+1}$ of $\mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}$ such that for every $j \in [t+1]$ it holds that

$$\left\{(\mathsf{pp}', \mathbf{x}') \in \mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}} \middle| \forall \ell(\mathsf{PP}, \mathbf{X}) \in \mathcal{L}_j \ : \ \ell(\mathsf{pp}', \mathbf{x}') \text{ is satisfied}\right\} = (\mathsf{pp}, \mathbf{x}) + H_j \ .$$

Then, it holds that $H_1 > H_2 > \cdots > H_{t+1}$ (i.e., H_{j+1} is a proper subgroup of H_j for every $j \in [t]$). Therefore, the order of every H_{j+1} divides that of H_j, and it holds that

$$\mathsf{factors}(|H_{t+1}|) \subsetneq \mathsf{factors}(|H_t|) \subsetneq \cdots \subsetneq \mathsf{factors}(|H_1|) \subseteq \mathsf{factors}(|\mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}|).$$

Since

$$|\mathsf{factors}(|\mathbb{Z}_N^{k_{\mathsf{pp}}+k_{\mathsf{in}}}|)| = |\mathsf{factors}(N^{k_{\mathsf{pp}}+k_{\mathsf{in}}})| = t - 1,$$

it is impossible to have t proper containments in the above chain and we reach a contradiction. ■

4 Extending Our Impossibility Result to the Multilinear Setting

In this section we extend our impossibility result to groups that are equipped with a d-linear map. Similarly to our proof in Sect. 3, once again we begin by considering functions whose public parameters, inputs and outputs consist of group elements and do not additionally contain any explicit bit-strings (see Sect. 5 for extending our proof to this case).

Recall that we denote by $\mathsf{factors}(N)$ the multi-set of prime factors of the λ-bit group order N (where the number of appearances of each prime factor is its multiplicity – e.g., $\mathsf{factors}(100) = \{2, 2, 5, 5\}$). We prove the following theorem (from which Theorem 3.1 follows by setting $d = 1$):

Theorem 4.1. *Let $d = d(\lambda)$ be a function of the security parameter $\lambda \in \mathbb{N}$, and let $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Sample}, \mathsf{Eval})$ be a generic d-linear-group delay function whose public parameters, inputs and outputs consist of $k_{\mathsf{pp}}(\lambda, T)$, $k_{\mathsf{in}}(\lambda, T)$ and $k_{\mathsf{out}}(\lambda, T)$ group elements, respectively, where $T \in \mathbb{N}$ is the delay parameter. Let $Q_{\mathsf{eqEval}}(\lambda, T)$ denote the number of equality queries issued by the algorithm Eval. Then, there exists a generic-group algorithm $\mathcal{A}$ that consists of $\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d} \cdot \max\{k_{\mathsf{out}}, Q_{\mathsf{eqEval}}\}$ parallel processors, each of which issues at most $O(\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d} \cdot |\mathsf{factors}(N)| \cdot \lambda)$ sequential oracle queries, such that*

$$\Pr\left[\mathbf{y}' = \mathbf{y} \;\middle|\; \begin{array}{c} N \leftarrow \mathsf{OrderGen}(1^\lambda), \widehat{\mathsf{pp}} \leftarrow \mathsf{Setup}^{\mathcal{O}}(N, T) \\ \widehat{\mathbf{x}} \leftarrow \mathsf{Sample}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}) \\ \widehat{\mathbf{y}} \leftarrow \mathsf{Eval}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}}) \\ \widehat{\mathbf{y}'} \leftarrow \mathcal{A}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}}) \end{array}\right] = 1$$

for all $\lambda \in \mathbb{N}$ and $T \in \mathbb{N}$, where the probability is taken over the internal randomness of $\mathsf{OrderGen}$, Setup and Sample. Moreover, $\mathcal{A}$ issues at most $O\left(\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d}\right)$ multilinear map queries, which may all be issued in parallel.

Theorem 4.1 is in fact identical to Theorem 3.1 expect for replacing the term $k_{\mathsf{pp}} + k_{\mathsf{in}}$ with the term $\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d}$, where d is the level of linearity, and note that $\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d} \leq (k_{\mathsf{pp}} + k_{\mathsf{in}} + 1)^d$ (i.e., the efficiency of our attacker degrades exponentially with the level of linearity). This shows that there are no constructions of generic-group delay functions in cyclic groups of known orders that are equipped with a d-linear map, for any d such that $\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d}$ is polynomial in the security parameter $\lambda \in \mathbb{N}$. For example, this holds for any constant d, and for functions whose public parameters and inputs consist of a constant number of group elements this holds for any $d = O(\log \lambda)$.

In what follows we first naturally extend the framework of generic groups and algorithms, described in Sect. 2.1, to the multilinear setting (see Sect. 4.1), and then prove Theorem 4.1 (see Sect. 4.2).

4.1 Generic Multilinear Groups

In order to generalize our impossibility result to rule out generic constructions in groups that are equipped with a multilinear map, we first extend the model of Maurer [Mau05] (recall Sect. 2.1) to support such groups. For simplicity of presentation, we start by defining the model and proving our impossibility result assuming that the multilinear map is symmetric. Then, in Sect. 5 we discuss how to naturally extend the model and the proof to accommodate asymmetric maps as well.

