The Discrete-Logarithm Problem with Preprocessing

Henry Corrigan-Gibbs and Dmitry Kogan

Stanford University

Abstract. This paper studies discrete-log algorithms that use *preprocessing*. In our model, an adversary may use a very large amount of precomputation to produce an "advice" string about a specific group (e.g., NIST P-256). In a subsequent online phase, the adversary's task is to use the preprocessed advice to quickly compute discrete logarithms in the group. Motivated by surprising recent preprocessing attacks on the discrete-log problem, we study the power and limits of such algorithms.

In particular, we focus on *generic* algorithms— these are algorithms that operate in every cyclic group. We show that any generic discrete-log algorithm with preprocessing that uses an S-bit advice string, runs in online time T, and succeeds with probability ϵ, in a group of prime order N, must satisfy $ST^2 = \widetilde{\Omega}(\epsilon N)$. Our lower bound, which is tight up to logarithmic factors, uses a synthesis of incompressibility techniques and classic methods for generic-group lower bounds. We apply our techniques to prove related lower bounds for the CDH, DDH, and multiple-discrete-log problems.

Finally, we demonstrate two new generic preprocessing attacks: one for the multiple-discrete-log problem and one for certain decisional-type problems in groups. This latter result demonstrates that, for generic algorithms with preprocessing, distinguishing tuples of the form $(g, g^x, g^{(x^2)})$ from random is much easier than the discrete-log problem.

1 Introduction

The problem of computing discrete logarithms in groups is fundamental to cryptography: it underpins the security of widespread cryptographic protocols for key exchange [31], public-key encryption [26,34], and digital signatures [46,53,68].

In the absence of an unconditional proof that computing discrete logarithms is hard, one fruitful research direction has focused on understanding the hardness of these problems against certain restricted classes of algorithms [6,61,71]. In particular, Shoup considered discrete-log algorithms that are *generic*, in the sense that they only use the group operation as a black box [71]. Generic algorithms are useful in practice since they apply to every group. In addition, lower bounds against generic algorithms are meaningful because, in popular elliptic-curve groups, generic attacks are the best known [38,51].

The traditional notion of generic algorithms models *online-only attacks*, in which the adversary simultaneously receives the description of a cyclic group

$\mathbb{G} = \langle g \rangle$ and a problem instance $g^x \in \mathbb{G}$. In this model, when the attack algorithm begins executing, the attacker has essentially no information about the group $\mathbb{G}$. Shoup [71] showed that, in this online-only setting, every generic discrete-log algorithm that succeeds with good probability in a group of prime order N must run in time at least $N^{1/2}$.

In practice, however, an adversary may have access to the description of the group $\mathbb{G}$ long before it has to solve a discrete-log problem instance. In particular, the vast majority of real-world cryptosystems use one of a handful of groups, such as NIST P-256, Curve25519 [12], or the DSA groups. In this setting, a real-world adversary could potentially perform a *preprocessing attack* [28, 32, 45] relative to a popular group: In an offline phase, the adversary would compute and store a data structure ("advice string") that depends on the group $\mathbb{G}$. In a subsequent online phase, the adversary could use its precomputed advice to solve the discrete-log problem in the group $\mathbb{G}$ much more quickly than would be possible in an online-only attack.

In recent work, Mihalcik [59] and Bernstein and Lange [13] demonstrated the surprising power of preprocessing attacks against the discrete-log problem. In particular, they construct a *generic* algorithm with preprocessing that computes discrete logarithms in every group of order N using $N^{1/3}$ bits of group-specific advice and roughly $N^{1/3}$ online time. Since their algorithm is generic, it applies to every group, including popular elliptic-curve groups. In contrast, Shoup's result shows that, without preprocessing, every generic discrete-log algorithm requires at least $N^{1/2}$ time. The careful use of a large amount of preprocessing—roughly $N^{2/3}$ operations—is what allows the attack of Mihalcik, Bernstein, and Lange to circumvent this lower bound.

As of now, there is no reason to believe that the attack of Mihalcik, Bernstein, and Lange is the best possible. For example, we know of no results ruling out a generic attack that uses $N^{1/2}$ precomputation to build an advice string of size $N^{1/8}$, which can be used to compute discrete logs in online time $N^{1/8}$.

The existence of such an attack would—at the very least—shake our confidence in 256-bit elliptic-curve groups. An attacker who wanted to break NIST P-256, for example, could perform a one-time 2^{128} precomputation to compute a 2^{32}-bit advice string. Given this advice string, an attacker could compute discrete logarithms on the P-256 curve in online time 2^{32}. The precomputed advice string would essentially be a "trapdoor" that would allow its holder to compute discrete-logs on the curve in seconds.

The possibility of such devastating discrete-log preprocessing attacks, and the lack of lower-bounds for such algorithms, leads us to ask:

How helpful can preprocessing be to generic discrete-log algorithms?

In this paper, we extend the classic model of generic algorithms to capture preprocessing attacks. To do so, we introduce the notion of *generic algorithms with preprocessing* for computational problems in cryptographic groups. These algorithms make only black-box use of the group operation, but may perform a large number of group operations during a preprocessing phase. Following prior

work on preprocessing attacks [28, 32, 35, 45], we measure the complexity of such algorithms by (a) the size of the advice string that the algorithm produces in the preprocessing phase, and (b) the running time of the algorithm's online phase.

These two standard cost metrics do not consider the preprocessing time required to compute the advice string. Ignoring the preprocessing cost only strengthens the resulting lower bounds, but it leaves open the question of how much preprocessing is really necessary to compute a useful advice string. Towards the end of this paper, we take up this question as well by extending our model to account for preprocessing time.

1.1 Our Results

We prove new lower bounds on generic algorithms with preprocessing that relate the time, advice, and preprocessing complexity of generic discrete-log algorithms, and algorithms for related problems. We also introduce new generic preprocessing attacks for the multiple-discrete-log problem and for certain distinguishing problems in groups.

Lower Bounds for Discrete Log and CDH. We prove in Theorem 2 that every generic algorithm that uses S bits of group-specific precomputed advice and that computes discrete logarithms in online time T with success probability ϵ must satisfy $ST^2 = \widetilde{\Omega}(\epsilon N)$, where the $\widetilde{\Omega}(\cdot)$ notation hides logarithmic factors in N. When $S = T$ the bound shows that, for constant ϵ, the best possible generic attack must use roughly $N^{1/3}$ bits of advice and runs in online time roughly $N^{1/3}$.

Our lower bound is tight, up to logarithmic factors, for the full range of parameters S, T, and ϵ, since the attack of Mihalcik [59] and Bernstein and Lange [13], which we summarize in Sect. 7.1, gives a matching upper bound. (These attacks sidestep Shoup's $N^{1/2}$-time lower bound for generic discrete-log algorithms [71] by using more than $N^{1/2}$ time in their preprocessing phase.) As a consequence, beating the preprocessing algorithm of Mihalcik, Bernstein, and Lange on the NIST P-256 curve, for example, would require developing a new non-generic attack.