Let $d = d(\lambda)$ be a function of the security parameter $\lambda \in \mathbb{N}$. In what follows, we consider computations in a source group of order N with a d-linear map into a target group of the same order, for a λ-bit integer N generated by the order generation algorithm $\mathsf{OrderGen}(1^\lambda)$. For the purpose of capturing generic computations in such groups, we consider a model which is obtained from Maurer's model by the following modifications:

1. Each element in the table $\mathbf{B}$ is now a pair in $\{\mathsf{source}, \mathsf{target}\} \times \mathbb{Z}_N$; meaning, it consists of a label which specifies whether this element is from the source group or from the target group, together with a $\mathbb{Z}_N$ element as before. All generic algorithms we consider now receive as input a generator for the source group; we capture this fact by always initializing $\mathbf{B}$ with the element $(\mathsf{source}, 1)$ in its first entry (we will forgo noting this explicitly).[8]
2. When the oracle receives a group operation query of the form $(i, j, +)$, it first verifies that the label of the element in the ith entry of the table $\mathbf{B}$ is the same as the label of the element in the jth entry (and that both entries are non-empty). If that is the case, then the oracle places $(\mathsf{label}, V_i + V_j)$ in the next vacant entry of the table, where label is the label of the elements at hand, and V_i and V_j are the $\mathbb{Z}_N$ elements in the ith entry and in the jth entry of $\mathbf{B}$, respectively.
3. When the oracle receives an equality query of the form $(i, j, =)$, it first verifies that the label of the element in the ith entry of the table $\mathbf{B}$ is the same as the label of the element in the jth entry (and that both entries are non-empty). If that is the case, then the oracle returns 1 if $V_i = V_j$ and 0 otherwise.
4. We add a third type of queries, which we refer to as *multilinear map queries*: On input $(i_1, \ldots, i_d, \times)$, the oracle first verifies that for each $j \in [d]$ the i_jth entry contains the label source. If so, it places $(\mathsf{target}, \prod_{j \in [d]} V_{i_j})$, where for every $j \in [d]$, V_{i_j} is the $\mathbb{Z}_N$ element in the i_jth entry of $\mathbf{B}$ and the multiplication is with respect to addition modulo N.

[8] The generator $(\mathsf{target}, 1)$ for the target group can be obtained using a single multilinear map query, as described below.

The definition of generic-group delay functions remains the same as in Sect. 2.2, other than the fact that all algorithms (i.e., Setup, Sample and Eval, as well as the adversarial algorithms $\mathcal{A}_0$ and $\mathcal{A}_1$ from Definition 2.1) get oracle access to the extended oracle described in this section, and two additional inputs: (1) The arity d of the map; and (2) the labels of the group elements that are placed in the table $\mathbf{B}$ when the algorithm starts its execution.

4.2 Proof of Theorem 4.1

We define the computation $\mathsf{Eval}^{\mathbb{Z}_N[\mathsf{Lin}_d(\mathsf{PP},\mathbf{X})]|\mathcal{L}}(N,T,\mathsf{PP},\mathbf{X})$ to be obtained from the original computation $\mathsf{Eval}^{\mathcal{O}}(N,T,\widehat{\mathsf{pp}},\widehat{\mathbf{x}})$ by a similar emulation to that from Sect. 3, with the following differences:

- The tuples PP and $\mathbf{X}$ consist of pairs of a label and a variable $\mathsf{PP} = ((\mathsf{grp}_1^{\mathsf{pp}},\mathsf{PP}_1),\ldots,(\mathsf{grp}_{k_{\mathsf{pp}}}^{\mathsf{pp}},\mathsf{PP}_{k_{\mathsf{pp}}}))$ and $\mathbf{X} = ((\mathsf{grp}_1^{\mathbf{x}},X_1),\ldots,(\mathsf{grp}_{k_{\mathsf{in}}}^{\mathbf{x}},X_{k_{\mathsf{in}}}))$, where each label is either source or target, and is determined according to the corresponding label of the original input $(\mathsf{pp},\mathbf{x})$.[9] We assume without loss of generality that the source variables in both PP and $\mathbf{X}$ appear before the target variables, denote the number of source variables in these tuples by $k_{\mathsf{pp}}^{\mathsf{src}}$ and $k_{\mathsf{in}}^{\mathsf{src}}$, respectively, and denote their total number by $k^{\mathsf{src}} = k_{\mathsf{pp}}^{\mathsf{src}} + k_{\mathsf{in}}^{\mathsf{src}}$.
- We define new variables

$$\begin{aligned}\mathbf{Z} = \mathsf{Lin}_d(\mathsf{PP},\mathbf{X}) = \ & \{Z_{\alpha_1,\ldots,\alpha_{k^{\mathsf{src}}}} | \alpha_1+\cdots+\alpha_{k^{\mathsf{src}}} \le d\} \\ & \cup \left\{\mathsf{PP}_1,\ldots,\mathsf{PP}_{k_{\mathsf{pp}}},X_1,\ldots,X_{k_{\mathsf{in}}}\right\},\end{aligned}$$

  where each variable of the form $Z_{\alpha_1,\ldots,\alpha_{k^{\mathsf{src}}}}$ is associated with the product $\mathsf{PP}_1^{\alpha_1}\cdots\mathsf{PP}_{k_{\mathsf{pp}}^{\mathsf{src}}}^{\alpha_{k_{\mathsf{pp}}^{\mathsf{src}}}} \cdot X_1^{\alpha_{k_{\mathsf{pp}}^{\mathsf{src}}+1}}\cdots X_{k_{\mathsf{in}}^{\mathsf{src}}}^{\alpha_{k^{\mathsf{src}}}}$. Additionally, for the standard basis $e_1,\ldots,e_{k^{\mathsf{src}}}$ we identify the variables $Z_{e_1},\ldots,Z_{e_{k^{\mathsf{src}}}}$ with the source variables $\mathsf{PP}_1,\ldots,\mathsf{PP}_{k_{\mathsf{pp}}^{\mathsf{src}}},X_1,\ldots,X_{k_{\mathsf{in}}^{\mathsf{src}}}$, respectively (thus, the union in the above definition of $\mathbf{Z}$ is not disjoint). The number of variables in $\mathbf{Z}$ is at most $g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})$ where $g_d(k) = \binom{k+d}{d}$ (the number of non-negative integer solutions to $\alpha_1+\cdots+\alpha_k \le d$).
- Each entry in the local table maintained by the computation (recall Sect. 3) includes a label – either source or target – in addition to a formal polynomial as before. The table is initialized so that its first $1 + k_{\mathsf{pp}} + k_{\mathsf{in}}$ entries are inhabited with the pairs $(\mathsf{source},1),(\mathsf{grp}_1^{\mathsf{pp}},\mathsf{PP}_1),\ldots,(\mathsf{grp}_{k_{\mathsf{pp}}}^{\mathsf{pp}},\mathsf{PP}_{k_{\mathsf{pp}}}),(\mathsf{grp}_1^{\mathbf{x}},X_1),\ldots,(\mathsf{grp}_{k_{\mathsf{in}}}^{\mathbf{x}},X_{k_{\mathsf{in}}})$. These labels are used in accordance with the oracle definition from Sect. 4.1: When group operation or equality queries are issued, the computation first makes the necessary label consistency checks; and when a group operation query is executed, the result polynomial is stored in the local table with the appropriate label.

[9] Typically, the labels are predetermined by the scheme, but if this is not the case then the labels can be recovered from the input.

– Multilinear map queries are simulated as follows. First, we check that all d polynomials that are the input to the query are stored in the local table with the label source (otherwise, the query is ignored). If so, then let $p_1(\mathbf{Z}), \ldots, p_d(\mathbf{Z})$ be the polynomials given as input to the query. By the queries allowed, it is guaranteed that $p_1, \ldots, p_d$ are linear polynomials which only involve the variables $\mathsf{PP}^{\mathsf{src}} = (\mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}^{\mathsf{src}}})$ and $\mathbf{X}^{\mathsf{src}} = (X_1, \ldots, X_{k_{\mathsf{in}}^{\mathsf{src}}})$. We compute the polynomial $p(\mathsf{PP}^{\mathsf{src}}, \mathbf{X}^{\mathsf{src}}) = \prod_{i \in [d]} p_i(\mathsf{PP}^{\mathsf{src}}, \mathbf{X}^{\mathsf{src}})$, and then we replace each product of variables $\mathsf{PP}_1^{\alpha_1} \cdots \mathsf{PP}_{k_{\mathsf{pp}}^{\mathsf{src}}}^{\alpha_{k_{\mathsf{pp}}^{\mathsf{src}}}} \cdot X_1^{\alpha_{k_{\mathsf{pp}}^{\mathsf{src}}+1}} \cdots X_{k_{\mathsf{in}}^{\mathsf{src}}}^{\alpha_{k_{\mathsf{in}}^{\mathsf{src}}}}$ with the single variable $Z_{\alpha_1, \ldots, \alpha_{k^{\mathsf{src}}}}$ to receive a linear polynomial $p'(\mathbf{Z})$. Finally, we store $(\mathsf{target}, p'(\mathbf{Z}))$ in the next vacant entry of the local table.
– Valid equality queries (i.e., when the entries to be compared have the same label) are answered as in Theorem 4.1. If $p_1(\mathbf{Z})$ and $p_2(\mathbf{Z})$ are to be compared, then the query is answered affirmatively if and only if the equality $p_1(\mathbf{Z}) = p_2(\mathbf{Z})$ follows from the equations in $\mathcal{L}$ (which are linear in $\mathbf{Z}$).
– For $\mathsf{pp} \in \mathbb{Z}_N^{k_{\mathsf{pp}}}$ and $\mathbf{x} \in \mathbb{Z}_N^{k_{\mathsf{in}}}$ we define

$$\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x}) = \left\{ \mathsf{pp}_1^{\alpha_1} \cdots \mathsf{pp}_{k_{\mathsf{pp}}^{\mathsf{src}}}^{\alpha_{k_{\mathsf{pp}}^{\mathsf{src}}}} \cdot x_1^{\alpha_{k_{\mathsf{pp}}^{\mathsf{src}}+1}} \cdots x_{k_{\mathsf{in}}^{\mathsf{src}}}^{\alpha_{k_{\mathsf{in}}^{\mathsf{src}}}} \middle| \alpha_1 + \cdots + \alpha_{k^{\mathsf{src}}} \leq d \right\} \cup \left\{ \mathsf{pp}_1, \ldots, \mathsf{pp}_{k_{\mathsf{pp}}}, x_1, \ldots, x_{k_{\mathsf{in}}} \right\} .$$

That is, $\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})$ contains all elements of $(\mathsf{pp}, \mathbf{x})$ and all products of at most d elements from the source variables of $(\mathsf{pp}, \mathbf{x})$. Given pointers $(\widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$, we can compute $\widehat{\mathbf{w}} = \{\widehat{(\mathsf{target}, z)} | z \in \mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})\}$ by using multilinear map queries. Then, given a linear polynomial $p(\mathbf{Z})$, we can compute $\widehat{(\mathsf{target}, p(\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})))}$ using $\mathbf{w}$, and if $p(\mathbf{Z})$ involves only the source variables $\mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}^{\mathsf{src}}}, X_1, \ldots, X_{k_{\mathsf{in}}^{\mathsf{src}}}$ then we can compute $\widehat{(\mathsf{source}, p(\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})))}$.
– The output of the computation is of the form $\mathbf{y}'(\mathbf{Z}) = ((\mathsf{grp}_1, y_1'(\mathbf{Z})), \ldots, (\mathsf{grp}_{k_{\mathsf{out}}}, y_{k_{\mathsf{out}}}'(\mathbf{Z})))$ where $\mathsf{grp}_i \in \{\mathsf{source}, \mathsf{target}\}$ and $y_i'(\mathbf{Z})$ is a linear polynomial for every $i \in [k_{\mathsf{out}}]$. Moreover, if $\mathsf{grp}_i = \mathsf{source}$ then $y_i'(\mathbf{Z})$ is guaranteed to involve only the source variables $\mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}^{\mathsf{src}}}, X_1, \ldots, X_{k_{\mathsf{in}}^{\mathsf{src}}}$. Therefore, given pointers $(\widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$, for every $i \in [k_{\mathsf{out}}]$ we can compute $\widehat{(\mathsf{grp}_i, y_i'(\mathbf{z}))}$ where $\mathbf{z} = \mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})$, and we denote