Our lower bound extends naturally to the computational Diffie-Hellman problem, for which we also prove an $ST^2 = \widetilde{\Omega}(\epsilon N)$ lower bound (Theorem 6), and the M-instance multiple-discrete-log problem, for which we prove an $ST^2/M + T^2 = \widetilde{\Omega}(\epsilon^{1/M} MN)$ lower bound (Theorem 8). The attacks of Sect. 7 show that these lower bounds are tight.

Lower Bound for DDH with Preprocessing. We also look at the more subtle case of distinguishing attacks. We show in Theorem 9, that every generic distinguisher with preprocessing that achieves advantage ϵ against the decisional Diffie-Hellman problem (DDH) must satisfy $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$. The quadratic dependence on the error probability makes this bound weaker than the previous ones. We know of no DDH distinguisher that matches this lower bound for all parameter ranges (e.g., for $\epsilon = N^{-1/4}$), and we leave the question of whether such a distinguisher exists as an open problem.

Lower Bound on Preprocessing Time. In addition, we prove lower bounds on the amount of computation required to produce the advice string in the preprocessing phase of a generic discrete-log algorithm. We show in Theorem 10 that any such algorithm that uses preprocessing time P, online time T, and achieves success probability ϵ must satisfy: $PT + T^2 = \Omega(\epsilon N)$. Our lower bound matches the preprocessing time used by the discrete-log preprocessing attack of Mihalcik, Bernstein, and Lange, and essentially rules out the existence of very fast generic algorithms that also use modest amounts of preprocessing. For example, any generic algorithm that runs in online time $T = N^{1/8}$ must use close to $N^{7/8}$ preprocessing time to succeed with good probability—no matter how large of an advice string it uses.

New Preprocessing Attacks. Finally, in Theorem 11, we introduce a new preprocessing algorithm for the multiple-discrete-log problem that shows that our lower bound is tight for constant ϵ. In addition, for the problem of distinguishing tuples of the form $(g, g^x, g^{(x^2)})$ from random, Theorem 13 gives a new algorithm that satisfies $ST^2 = \widetilde{O}(\epsilon^2 N)$. The existence of such an algorithm is especially surprising because solving the $(g, g^x, g^{(x^2)})$ distinguishing problem is *as hard as* computing discrete logarithms for *online-only* algorithms. In contrast, our algorithm shows that this problem is *substantially easier* than computing discrete logarithms for *preprocessing* algorithms: computing discrete logarithms requires $S = T = 1/\epsilon = N^{1/4}$ while our new distinguishing attack requires $S = T = 1/\epsilon = N^{1/5}$.

1.2 Our Techniques

The starting point of our lower bounds is an incompressibility argument, which is also at the heart of classic lower bounds against preprocessing algorithms (also known as "non-uniform algorithms") for inverting one-way permutations [42,76,77] and random functions [32]. At a high level, our approach is to show that if there exists a generic discrete-log algorithm $\mathcal{A}$ that (a) uses few bits of preprocessed advice and (b) uses few online group operations, then we can use such an algorithm $\mathcal{A}$ to compress a random permutation.

Incompressibility. The first technical challenge is that a straightforward application of incompressibility techniques does not suffice in the setting of generic groups. To explain the difficulty, let us sketch the argument that a random permutation oracle π is one-way, even against preprocessing adversaries [28, 42, 76, 77]. The argument builds a compression scheme by invoking $\mathcal{A}(x)$ on some point x in the image of π and answering $\mathcal{A}$'s queries to π. The key observation is that when $\mathcal{A}$ produces its output $y = \pi^{-1}(x)$, we have learned some extra information about π beyond the information that the query responses contain. In this way, each invocation of $\mathcal{A}$ yields some "profit," in terms of our knowledge of π. We can use this profit to compress π.

To apply this argument to generic groups, we could replace the random permutation oracle π by an oracle that implements the group operation for a random group. (We define the model precisely in Sect. 2.) The challenge is that

a group-operation oracle has extra structure that a random permutation oracle does not. This extra structure fouls up the standard incompressibility argument, since the query responses that the compression routine must feed to $\mathcal{A}$ might themselves contain enough information to recover the discrete log that $\mathcal{A}$ will later output. If this happens, the compression scheme will not "profit" at all from invoking $\mathcal{A}$, and we will not be able to use $\mathcal{A}$ to compress the oracle.

To handle this case, we notice that this sort of compression failure only occurs when two distinct queries to the group oracle return the same string. By using a slightly more sophisticated compression routine, which notices and compensates for these "collision" events, we achieve compression even where the traditional incompressibility argument would have failed. (Dodis et al. [33] use a similar observation in their analysis of the RSA-FDH signature scheme.)

To keep track of when these collision events occur, we adopt an idea from Shoup's generic-group lower-bound proof [71], which does not use incompressibility at all. Shoup's idea is to keep a careful accounting of the information that the adversary's queries have revealed about the generic-group oracle at any point during the execution. Our compression scheme exploits a similar accounting strategy, which allows it to halt the adversary $\mathcal{A}$ as soon as the compressor notices that continuing to run $\mathcal{A}$ would be "unprofitable."

Handling Randomized Algorithms. The second technical challenge we face is in handling algorithms that succeed with arbitrarily small probability ϵ. The standard incompressibility methods invoke the algorithm $\mathcal{A}$ on many inputs, and the compression routine succeeds only if *all* of these executions succeed. If the algorithm $\mathcal{A}$ fails often, then we will fail to construct a useful compression scheme.

The naïve way around this problem would be to amplify $\mathcal{A}$'s success probability by having the compression scheme run the algorithm $\mathcal{A}$ many times on each input. The problem is that amplifying the success probability in this way decreases the "profit" that we gain from $\mathcal{A}$, since the compression scheme has to answer many more group-oracle queries in the amplified algorithm than in the unamplified algorithm. As a result, this naïve amplification strategy yields an $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$ lower bound that is loose in its dependence on the success probability ϵ.

Our approach is to leverage the observation, applied fruitfully to the random-permutation model by De et al. [28], that it is without loss of generality to assume that the compression and decompression algorithms share a common string of independent random bits. Rather than amplifying the success probability of $\mathcal{A}$ by iteration, the compression scheme simply finds a set of random bits in the shared random string that cause $\mathcal{A}$ to produce the correct output. The compression scheme then writes this pointer out as part of the compressed representation of the group oracle. This optimization yields the tight $ST^2 = \widetilde{\Omega}(\epsilon N)$ lower bound.

Along the way, we exploit the random self-reducibility of the discrete-log problem to transform an average-case discrete-log algorithm, which succeeds on a *random* instance with probability ϵ, to a worst-case algorithm, which succeeds on *every* instance with probability ϵ. Using the random self-reduction substantially

simplifies the incompressibility argument, since it allows the compression routine to invoke the algorithm $\mathcal{A}$ on arbitrary inputs.

Generalizing to Decisional Problems. The final technical challenge is to extend our core incompressibility argument to give lower bounds for the decisional Diffie-Hellman Problem (DDH). The difficulty with using a DDH algorithm to build a compression scheme is that each execution of the DDH distinguisher only produces a single bit of information. Furthermore, if the distinguishing advantage ϵ is small, the distinguisher produces only a fraction of a bit of information. The straightforward amplification would again work but would yield a very loose $ST^2 = \widetilde{\Omega}(\epsilon^4 N)$ bound.

To get around this issue, we execute the distinguisher on large batches of input instances. We judiciously choose the batch size to balance the profit from each batch with the probability that all runs in a batch succeed. Handling collision events in this case requires extra care. Putting these ingredients together, we achieve an $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$ lower bound for the DDH problem.

1.3 Related Work

This paper builds upon two major lines of prior work: one on *preprocessing* lower bounds for *symmetric-key problems*, and the other on *online* lower bounds for *generic algorithms in groups*. We prove *preprocessing* lower bounds for *generic algorithms* and, indeed, our proofs use a combination of techniques from both prior settings.

Incompressibility Methods. One prominent related area of research puts lower bounds on the efficiency of *preprocessing algorithms* for inverting random functions and random permutations. An early motivation was Hellman's preprocessing algorithm ("Hellman tables") for inverting random functions [45]. Fiat and Naor [35] later extended the technique to allow inverting general functions and Oechslin [63] proposed practical improvements to Hellman's construction.

Yao [77] used an incompressibility argument to show the optimality of Hellman's method for inverting random permutations. Gennaro and Trevisan [42] and Wee [76] proved related lower bounds, also using incompressibility methods. Barkan et al. [9] showed that, in a restricted model of computation, Hellman's method is optimal for inverting random functions (not just permutations).

De et al. [28] demonstrated how to use *randomized encodings*, essentially an incompressibility argument augmented with random oracles, to give alternative proofs of preprocessing lower bounds on the complexity of inverting random permutations and breaking general pseduo-random generators. We adopt the powerful randomized encoding technique of De et al. in our proofs. Dodis et al. [32] applied this technique to show that salting [60] defeats preprocessing attacks against certain computational tasks (e.g., collision finding) in the random-oracle model [10]. Abusalah et al. [2] used the technique to construct proofs of space from random functions.

Unruh [74] gave an elegant framework for proving the hardness of computational problems in the random-oracle model against preprocessing adversaries

(or against algorithms with "auxiliary input," in his terminology). He proves that if a computational problem is hard when a certain number of points of the random oracle are fixed ("presampled"), then the problem is hard in the random-oracle model against preprocessing adversaries using a certain amount of oracle-dependent advice. This presampling technique gives an often simpler alternative to incompressibility-based lower bounds. Coretti et al. [24] recently introduced new variants of Unruh's presampling technique that give tighter lower bounds against preprocessing adversaries for a broad set of problems.

Generic-Group Lower Bounds. All of the aforementioned work studies precomputation attacks on one-way permutations and one-way functions, which are essentially symmetric-key primitives. In the setting of public-key cryptography, a parallel— and quite distinct—line of work studies lower bounds on algorithms for the discrete-log problem and related problems in generic groups. All of these lower bounds study online-only algorithms (i.e., that do not use preprocessing).

In particular, Shoup [71] introduced the modern *generic-group model* to capture algorithms that make black-box use of a group operation. In Shoup's model, which draws on earlier treatments of black-box algorithms for groups [6,61], the discrete-logarithm problem in a group of prime order N requires time $\Omega(N^{1/2})$ to solve. Shoup's model captures many popular discrete-log algorithms, including Shanks' Baby-Step Giant-Step algorithm [70], Pollard's Rho and Kangaroo algorithms [67], and the Pohlig-Hellman algorithm [66]. For computing discrete logarithms on popular elliptic curves, variants of these algorithms are the best known [11, 39, 75, 80].

Subsequent works used Shoup's model to prove lower bounds against generic algorithms for RSA-type problems [27], knowledge assumptions [30], the multiple-discrete-log problem [79], assumptions in groups with pairings [15], and for algorithms with access to additional oracles [57]. A number of works also prove the security of specific cryptosystems in the generic-group model [20, 21, 29, 36, 49, 69, 72]. Other work studies computational problems in generic *rings*, to analyze generic algorithms for RSA-type problems [4, 55].

Preprocessing Attacks in Generic Groups. The works most relevant to our new algorithms with preprocessing are Mihalcik's master's thesis [59], which surveys preprocessing attacks on the discrete-logarithm problem, and the paper of Bernstein and Lange [13], which demonstrated preprocessing attacks—both generic and non-generic— on a wide range of symmetric- and public-key primitives. We design new preprocessing attacks against the multiple-discrete-logarithm problem and against a large class of distinguishing problems in groups.

Non-generic discrete-log algorithms. In certain groups there are non-generic discrete-log attacks that dramatically outperform the generic ones. The landscape of non-generic discrete-log algorithms is vast, so we refer the reader to the 2000 survey of Odlyzko [62] and the 2014 survey of Joux et al. [47] for details. To give a taste of these results: when computing discrete logarithms in finite fields $\mathbb{F}_{p^n}$, the running time of the best discrete logarithms depend on the relative size of p and n. When $p \ll n$, a recent algorithm of Barbulescu et al. [8]

computes discrete logarithms in quasi-polynomial time. When $p \gg n$, the best methods are based on "index calculus" techniques and run in sub-exponential time $e^{O((\log p)^{1/3}(\log\log p)^{2/3})}$ [44, 56]. The analysis of these algorithms is heuristic, in that it relies on some unproved (but reasonable) number-theoretic assumptions.

In certain classes of elliptic-curve groups, there are non-generic algorithms for the discrete-log problem that outperform the generic algorithms [41]; some such algorithms run in sub-exponential time [58], or even in polynomial time [73]. In the standard elliptic-curve groups used for key exchange (e.g., NIST P-256) however, the generic preprocessing attacks discussed in this paper are still essentially the best known.

Non-generic discrete-log algorithms also benefit from preprocessing. Coppersmith demonstrated a sub-exponential-time preprocessing attack on the integer factorization problem [23] that also yields a non-generic sub-exponential-time preprocessing attack on the finite-field discrete-log problem [7, 13]. Adrian et al. [3] show how to use such an attack compute discrete logs modulo a 512-bit prime in less than a minute of online time.

Organization of This Paper. In Sect. 2, we introduce notation, our model of computation, and a key lemma. In Sect. 3, we prove a lower bound on generic algorithms with preprocessing for the discrete-logarithm and CDH problems. In Sects. 4 and 5, we extend these bounds to the multiple-discrete-logarithm and DDH problems. In Sect. 6, we investigate the amount of precomputation such generic preprocessing algorithms require. In Sect. 7, we introduce new generic preprocessing attacks. In Sect. 8, we conclude with open questions.

2 Background

In this section, we recall the standard model of computation in generic groups, we introduce our model of generic algorithms with preprocessing, and we recall an incompressibility lemma that will be essential to our proofs.