$$\widehat{\mathbf{y}'(\mathbf{z})} = (\widehat{(\mathsf{grp}_1, y_1'(\mathbf{z}))}, \ldots, \widehat{(\mathsf{grp}_{k_{\mathsf{out}}}, y_{k_{\mathsf{out}}}'(\mathbf{z}))}) .$$

We now turn to present the proof of Theorem 4.1.

Proof. Let $\mathsf{DF} = (\mathsf{Setup}, \mathsf{Sample}, \mathsf{Eval})$ be a generic d-linear-group delay function, and consider the following adversary $\mathcal{A}$:

The adversary $\mathcal{A}$

The adversary $\mathcal{A}$ on input $(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$ and oracle access to $\mathcal{O}$ is defined as follows:

1. Initialize a set $\mathcal{L} = \emptyset$ of linear equations in the formal variables $\mathsf{Lin}_d(\mathsf{PP}, \mathbf{X}) = \mathbf{Z}$, where $\mathsf{PP} = (\mathsf{PP}_1, \ldots, \mathsf{PP}_{k_{\mathsf{pp}}})$ and $\mathbf{X} = (X_1, \ldots, X_{k_{\mathsf{in}}})$.
2. Compute $\widehat{\mathbf{w}} = \{\widehat{(\mathsf{target}, z)} | z \in \mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})\}$.

3. Repeat the following steps for $t = g_d(k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| + 1$ iterations:
   (a) Compute $\mathbf{y}'(\mathbf{Z}) = \mathsf{Eval}^{\mathbb{Z}_N[\mathsf{Lin}_d(\mathsf{PP},\mathbf{X})]|\mathcal{L}}(N, T, \mathsf{PP}, \mathbf{X})$. Let m denote the number of equality queries that are negatively answered in the computation, and let $\ell_1(\mathbf{Z}), \ldots, \ell_m(\mathbf{Z})$ be the linear equations that would have followed from each of these queries had it been affirmatively answered.
   (b) For each $i \in [m]$, if $\ell_i(\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x}))$ holds then add $\ell_i(\mathbf{Z})$ to $\mathcal{L}$. If at least one linear equation was added to $\mathcal{L}$ then skip step 3(c) and continue to the next iteration.
   (c) Compute and output $\widehat{\mathbf{y}'(\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x}))}$, then terminate.
4. Output $\perp$.

Query Complexity. Steps 1 and 3(a) require no oracle queries. Step 2 requires at most $g_d(k_{\mathsf{pp}} + k_{\mathsf{in}})$ parallel processors, each issuing a single multilinear map query. Step 3(b) requires $m \cdot g_d(k_{\mathsf{pp}} + k_{\mathsf{in}})$ parallel processors, each issuing $O(\log N)$ sequential queries for checking whether $\ell_i(\mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x}))$ hold (using the precomputed $\widehat{\mathbf{w}}$) for any $i \in [m]$ (and it holds that $m \leq Q_{\mathsf{eqEval}}$). Step 3(c) is executed at most once and requires $k_{\mathsf{out}} \cdot g_d(k_{\mathsf{pp}} + k_{\mathsf{in}})$ parallel processors, each issuing $O(\log N)$ queries (using the precomputed $\widehat{\mathbf{w}}$).

Finally, note that for a composite order N, the attacker $\mathcal{A}$ is not required to compute the factorization of N in order to determine the number of iterations. Specifically, for a λ-bit modulos N it always holds that $|\mathsf{factors}(N)| < \lambda$, and $\mathcal{A}$ can use this upper bound for determining an upper bound on the number of iterations.

Fix an iteration $j \in [t]$ where $t = g_d(k_{\mathsf{pp}} + k_{\mathsf{in}}) \cdot |\mathsf{factors}(N)| + 1$, let $\mathcal{L}_j$ denote the state of the set $\mathcal{L}$ of linear equations at the beginning of the jth iteration, and consider the two computations $\mathbf{y} = \mathsf{Eval}^{\mathcal{O}}(N, T, \widehat{\mathsf{pp}}, \widehat{\mathbf{x}})$ and $\mathbf{y}'_j(\mathbf{Z}) = \mathsf{Eval}^{\mathbb{Z}_N[\mathsf{Lin}_d(\mathsf{PP},\mathbf{X})]|\mathcal{L}_j}(N, T, \mathsf{PP}, \mathbf{X})$. By the condition specified in step 3(b) for adding a linear equation ℓ to $\mathcal{L}$, any $\ell \in \mathcal{L}_j$ is satisfied by $\mathbf{z} = \mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})$ (i.e., $\ell(\mathbf{z})$ holds). Therefore, every equality query that is negatively answered in the computation of $\mathbf{y}$ is also negatively answered in the computation of $\mathbf{y}'_j(\mathbf{Z})$. Hence, one of the following two cases must happen:

- Case I: All equality queries in both computations are answered the same way. In this case, the output of both computations is the same vector of linear polynomials in terms of $\mathbf{z} = \mathsf{Products}_{\leq d}(\mathsf{pp}, \mathbf{x})$ and $\mathbf{Z} = \mathsf{Lin}_d(\mathsf{PP}, \mathbf{X})$, respectively, and also each coordinate in the output has the same $\{\mathsf{source}, \mathsf{target}\}$ label, so it holds that $\mathbf{y} = \mathbf{y}'_j(\mathbf{z})$. Furthermore, since all negatively answered queries in the computation of $\mathbf{y}'_j(\mathbf{Z})$ are also negatively answered in the computation of $\mathbf{y}$, then for all $i \in [m]$ the linear equation $\ell_i(\mathbf{z})$ is not satisfied. Therefore, step 3(c) is reached in this case and $\mathcal{A}$ succeeds in outputting $\mathbf{y}$.
- Case II: There exists an equality query that is positively answered in the computation of $\mathbf{y}$ but is negatively answered in the computation of $\mathbf{y}'_j(\mathbf{Z})$. This means that there exists an $i \in [m]$ for which $\ell_i(\mathbf{z})$ holds, but $\ell_i(\mathbf{Z})$ is not implied by the linear equations in $\mathcal{L}_j$. Thus, ℓ_i is added to $\mathcal{L}$ and the algorithm skips to the next iteration.

So far we have shown that $\mathcal{A}$ outputs $\mathbf{y}$ (i.e., the correct output) whenever step 3(c) is reached. We now complete the proof by showing that step 4 is never reached (i.e., that step 3(c) is always reached). Suppose towards contradiction

that $t = g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})\cdot|\mathsf{factors}(N)|+1$ iterations are performed, but none of them reaches step 3(c). For every $j \in [t]$ recall that $\mathcal{L}_j$ denotes the state of the set $\mathcal{L}$ at the beginning of the jth iteration, and denote by $\mathcal{L}_{t+1}$ the state of $\mathcal{L}$ when reaching step 4. Then, it holds that $\mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \cdots \subsetneq \mathcal{L}_{t+1}$, since for every $j \in [t]$ the set $\mathcal{L}_{j+1}$ contains at least one linear equation that is not implied by $\mathcal{L}_j$. Also, as already mentioned, for every $j \in [t+1]$ and $\ell \in \mathcal{L}_j$ the linear equation $\ell(\mathbf{z})$ is satisfied. For a system of linear equations $\mathcal{M}$ with k variables over $\mathbb{Z}_N$, if there exists a solution $\mathbf{z} \in \mathbb{Z}_N^k$ to the system $\mathcal{M}$ then the set of solutions forms a coset of a subgroup of $\mathbb{Z}_N^k$. That is, there exists a subgroup H of $\mathbb{Z}_N^k$ such the the set of solutions to $\mathcal{M}$ is $\mathbf{z}+H$. Therefore, there exist subgroups $H_1,\ldots,H_{t+1}$ of $\mathbb{Z}_N^{g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})}$ such that for every $j \in [t+1]$ it holds that

$$\left\{\mathbf{z}' \in \mathbb{Z}_N^{g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})} \middle| \forall \ell(\mathbf{Z}) \in \mathcal{L}_j \ : \ \ell(\mathbf{z}') \text{ is satisfied}\right\} = \mathbf{z}+H_j \ .$$

Then, it holds that $H_1 > H_2 > \cdots > H_{t+1}$ (i.e., H_{j+1} is a proper subgroup of H_j for every $j \in [t]$). Therefore, the order of every H_{j+1} divides that of H_j, and it holds that

$$\mathsf{factors}(|H_{t+1}|) \subsetneq \mathsf{factors}(|H_t|) \subsetneq \cdots \subsetneq \mathsf{factors}(|H_1|) \subseteq \mathsf{factors}(|\mathbb{Z}_N^{g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})}|).$$

Since

$$|\mathsf{factors}(|\mathbb{Z}_N^{g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})}|)| = |\mathsf{factors}(N^{g_d(k_{\mathsf{pp}}+k_{\mathsf{in}})})| = t-1,$$

it is impossible to have t proper containments in the above chain and we reach a contradiction. ■

5 Additional Extensions

In this section we first discuss two extensions of our results, showing that our proofs extend to delay functions whose public parameters, inputs and outputs may include arbitrary bit-strings (in addition to group elements), and to asymmetric multilinear maps. Then, we pose an open problem regarding incremental computation of Smith normal forms.

Allowing explicit bit-strings as part of pp, x and y. Our proofs from Sects. 3 and 4 readily extend to the case where the public parameters pp, the input $\mathbf{x}$ and the output $\mathbf{y}$ may include arbitrary bit-strings, in addition to group elements. We review the necessary adjustments for our proof from Sect. 3, and note that essentially identical adjustments can be applied to our proof in the multilinear setting as well. Concretely:

- In addition to N, T, $\widehat{\mathsf{pp}}$ and $\widehat{\mathbf{x}}$, the evaluation algorithm Eval now receives as input two bit-strings, pp_s and x_s, denoting the bit-string parts of pp and of the input $\mathbf{x}$, respectively, and outputs a bit-string y_s in addition to $\widehat{\mathbf{y}}$. The computation $\mathsf{Eval}^{\mathbb{Z}_N[\mathsf{PP},\mathbf{X}]|\mathcal{L}}\left(N,T,\left(\mathsf{PP},\mathsf{pp}_s\right),\left(\mathbf{X},x_s\right)\right)$ is then defined via an emulation of the computation $\mathsf{Eval}^{\mathcal{O}}\left(N,T,\left(\widehat{\mathsf{pp}},\mathsf{pp}_s\right),\left(\widehat{\mathbf{x}},x_s\right)\right)$ similarly to Sect. 3: The local table maintained by the emulation and the way queries

are emulated are defined as in Sect. 3, and the output of this emulation is now a pair $(\mathbf{y}(\mathsf{PP},\mathbf{X}), y_s)$, where $\mathbf{y}(\mathsf{PP},\mathbf{X})$ is a vector of k_{out} polynomials $y_1(\mathsf{PP},\mathbf{X}),\ldots,y_{k_{\mathsf{out}}}(\mathsf{PP},\mathbf{X})$ in PP and in $\mathbf{X}$, and y_s is an explicit bit-string.