Notation. We use $\mathbb{Z}_N$ to denote the ring of integers modulo N, $[N]$ indicates the set $\{1, \ldots, N\}$, and $\mathbb{Z}^+$ indicates the set of positive integers. Throughout this paper, we take N to be prime, so $\mathbb{Z}_N$ is also a field. We use the notation $x \leftarrow 5$ to indicate the assignment of a value to a variable and, when S is a finite set, the notation $x \xleftarrow{R} S$ indicates that x is a sample from the uniform distribution over S. For a probability distribution $\mathcal{D}$, $d \sim \mathcal{D}$ indicates that d is a random variable distributed according to $\mathcal{D}$. The statement $f(x) \stackrel{\text{def}}{=} x^2 - x$ indicates the definition of a function f. All logarithms are base two, unless otherwise noted.

We use the standard Landau notation $O(\cdot)$, $\Theta(\cdot)$, $\Omega(\cdot)$, and $o(\cdot)$ to indicate the asymptotics of a function. For example $f(N) = O(g(N))$ if there exists a constant $c > 0$ such that for all large enough N, $|f(N)| \leq c \cdot g(N)$. When there are many variables inside the big-O, as in $f(N) = O(N/ST)$, all variables other than N are implicit functions of N. The tilde notation $\widetilde{O}(\cdot)$ and $\widetilde{\Omega}(\cdot)$ hides polylogarithmic factors in N. So, we can say for example that $S \log^2 N = \widetilde{O}(S)$.

Generic Algorithms. Following Shoup [71], we model a generic group using a random injective function σ that maps the integers in $\mathbb{Z}_N$ (representing the set of

discrete logarithms) to a set of labels $\mathcal{L}$ (representing the set of group elements). We then write the elements of an order-N group as $\{\sigma(1), \sigma(2), \ldots, \sigma(N)\}$, instead of the usual $\{g, g^2, \cdots, g^N\}$. We often say that $i \in \mathbb{Z}_N$ is the "discrete log" of its label $\sigma(i) \in \mathcal{L}$.

The *generic group oracle* $\mathcal{O}_\sigma(\cdot, \cdot)$ for a labeling function σ takes as input two strings $s_i, s_j \in \mathcal{L}$ and responds as follows:

- If the arguments to the oracle are in the image of σ, then we can write $s_i = \sigma(i)$ and $s_j = \sigma(j)$. The oracle responds with $\sigma(i + j)$, where the addition is modulo the group order N.
- If either of the arguments to the oracle falls outside of the image of σ, the oracle returns $\perp$.

Given such an oracle and a label $\sigma(x)$, it is possible to compute $\sigma(\alpha x)$ for any constant $\alpha \in \mathbb{Z}_N$ using $O(\log N)$ oracle queries, by repeated squaring.

Some authors define the group oracle $\mathcal{O}_\sigma$ with a second functionality that maps labels $\sigma(x)$ to their inverses $\sigma(-x)$ in a single query. Our oracle can simulate this inversion oracle in at most $O(\log N)$ queries. To do so: given an element $\sigma(x)$, compute the element $\sigma((N-1)x) = \sigma(-x)$. Since providing an inversion oracle can decrease a generic algorithm's running time by at most a logarithmic factor, we omit it for simplicity.

A *generic algorithm* for $\mathbb{Z}_N$ on $\mathcal{L}$ is a probabilistic algorithm that takes as input a list of labels $(\sigma(x_1), \ldots, \sigma(x_L))$ and has oracle access to $\mathcal{O}_\sigma$. We measure the time complexity of a generic algorithm by counting the number of queries it makes to the generic group oracle.

Although the generic algorithms we consider may be probabilistic, we require that for every choice of σ, inputs, and random tapes, every algorithm halts after a finite number of steps. In this way, for every group order $N \in \mathbb{Z}^+$, we can compute an upper bound on the number of random bits the algorithm uses by iterating over all possible labelings, inputs, and random tapes. For this reason, we need only consider finite probability spaces in our discussion.

Generic Algorithms with Preprocessing. A *generic algorithm with preprocessing* is a pair of generic algorithms $(\mathcal{A}_0, \mathcal{A}_1)$ for $\mathbb{Z}_N$ on $\mathcal{L}$ such that:

- Algorithm $\mathcal{A}_0$ takes the label $\sigma(1)$ as input, makes some number of queries to the oracle $\mathcal{O}_\sigma$ ("preprocessing queries"), and outputs an advice string st_σ.
- Algorithm $\mathcal{A}_1$ takes as input the advice string st_σ and a list of labels $(\sigma(x_1), \ldots, \sigma(x_L))$, makes some number of queries to the oracle $\mathcal{O}_\sigma$ ("online queries"), and produces some output.

We typically measure the complexity of the algorithm $(\mathcal{A}_0, \mathcal{A}_1)$ by (a) the size of the advice string st_σ that $\mathcal{A}_0$ outputs, and (b) the number of oracle queries that algorithm $\mathcal{A}_1$ makes.

In Sect. 6, we consider generic algorithms with preprocessing for which the running time of $\mathcal{A}_0$ (i.e., the preprocessing time) is also bounded. In all other sections, we put no running time bound on $\mathcal{A}_0$, so without loss of generality, we may assume in these sections that $\mathcal{A}_0$ is deterministic.

Incompressibility Arguments. We use the following proposition of De et al. [28], which formalizes the notion that it is impossible to compress every element in a set $\mathcal{X}$ to a string less than $\log|\mathcal{X}|$ bits long, even relative to a random string.

Proposition 1 (De, Trevisan, and Tulsiani [28]). *Let $E : \mathcal{X} \times \{0,1\}^\rho \to \{0,1\}^m$ and $D : \{0,1\}^m \times \{0,1\}^\rho \to \mathcal{X}$ be randomized encoding and decoding procedures such that, for every $x \in \mathcal{X}$, $\Pr_{r \leftarrow \{0,1\}^\rho}\left[D(E(x,r),r) = x\right] \geq \delta$. Then $m \geq \log|\mathcal{X}| - \log 1/\delta$.*

Notice that the encoding and decoding algorithms of Proposition 1 take *the same* random string r as input. Additionally, that bound on the string length m is independent of the number of random bits that these routines take as input. As a consequence, Proposition 1 holds even when the algorithms E and D have access to a common random oracle.

3 Lower Bound for Discrete Logarithms

In this section we prove that every generic algorithm that uses S bits of group-specific precomputed advice and that computes discrete logs in online time T with probability ϵ must satisfy $ST^2 = \widetilde{\Omega}(\epsilon N)$.

Theorem 2. *Let N be a prime. Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a pair of generic algorithms for $\mathbb{Z}_N$ on $\mathcal{L}$, such that $\mathcal{A}_0$ outputs an S-bit state, $\mathcal{A}_1$ makes at most T oracle queries, and*

$$\Pr_{\sigma,x,\mathcal{A}_1}\left[\mathcal{A}_1^{\mathcal{O}_\sigma}\left(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x)\right) = x\right] \geq \epsilon,$$

where the probability is taken over the uniformly random choice of the labeling σ, the instance $x \in \mathbb{Z}_N$, and the coins of $\mathcal{A}_1$. Then $ST^2 = \widetilde{\Omega}(\epsilon N)$.

Remark. The statement of Theorem 2 models the case in which the group generator $\sigma(1)$ is fixed, and the online algorithm must compute the discrete-log of the instance $\sigma(x)$ with respect to the fixed generator. Using a fixed generator is essentially without loss of generality, since an algorithm that computes discrete logarithms with respect to one generator can also be used to compute discrete logarithms with respect to any generator by increasing its running time by a factor of two. Because of this, we treat the generator as fixed throughout this paper.

Remark. Theorem 2 treats only prime-order groups. In the more general case of composite-order groups a similar result holds, except that the bound is $ST^2 = \widetilde{\Omega}(\epsilon p)$, where p is the largest prime factor of the group order. Since the techniques needed to arrive at this more general result are essentially the same as in the proof of Theorem 2, we focus on the prime-order case for simplicity.

We first give the idea behind the proof of Theorem 2 and then present a detailed proof.

Proof Idea for Theorem 2. Our proof uses an incompressibility argument. The basic idea is to compress the random labeling function σ using a discrete-log algorithm with preprocessing $(\mathcal{A}_0, \mathcal{A}_1)$. To do so, we write $\mathcal{A}_0$'s S-bit advice about σ into the compressed string. We then run $\mathcal{A}_1$ on many discrete-log instances $\sigma(x)$ and we write the T responses to $\mathcal{A}_1$'s queries into the compressed string. For each execution of $\mathcal{A}_1$, we only need to write T values of σ into the compressed string, but we get $T+1$ values of σ back, since the output of $\mathcal{A}_1(\sigma(x))$ gives us the value of x "for free." If S and T are simultaneously small, then we can compress σ using this method, which yields a contradiction.

However, this naïve technique might never yield any compression at all. The problem is that the T responses to $\mathcal{A}_1$'s queries might contain "collision events," in which the response to one of $\mathcal{A}_1$'s queries is equal to a previously seen query response. For example, say that $\mathcal{A}_1$ makes a query of the form $\mathcal{O}_\sigma(\sigma(x), \sigma(3))$ and the oracle's response is a string $\sigma(7)$ that also appeared in response to a previous query. In this case, just seeing the queries of $\mathcal{A}_1$ and their responses is enough to conclude that $x + 3 = 7 \bmod N$, which immediately yields the discrete log $x = 4$. This is problematic because even if $\mathcal{A}_1$ eventually halts and outputs $x = 4$, we have not received any "profit" from $\mathcal{A}_1$ since the T query responses themselves already contain all of the information we need to conclude that $x = 4$.

To profit in spite of these collisions, our compression scheme halts the execution of $\mathcal{A}_1$ as soon as it finds such a collision, since every collision event yields the discrete log being sought. The profit comes from the fact that, as long as the list of previous query responses is not too long, encoding a pointer to the collision-causing response requires many fewer bits than encoding an arbitrary element in the range of σ.

Our lower bound needs to handle randomized algorithms $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ that succeed with arbitrarily small probability ϵ. Yet to use $\mathcal{A}$ to compress σ, the algorithm $\mathcal{A}_1$ must succeed with very high probability. That is because the compression routine may invoke $\mathcal{A}_1$ as many as N times, and each execution must succeed for the compression scheme to succeed. The random self-reducibility of the discrete-log problem allows us to convert an average-case algorithm that succeeds on an ϵ fraction of instances (for a given labeling σ) to a worst-case algorithm that succeeds with probability ϵ on every instance (for a given labeling σ).

We still need to handle the fact that ϵ may be quite small. The straightforward way to amplify the success probability of $\mathcal{A}_1$ would be to construct an algorithm $\mathcal{A}_1'$ that runs R independent executions of $\mathcal{A}_1$ and that succeeds with probability at least $1 - \epsilon^R$. We could then use the amplified algorithm $(\mathcal{A}_0, \mathcal{A}_1')$ to compress σ.

The problem in our setting is that this simple amplification strategy yields a loose lower bound: if we run $\mathcal{A}_1$ for R iterations, and each iteration makes T queries, our compression scheme ends up "paying" for RT queries instead of T queries for each bit of "profit" it gets (i.e., for each output of $\mathcal{A}_1'$). Carrying this argument through yields an $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$ bound, which is worse than our goal of $\widetilde{\Omega}(\epsilon N)$.

Our idea is to leverage the correlated randomness between the compressor and decompressor to our advantage. In our compression scheme, the compressor runs

$\mathcal{A}_1$ using R sets of independent random coins, sampled from the random string shared with the decompressor. The compressor then writes into the compressed representation a $\log R$-bit pointer to a set of random coins (if one exists) that caused $\mathcal{A}_1$ to succeed. Using this strategy, instead of paying for RT queries per execution of $\mathcal{A}_1$, the compression scheme only pays for T queries, plus a small pointer. We can then choose R large enough to ensure that at least one of the R executions succeeds with extremely high probability. □

We now turn to the proof.

We say that a discrete-log algorithm succeeds in the *worst case* if it succeeds on every problem instance $\sigma(x)$ for $x \in \mathbb{Z}_N$. We say that a discrete-log algorithm succeeds in the *average case* if it succeeds on a random problem instance $\sigma(x)$ for $x \xleftarrow{R} \mathbb{Z}_N$.

We first use the random self-reducibility of the discrete-log problem to show that an average-case discrete-log algorithm implies a worst-case discrete-log algorithm. A lower bound on worst-case algorithms is therefore enough to prove Theorem 2. This is formalized in the next lemma.

Lemma 3 (Adapted from Abadi, Feigenbaum, and Kilian [1]). *Let N be a prime. Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a pair of generic algorithms for $\mathbb{Z}_N$ on $\mathcal{L}$ such that $\mathcal{A}_0$ outputs an S-bit advice string and $\mathcal{A}_1$ makes at most T oracle queries. Then, there exists a generic algorithm $\mathcal{A}'_1$ that makes at most $T + O(\log N)$ oracle queries and, for every $\sigma : \mathbb{Z}_N \to \mathcal{L}$, if $\Pr_{x,\mathcal{A}_1}\left[\mathcal{A}_1^{\mathcal{O}_\sigma}(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x)) = x\right] \geq \epsilon$, then for every $x \in \mathbb{Z}_N$, $\Pr_{\mathcal{A}'_1}\left[\mathcal{A}_1'^{\mathcal{O}_\sigma}(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x)) = x\right] \geq \epsilon$.*

Proof. On input $(\mathsf{st}_\sigma, \sigma(x))$, algorithm $\mathcal{A}'_1$ executes the following steps: First, it samples a random $r \xleftarrow{R} \mathbb{Z}_N$ and computes $\sigma(x + r)$, using $O(\log N)$ group operations. Then, it runs $\mathcal{A}_1(\mathsf{st}_\sigma, \sigma(x + r))$. Finally, when $\mathcal{A}_1$ outputs a discrete log x', algorithm $\mathcal{A}'_1$ outputs $x = x' - r \bmod N$.