– The adversary $\mathcal{A}$ now receives the bit-strings pp_s and x_s, in addition to its inputs from Sect. 3. In Step 2(a) it now runs the emulation $\mathsf{Eval}^{\mathbb{Z}_N[\mathsf{PP},\mathbf{X}]|\mathcal{L}}(N, T, (\mathsf{PP},\mathsf{pp}_s), (\mathbf{X}, x_s))$ to obtain its output $(\mathbf{y}'(\mathsf{PP},\mathbf{X}), y'_s)$. In Step 2(c) it computes $\mathbf{y}'(\widehat{\mathsf{pp}},\mathbf{x})$ and outputs $\left(\mathbf{y}'(\widehat{\mathsf{pp}},\mathbf{x}), y'_s\right)$. The main additional observation is that for each iteration $j \in [(k_{\mathsf{pp}}+k_{\mathsf{in}})\cdot|\mathsf{factors}(N)|+1]$, if all equality queries in the emulation $\mathsf{Eval}^{\mathbb{Z}_N[\mathsf{PP},\mathbf{X}]|\mathcal{L}}(N, T, (\mathsf{PP},\mathsf{pp}_s), (\mathbf{X}, x_s))$ in that iteration are answered consistently with the equality pattern in $\mathsf{Eval}^{\mathcal{O}}(N, T, (\widehat{\mathsf{pp}},\mathsf{pp}_s), (\widehat{\mathbf{x}}, x_s))$, then the bit-string component y'_s outputted by the emulation in this iteration is the same as the bit-string component y_s outputted by the original computation $\mathsf{Eval}^{\mathcal{O}}(N, T, (\widehat{\mathsf{pp}},\mathsf{pp}_s), (\widehat{\mathbf{x}}, x_s))$. Hence, when Case I from our analysis is reached, it is still the case that the adversary is successful in outputting the correct output.

Asymmetric multilinear maps. Our impossibility result from Sect. 4 can be adjusted in order to rule out the existence of generic-group delay functions in groups with *asymmetric* multilinear maps; i.e., collections of $d+1$ groups – d source groups and a single target group, each of which is of order N – which are equipped with a d-linear operation mapping d elements, an element from each source group, into an element in the target group.

First, the model has to be extended to support such groups. This is done in a natural manner, by considering $d+1$ labels (instead of 2): $\mathsf{source}_1,\ldots,\mathsf{source}_d$ and target. Now, each entry in the table $\mathbf{B}$ is pair of the form (label, a), where label is one of the aforementioned labels, and $a \in \mathbb{Z}_N$; and we assume that the table $\mathbf{B}$ is always initialized with the pairs $(\mathsf{source}_1, 1),\ldots,(\mathsf{source}_d, 1)$ in its first d entries, respectively. Upon receiving a multilinear operation query, the oracle now verifies that the labels in the entries (implicitly) given as input to the oracle are indeed $\mathsf{source}_1,\ldots,\mathsf{source}_d$.

The proof is then obtained from the proof of Theorem 4.1 by adjusting it to this generalized generic model. Roughly speaking, the main adjustment is that now the linearization procedure needs to take into consideration the particular group of each input element. More concretely, the new formal variables introduced by this linearization (denoted by $\mathbf{Z}$ in the proof of Theorem 4.1) do not include all products of degree at most d of the formal variables replacing the source group elements in the public parameters and in the input. Instead, they include all products of at most d elements, with *distinct labels* from $\{\mathsf{source}_1,\ldots,\mathsf{source}_d\}$. Hence, the number of new formal variables introduced by the linearization phase is now at most $\left((k_{\mathsf{pp}}+k_{\mathsf{in}})/d+1\right)^d$, rather than $\binom{k_{\mathsf{pp}}+k_{\mathsf{in}}+d}{d}$.

Incremental computation of Smith normal forms. As discussed in Sect. 1.3 and described in detail in Appendix A, our attacker is efficient not only in its number of parallel processors and generic group operations but also in its additional internal computation. Specifically, in each iteration our attacker performs

a single invocation of any algorithm for computing Smith normal form. However, throughout the attack the matrices to which we apply such an algorithm are not independent of each other, but rather each matrix is obtained from the previous one by adding one more row and column. Thus, any algorithm that can compute Smith normal forms in an incremental manner may lead to substantial improvements in the practical running time of our attacker. Finally, we note that efficiently realizing our attacker's internal computation is not essential for our result in the generic-group model, and that basing our approach on fast algorithms for Smith normal forms is just one concrete possibility.

A Fast Internal Computation via Smith Normal Forms

As discussed in Sect. 1.3, the most significant operation that is performed by our attacker which is non-trivial in terms of its computational cost is checking in each iteration whether or not a given linear equation follows from the linear equations already in the set $\mathcal{L}$. When considering groups of prime order, this can be done simply by testing for linear independence among the vectors of coefficients corresponding to these equations. When considering groups of composite order, this is a bit more subtle, and in what follows we show that this can be done for example by relying on fast algorithms for computing the Smith normal form of integer matrices (e.g., [Sto96]) and without knowing the factorization of the order of the group.