Notice that $\mathcal{A}'_1$ invokes $\mathcal{A}_1$ on $\sigma(x + r)$, which is the image of a uniformly random point in $\mathbb{Z}_N$. Since $\mathcal{A}_1$ succeeds with probability at least ϵ over the random choice of $x \xleftarrow{R} \mathbb{Z}_N$ and its coins, $\mathcal{A}'_1$ succeeds with probability ϵ, only over the choice of its coins. □

To prove Theorem 2, we will use the generic algorithms $(\mathcal{A}_0, \mathcal{A}_1)$ to construct a randomized encoding scheme that compresses a good fraction of the labeling functions σ. The following lemma gives us such a scheme.

Lemma 4. *Let N be a prime. Let $G = \{\sigma_1, \sigma_2, \dots\}$ be a subset of the labeling functions from $\mathbb{Z}_N$ to $\mathcal{L}$. Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a pair of generic algorithms for $\mathbb{Z}_N$ on $\mathcal{L}$ such that for every $\sigma \in G$ and every $x \in \mathbb{Z}_N$, $\mathcal{A}_0$ outputs an S-bit advice string, $\mathcal{A}_1$ makes at most T oracle queries, and $(\mathcal{A}_0, \mathcal{A}_1)$ satisfy*

$$\Pr_{\mathcal{A}_1}\left[\mathcal{A}_1^{\mathcal{O}_\sigma}\left(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x)\right) = x\right] \geq \epsilon\,.$$

Then, there exists a randomized encoding scheme that compresses elements of G to bitstrings of length at most

$$\log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} + S + 1 - \frac{\epsilon N}{6T(T+1)(\log N + 1)}\,,$$

and succeeds with probability at least $1/2$.

We prove Lemma 4 in Sect. 3.1. Given the above two lemmas, we can prove Theorem 2.

Proof of Theorem 2. We say that a labeling σ is "good" if $(\mathcal{A}_0, \mathcal{A}_1)$ computes discrete logs with probability at least $\epsilon/2$ on σ. More precisely, a labeling σ is "good" if:

$$\Pr_{x,\mathcal{A}_1}\left[\mathcal{A}_1^{\mathcal{O}_\sigma}\left(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x)\right) = x\right] \geq \epsilon/2\,,$$

where the probability is taken over the choice of $x \in \mathbb{Z}_N$ as well as over the random tape of $\mathcal{A}_1$. Let G be the set of good labelings. A standard averaging argument [5, Lemma A.12] guarantees that an $\epsilon/2$ fraction of injective mappings from $\mathbb{Z}_N$ to $\mathcal{L}$ are good. Then $|G| \geq \epsilon/2 \cdot |\mathcal{L}|!/(|\mathcal{L}| - N)!$, where we've used the fact that the number of injective functions from $\mathbb{Z}_N$ to $\mathcal{L}$ is $|\mathcal{L}|!/(|\mathcal{L}| - N)!$.

Lemma 3 then implies that there exists a pair of generic algorithms $(\mathcal{A}_0, \mathcal{A}_1')$ such that for every $\sigma \in G$ and every $x \in \mathbb{Z}_N$, $\mathcal{A}_1'^{\mathcal{O}_\sigma}(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x))$ makes at most $T' = T + O(\log N)$ queries, and outputs x with probability at least $\epsilon/2$. Lemma 4 then implies that we can use $(\mathcal{A}_0, \mathcal{A}_1')$ to compress any labeling $\sigma \in G$ to a string of bitlength at most

$$\log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} + S + 1 - \frac{(\epsilon/2)N}{6T'(T'+1)(\log N + 1)}\,, \tag{1}$$

where the encoding scheme works with probability at least $1/2$. By Proposition 1, this length must be at least $\log |G| - \log 2$. Thus, it must hold that

$$\log \frac{|\mathcal{L}|}{(|\mathcal{L}| - N)!} + S + 1 - \frac{\epsilon N}{12T'(T'+1)(\log N + 1)} \geq \log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} - \log \frac{4}{\epsilon}\,.$$

Rearranging, we obtain

$$S \geq \frac{\epsilon N}{O(T^2) \cdot \mathrm{polylog}(N)} - \log \frac{8}{\epsilon}\,.$$

We may assume without loss of generality that $\epsilon \geq 1/N$, since an algorithm that just guesses the discrete log achieves this advantage. Therefore, $\log \frac{8}{\epsilon} = O(\log N)$, and we get

$$(S + O(\log N))T^2 = \widetilde{\Omega}(\epsilon N)\,,$$

which implies that $ST^2 = \widetilde{\Omega}(\epsilon N)$.

□

3.1 Proof of Lemma 4

Recall that a randomized encoding scheme consists of an encoding and a decoding routine, such that both routines take the same string r of random bits as input. The encoding scheme we construct for the purposes of Lemma 4 operates on

labelings σ. That is, the encoding routine takes a labeling $\sigma \in G$ and the random bits r, and constructs a compressed representation of σ. Correspondingly, the decoding routine takes this compressed representation and the *same* random bits r, and reconstructs σ.

While the encoding routine runs, it builds up a table of pairs $(f, \sigma(i)) \in (\mathbb{Z}_N[X] \times \mathcal{L})$. The decoder constructs a similar table during its execution. At any point during the encoding process, the table contains a representation of the information about σ that the encoder has communicated to the decoder up to the current point in the encoding process. The indeterminate X that appears in this table represents a discrete log value $x \in \mathbb{Z}_N$, which the decoder does not know. Once the decoder has enough information to determine x, each of the routines replaces every non-constant polynomial $f(X)$ in the table with its evaluation $f(x)$ at the point x. Subsequently, both routines can introduce a new variable X into the table, which represents a different unknown discrete logarithm in $\mathbb{Z}_N$. Therefore, at any point during the execution, there is at most a single indeterminate X in the table. Finally, when each of the routines completes, the table contains only constant polynomials, and the table fully determines σ.

We stress that the table is not part of the compressed representation of σ, but is part of the internal state of both routines.

Simulating $\mathcal{A}_1$'s Random Tape. Since the algorithm $\mathcal{A}_1$ is randomized, each time the encoder (or decoder) runs the algorithm $\mathcal{A}_1$, it must provide $\mathcal{A}_1$ with a fresh random tape. Both routines take as input a common random bitstring, and the encoder can reserve a substring of it to feed to each invocation of $\mathcal{A}_1$ as that algorithm's random tape. Since $\mathcal{A}_1$ always terminates, the encoder can determine an upper bound on the number of random bits that $\mathcal{A}_1$ will need for a given group size N and can partition the common random string accordingly.

The decoder follows the same process, and the fact that the encoder and decoder take the same random string r as input ensures that $\mathcal{A}_1$ behaves identically during the encoding and decoding processes.