The Smith normal form. The Smith normal form is a canonical diagonal form for equivalence of matrices over a principal ideal ring R. For any $\mathbf{A} \in R^{n \times m}$ there exist square invertible matrices $\mathbf{S}$ and $\mathbf{T}$ over R such that $\mathbf{D} = \mathbf{SAT}$ is the all-zeros matrix except for the first r terms $s_1, \ldots, s_r$ on its main diagonal, where $s_i | s_{i+1}$ for every $0 \leq i \leq r-1$. The matrix $\mathbf{D}$ is called the Smith normal form of $\mathbf{A}$ and it is unique up to multiplications of its non-zero terms by units. The Smith normal form was first proven to exist by Smith [Smi61] for matrices over the integers, and in this case each s_i is positive, $r = \mathrm{rank}(\mathbf{A})$ and $|\det(\mathbf{S})| = |\det(\mathbf{T})| = 1$. For our purposes we consider Smith forms of integer matrices, and we will not be relying on the fact that $s_i | s_{i+1}$ for every $0 \leq i \leq r-1$.

A fast algorithm for computing Smith normal forms over the integers was presented by Storjohann [Sto96]. His algorithm requires $\tilde{O}\left(n^{\omega-1} m \cdot \mathsf{M}\left(n \log ||\mathbf{A}||\right)\right)$ bit operations for computing the Smith normal form of a matrix $\mathbf{A} \in \mathbb{Z}^{n \times m}$, where ω is the exponent for matrix multiplication over rings (i.e., two $n \times n$ matrices can be multiplied in $O(n^\omega)$ ring operations), $\mathsf{M}(t)$ bounds the number of bit operations required for multiplying two $\lceil t \rceil$-bit integers, and $||\mathbf{A}|| = \max |\mathbf{A}_{i,j}|$.

Efficiently realizing our attacker. Let $\mathcal{L}$ be a set of linear equations over $\mathbb{Z}_N$ in the formal variables $\mathbf{Z} = (Z_1, \ldots, Z_k)$, and let $\ell(\mathbf{Z})$ be an additional such linear equation. Then, we would like to determine whether or not there exists $\mathbf{z} \in \mathbb{Z}_N^k$ such that $\ell'(\mathbf{z})$ holds for every $\ell'(\mathbf{Z}) \in \mathcal{L}$ but $\ell(\mathbf{z})$ does not hold (i.e., ℓ is not implied by $\mathcal{L}$).

Denote $\mathcal{L} = \{\langle \mathbf{a}^{(i)}, \mathbf{Z} \rangle = b_i \mod N : i \in [t]\}$, where $t = |\mathcal{L}|$, $\mathbf{a}^{(i)} \in \mathbb{Z}^k$ and $b_i \in \mathbb{Z}$ for every $i \in [t]$ (that is, we identify $\mathbb{Z}_N$ with $\{0, \ldots, N-1\} \subseteq \mathbb{Z}$).

First, we convert our equations to equations over $\mathbb{Z}$ by adding new variables $\mathbf{W} = (W_1, \ldots, W_t)$ and for each $i \in [t]$ we convert the equation $\langle \mathbf{a}^{(i)}, \mathbf{Z} \rangle = b_i \mod N$ into the equation

$$\langle \mathbf{a}^{(i)}, \mathbf{Z} \rangle + N \cdot W_i = b_i \ .$$

In matrix notation we let

$$\mathbf{A} = \left[\begin{array}{c|c} \mathbf{a}^{(1)} & \\ \vdots & N \cdot I_{t\times t} \\ \mathbf{a}^{(t)} & \end{array}\right] \in \mathbb{Z}^{(k+t)\times t}, \ \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_t \end{bmatrix} \in \mathbb{Z}^t, \ \mathbf{v} = \begin{bmatrix} \mathbf{Z} \\ \mathbf{W} \end{bmatrix} \in \mathbb{Z}^{k+t},$$

and then our system of linear equations is $\mathbf{Av} = \mathbf{b}$. Next, we compute the Smith normal form of $\mathbf{A}$, that is, we find matrices $\mathbf{S} \in \mathbb{Z}^{(k+t)\times(k+t)}$ and $\mathbf{T} \in \mathbb{Z}^{t\times t}$ that are invertible over $\mathbb{Z}$ (i.e., $|\det\mathbf{S}| = |\det\mathbf{T}| = 1$), such that the matrix $\mathbf{D} = \mathbf{SAT}$ is zero everywhere except for the first r terms on its main diagonal for some $0 \leq r \leq t$. Now, by multiplying from left by $\mathbf{S}$, our system is the same as $\mathbf{SATT}^{-1}\mathbf{v} = \mathbf{Sb}$, and denoting $\mathbf{u} = \mathbf{T}^{-1}\mathbf{v}$ and $\mathbf{c} = \mathbf{Sb}$, we obtain the equivalent system $\mathbf{Du} = \mathbf{c}$. Let $d_1, \ldots, d_r$ be the non-zero diagonal values of $\mathbf{D}$. If there exists $i \in [r]$ such that d_i does not divide c_i, or $r \leq i \leq k+t$ such that $c_i \neq 0$ then the system does not have any solution. Otherwise, the general solution for the system $\mathbf{Du} = \mathbf{c}$ is of the form $\mathbf{u} = (u_1, \ldots, u_{k+t}) = (c_1/d_1, \ldots, c_r/d_r, y_1, \ldots, y_s)$, where $s = k + t - r$ and the y coordinates can take any value.