Encoding Routine. The encoding routine, on input σ, uses two parameters $d, R \in \mathbb{Z}^+$, which we will set later, and proceeds as follows:

1. Compute $\mathsf{st}_\sigma \leftarrow \mathcal{A}_0(\sigma(1))$. The encoder can respond to all of the algorithm's oracle queries since the encoder knows all of σ. Write the S-bit output st_σ into the encoding.
2. Encode the image of σ as a subset of $\mathcal{L}$ using $\log \binom{|\mathcal{L}|}{N}$ bits, and append it to the encoding.
3. Initialize the table of pairs to an empty list.
4. Repeat d times:
   (a) Choose the first string in the lexicographical order of the image of σ that does not yet appear in the table. Call this string $\sigma(x)$ and add the pair $(X, \sigma(x))$ to the table.
   (b) Run $\mathcal{A}_1(\mathsf{st}_\sigma, \sigma(x))$ up to R times using independent randomness from the encoder's random string in each run. The encoder answers all of $\mathcal{A}_1$'s oracle queries using its knowledge of σ. If $\mathcal{A}_1$ fails on all R executions,

abort the entire encoding routine. Otherwise, write into the encoding the index $r^* \in [R]$ of the successful execution (using $\log R$ bits).

(c) Write a placeholder of $\log T$ zeros into the encoding. (The routine overwrites these zeros with a meaningful value once this execution of $\mathcal{A}_1$ terminates.)

(d) Rerun $\mathcal{A}_1(\mathsf{st}_\sigma, \sigma(x))$ using the r^*-th random tape. While $\mathcal{A}_1$ is running, it makes a number of queries and then outputs its guess of the discrete log x. The encoding routine processes each of $\mathcal{A}_1$'s queries $(\sigma(i), \sigma(j))$ as follows:

i. If either of the query arguments is outside of the range of σ, reply $\perp$ and continue to the next query.

ii. If either (or both) of the arguments is missing from the table, then this is an "unexpected" query input. Add each such input, together with its discrete log, to the table, and append the discrete-log value i to the encoding, using $\log(N - |\mathsf{Table}|)$ bits.

iii. Otherwise, look up the linear polynomials f_i, f_j representing $\sigma(i), \sigma(j)$ in the table, and compute the linear polynomial $f_i + f_j$ representing the response $\sigma(i + j)$. We then distinguish between three cases:

A. If $(f_i + f_j, \sigma(i + j))$ is already in the table, simply reply with $\sigma(i + j)$.

B. If $\sigma(i + j)$ does not appear in the table, then add $\sigma(i + j)$ to the encoding, using $\log(N - |\mathsf{Table}|)$ bits, and reply with $\sigma(i + j)$.

C. If $\sigma(i + j)$ appears in the table but its discrete log in the table is a polynomial f_k such that $f_k \neq f_i + f_j$, encode the reply to this query as a $(\log |\mathsf{Table}|)$-bit pointer to the table entry $(f_k, \sigma(i+j))$ and add this pointer the encoding. Stop this execution of $\mathcal{A}_1$, and indicate this "early stop" by writing the actual number of queries $t \leq T$ into its placeholder above.

(e) When the execution $\mathcal{A}_1(\mathsf{st}_\sigma, \sigma(x))$ outputs x, evaluate all of the polynomials in the table at the point x.

5. Append the remaining values that do not yet appear in the table to the encoding in lexicographic order.

Decoding Routine. The decoder proceeds analogously to the encoder. A key property of our randomized encoding scheme is that each position in the encoded string corresponds to the same state of the table in both the encoding and the decoding routines. In other words, when the decoding routine reads a certain position in the encoded string, its internal table is identical to the internal table the encoding routine had when it wrote to that position in the encoded string. The table allows the decoder to correctly classify each query to the correct category.

Note that in the case of a collision query (case C above), the decoder can use the collision to recover the value x of the indeterminate X. Specifically, for a query (u, v) where $u, v \in \mathcal{L}$, the decoder reads the reply $w \in \mathcal{L}$ from the encoding string, looks up the polynomials f_u, f_v, and f_w in the table, and solves for X the equation $f_w = f_u + f_v \bmod N$. This equation always has a unique solution, since

N is a prime and f_u, f_v and f_w are linear polynomials in X such that $f_u + f_v$ is not identical to f_w.

The full description of the decoder appears in the full version of this paper [25].

Encoding Length. The encoding contains:

- the advice to the algorithm about the labeling σ (S bits),
- the encoding of the image of σ ($\log \binom{|\mathcal{L}|}{N}$ bits),
- for each of the d invocations of $\mathcal{A}_1$, the index r^* of the random tape on which it succeeded ($d \cdot \log R$ bits in total),
- for the i-th entry added to the table ($0 \le i < N$), if the entry was added
  - as the result of resolving a collision within the table, $\log i$ bits,
  - from the output of $\mathcal{A}_1$, 0 bits,
  - otherwise, $\log(N - i)$ bits,
- a counter indicating the number of queries for which to run each execution ($d \cdot \log T$ bits in total).

Observe that each of the d executions of $\mathcal{A}_1$ saves $\log(N - |\mathsf{Table}|)$ bits compared to the straightforward encoding (either due to $\mathcal{A}_1$ successfully computing the discrete log of its input, or finding a collision), but incurs an additional cost of at most $\log R + \log T + \log |\mathsf{Table}|$ bits. Since each execution of $\mathcal{A}_1$ adds at most $3T + 1$ rows to the table (T replies plus $2T$ unexpected inputs and either one collision or one output of $\mathcal{A}_1$) we have that $|\mathsf{Table}| \le d \cdot (3T + 1)$. Setting $d = \lfloor N/((2RT + 1)(3T + 1)) \rfloor$ guarantees that each of the d executions results in a net profit of

$$\log \frac{N - |\mathsf{Table}|}{RT|\mathsf{Table}|} \ge \log \frac{N - d(3T+1)}{RdT(3T+1)} \ge \log \frac{1 - \frac{1}{2RT+1}}{\frac{RT}{2RT+1}} = \log 2 = 1$$

bit. In this case, the total bitlength of the encoding is at most

$$\begin{aligned} &S + \log \binom{|\mathcal{L}|}{N} + \sum_{i=0}^{N-1} \log(N - i) - d = \log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} + S - d \\ &\le \log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} + S - \frac{N}{(2RT+1)(3T+1)} + 1 \\ &\le \log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} + S - \frac{N}{6RT(T+1)} + 1\,. \end{aligned}$$