Now, let $\ell(\mathbf{Z})$ be another linear equation in $\mathbb{Z}_N$, and denote it by $\langle \mathbf{a}', \mathbf{Z} \rangle = b' \mod N$, where $\mathbf{a}' \in \mathbb{Z}^k$ and $b' \in \mathbb{Z}$ (recall that we identify $\mathbb{Z}_N$ with $\{0, \ldots, N-1\} \subseteq \mathbb{Z}$ as mentioned above). We may substitute $\mathbf{Z} = \mathbf{T}'\mathbf{u}$, where $\mathbf{T}' \in \mathbb{Z}^{k\times t}$ consists of the first k rows of $\mathbf{T}$. Then, we obtain the linear equation $\langle \mathbf{a}', \mathbf{T}'\mathbf{u} \rangle = b' \mod N$. Substituting the general solution $\mathbf{u} = (c_1/d_1, \ldots, c_r/d_r, y_1, \ldots, y_s)$, we obtain a linear equation of the form $\sum_{i=1}^{s} \alpha_i y_i = \beta \mod N$. If $\beta = 0 \mod N$ and $\alpha_i = 0 \mod N$ for all $i \in [s]$ then every $z \in \mathbb{Z}^k$ satisfying $\mathcal{L}$ also satisfies $\ell(\mathbf{Z})$. Otherwise, if $\beta \neq 0 \mod N$ then the solution corresponding to $(y_1, \ldots, y_s) = (0, \ldots, 0)$ satisfies $\mathcal{L}$ but does not satisfy $\ell(\mathbf{Z})$, and if $\beta = 0 \mod N$ but there exists $i \in [s]$ such that $\alpha_i \neq 0 \mod N$ then the solution corresponding to $(y_1, \ldots, y_s) = e_i$ satisfies $\mathcal{L}$ but does not satisfy $\ell(\mathbf{Z})$.

References

[BBB+18] Boneh, D., Bonneau, J., Bünz, B., Fisch, B.: Verifiable delay functions. In: Shacham, H., Boldyreva, A. (eds.) CRYPTO 2018. LNCS, vol. 10991, pp. 757–788. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96884-1_25

[BBF18] Boneh, D., Bünz, B., Fisch, B.: A survey of two verifiable delay functions. Cryptology ePrint Archive, Report 2018/712 (2018)

[BGJ+16] Bitansky, N., Goldwasser, S., Jain, A., Paneth, O., Vaikuntanathan, V., Waters, B.: Time-lock puzzles from randomized encodings. In: Proceedings of the 7th Conference on Innovations in Theoretical Computer Science, pp. 345–356 (2016)

[BL96] Boneh, D., Lipton, R.J.: Algorithms for black-box fields and their application to cryptography. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 283–297. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-68697-5_22

[CP18] Cohen, B., Pietrzak, K.: Simple proofs of sequential work. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018. LNCS, vol. 10821, pp. 451–467. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78375-8_15

[DGM+19] Döttling, N., Garg, S., Malavolta, G., Vasudevan, P.N.: Tight verifiable delay functions. Cryptology ePrint Archive, Report 2019/659 (2019)

[DMP+19] De Feo, L., Masson, S., Petit, C., Sanso, A.: Verifiable delay functions from supersingular isogenies and pairings. Cryptology ePrint Archive, Report 2019/166 (2019)

[DN92] Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 139–147. Springer, Heidelberg (1993). https://doi.org/10.1007/3-540-48071-4_10

[EFK+19] Ephraim, N., Freitag, C., Komargodski, I., Pass, R.: Continuous verifiable delay functions. Cryptology ePrint Archive, Report 2019/619 (2019)

[GW11] Gentry, C., Wichs, D.: Separating succinct non-interactive arguments from all falsifiable assumptions. In: Proceedings of the 43rd Annual ACM Symposium on Theory of Computing, pp. 99–108 (2011)

[JS08] Jager, T., Schwenk, J.: On the equivalence of generic group models. In: Baek, J., Bao, F., Chen, K., Lai, X. (eds.) ProvSec 2008. LNCS, vol. 5324, pp. 200–209. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88733-1_14

[Kil92] Kilian, J.: A note on efficient zero-knowledge proofs and arguments. In: Proceedings of the 24th Annual ACM Symposium on Theory of Computing, pp. 723–732 (1992)

[LW15] Lenstra, A.K., Wesolowski, B.: A random zoo: sloth, unicorn, and trx. Cryptology ePrint Archive, Report 2015/366 (2015)

[Mau05] 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

[Mic94] Micali, S.: CS proofs. In: Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science, pp. 436–453 (1994)

[MMV11] Mahmoody, M., Moran, T., Vadhan, S.: Time-lock puzzles in the random oracle model. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 39–50. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22792-9_3

[MMV13] Mahmoody, M., Moran, T., Vadhan, S.P.: Publicly verifiable proofs of sequential work. In: Proceedings of the 4th Conference on Innovations in Theoretical Computer Science, pp. 373–388 (2013)

[MSW19] Mahmoody, M., Smith, C., Wu, D.J.: A note on the (im)possibility of verifiable delay functions in the random oracle model. Cryptology ePrint Archive, Report 2019/663 (2019)

[MW98] Maurer, U., Wolf, S.: Lower bounds on generic algorithms in groups. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 72–84. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054118

[Nec94] Nechaev, V.I.: Complexity of a determinate algorithm for the discrete logarithm. Math. Notes **55**(2), 91–101 (1994). https://doi.org/10.1007/BF02113297

[Pie19] Pietrzak, K.: Simple verifiable delay functions. In: Proceedings of the 10th Conference on Innovations in Theoretical Computer Science, pp. 60:1–60:15 (2019)

[RSW96] Rivest, R.L., Shamir, A., Wagner, D.A.: Time-lock puzzles and timed-release crypto (1996)

[Sho97] 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

[Smi61] Smith, H.J.S.: On systems of linear indeterminate equations and congruences. Philos. Trans. R. Soc. **151**(1), 293–326 (1861)

[Sto96] Storjohann, A.: Near optimal algorithms for computing Smith normal forms of integer matrices. In: Proceedings of the International Symposium on Symbolic and Algebraic Computation, pp. 267–274 (1996)

[Wes19] Wesolowski, B.: Efficient verifiable delay functions. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 379–407. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_13