We need to choose R large enough to ensure that the encoding routine fails with probability at most $1/2$. If we choose $R = (1 + \log N)/\epsilon$, then the probability that R invocations of $\mathcal{A}_1$ all fail is, by a union bound, at most $(1 - \epsilon)^R \le e^{-\epsilon R} \le 2^{-\epsilon R} \le 2^{-1 - \log N} \le 1/(2N)$. The encoding scheme invokes $\mathcal{A}_1$ on at most N different inputs, so by a union bound, the probability that any invocation fails is at most $1/2$. Overall, the encoding length is at most:

$$\log \frac{|\mathcal{L}|!}{(|\mathcal{L}| - N)!} + S + 1 - \frac{\epsilon N}{6T(T+1)(\log N + 1)} \text{ bits},$$

which completes the proof of Lemma 4. □

3.2 Discrete Logarithms in Short Intervals

When working in groups of large order N, it is common to rely on the hardness of the *short-exponent discrete-log problem*, rather than the standard discrete-log problem [43, 52, 64, 65]. In the usual discrete-log problem, a problem instance is a pair of the form $(g, g^x) \in \mathbb{G}^2$ for $x \xleftarrow{R} \mathbb{Z}_N$. The short-exponent problem is identical, except that x is sampled at random from $\{1, \ldots, W\} \subset \mathbb{Z}_N$, for some interval width parameter $W < N$. Using short exponents speeds up the Diffie-Hellman key-agreement protocol when it is feasible to set the interval width W to be much smaller than the group order N [64]. A variant of Pollard's "Lambda Method" [40, 67] solves the short-exponent discrete-log problem in every group in time $O(W^{1/2})$, so W cannot be too small.

The following corollary of Theorem 2 shows that the short-exponent problem is no easier for generic algorithms with preprocessing than computing a discrete-logarithm in an order-W group.

Corollary 5 (Informal). *Let $\mathcal{A}$ be a generic algorithm with preprocessing that solves the short-exponent discrete-log problem in an interval of width W. If $\mathcal{A}$ uses S bits of group-specific advice, runs in online time T, and succeeds with probability ϵ, then $ST^2 = \widetilde{\Omega}(\epsilon W)$.*

Proof. We claim that the algorithm $\mathcal{A}$ of the corollary solves the standard discrete-log problem with probability $\epsilon' = \epsilon \cdot (W/N)$. The reason is that a standard discrete-log instance g^x for $x \xleftarrow{R} \mathbb{Z}_N$ has a short exponent (i.e., $x \in [W]$) with probability W/N. Algorithm $\mathcal{A}$ solves these short instances with probability ϵ. By Theorem 2, $ST^2 = \widetilde{\Omega}(\epsilon' N) = \widetilde{\Omega}(\epsilon W)$. □

As an application: decryption in the Boneh-Goh-Nissim cryptosystem [18] requires solving a short-exponent discrete-log problem in an interval of width W, for a polynomially large width W. The designers of that system suggest using a size-W table of precomputed discrete logs (i.e., $S = \widetilde{O}(W)$) to enable decryption in constant time. Corollary 5 shows that the best generic decryption algorithm that uses a size-S table requires roughly $\sqrt{W/S}$ time.

3.3 The Computational Diffie-Hellman Problem

A generic algorithm for the computational Diffie-Hellman problem takes as input a triple of labels $(\sigma(1), \sigma(x), \sigma(y))$ and must output the label $\sigma(xy)$. The following theorem demonstrates that in generic groups—even allowing for preprocessing—the computational Diffie-Hellman problem is as hard as computing discrete logarithms.

Theorem 6 (Informal). *Let $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ be a generic algorithm with preprocessing for the computational Diffie-Hellman problem in a group of prime order N. If $\mathcal{A}$ uses S bits of group-specific advice, runs in online time T, and succeeds with probability ϵ, then $ST^2 = \widetilde{\Omega}(\epsilon N)$.*

We present only the proof idea, since the structure of the proof is very similar to that of Theorem 2.

Proof Idea. The primary difference from the proof of Theorem 2, is that, we run $\mathcal{A}_1$ on pairs of labels $(\sigma(x), \sigma(y))$, and a successful run of $\mathcal{A}_1$ produces the CDH value $\sigma(xy)$. Since we run $\mathcal{A}_1$ on two labels at once, the encoder's table now has two formal variables: X and Y.

In this case, whenever the encoder encounters a collision, it gets a single linear relation on X and Y modulo the group order N. Since there are at most N solutions (x_0, y_0) to a linear relation in X and Y over $\mathbb{Z}_N$, the encoder can describe the solution to the decoder using $\log(N - |\mathsf{Table}|)$ bits. The encoder gets some profit, in terms of encoding length, since it will get two discrete logs for the cost of one discrete log and one pointer into the table (of length $\log |\mathsf{Table}|$ bits).

The rest of the proof is as in Theorem 2. □

3.4 Lower Bounds for Families of Groups

The lower bound of Theorem 2 suggests that one way to mitigate the risk of generic preprocessing attacks is to increase the group size. Doubling the size of group elements from $\log N$ to $2 \log N$ recovers the same level of security as if the attacker could not do any preprocessing. The downside of this mitigation strategy is that increasing the group size also increases the cost of each group operation and requires using larger cryptographic keys (e.g., when using the group for Diffie-Hellman key exchange [31]).

One might ask whether it would be possible to defend against preprocessing attacks without having to pay the price of using longer keys. One now-standard method to defend against preprocessing attacks when using a common cryptographic hash function H is to use "salts" [60]. When using salts, each user u of the hash function H chooses a random salt value s_u from a large space of possible salts. User u then uses the salted function $H_u(x) \stackrel{\text{def}}{=} H(s_u, x)$ as her hash function, and the salt value u can be made public. Chung et al. [22] showed that this approach can result in obtaining collision-resistant hashing against preprocessing attacks, and Dodis et al. [32] demonstrated the effectiveness of this approach for a variety of cryptographic primitives.

The analogue to salting in generic groups would be to have a large family of groups (e.g., of elliptic-curve groups) $\{\mathbb{G}_k\}_{k=1}^{K}$ indexed by a key k. Rather than having all users share a single group—as is the case today with NIST P-256—different users and systems could use different groups $\mathbb{G}_k$ sampled from this large family. In particular, pairs of users executing the Diffie-Hellman key-exchange protocol could first jointly sample a group $\mathbb{G}_k$ from this large family and then perform their key exchange in $\mathbb{G}_k$.

We show that using group families in this way effectively defends against generic preprocessing attacks, as long as the family contains a large enough number of groups.

To model group families, we replace the labeling function $\sigma : \mathbb{Z}_N \to \mathcal{L}$ with a keyed family of labeling functions $\sigma_{\text{key}} : [K] \times \mathbb{Z}_N \to \mathcal{L}$. The keyed generic-

group oracle $\mathcal{O}_{\sigma_{\text{key}}}(\cdot,\cdot,\cdot)$ then takes a key k and two labels $\sigma_1, \sigma_2 \in \mathcal{L}$ and returns $\sigma_{\text{key}}(k, x+y)$ if there exist $x, y \in \mathbb{Z}_N$ such that $\sigma_{\text{key}}(k,x) = \sigma_1$ and $\sigma_{\text{key}}(k,y) = \sigma_2$. The oracle returns $\perp$ otherwise. In addition, when fed the pair $(k, \star)$, for a key $k \in [K]$ and a special symbol $\star$, the oracle returns the identity element in the kth group: $\sigma(k,1)$.

The following theorem demonstrates that using a large keyed family of groups effectively defends against generic preprocessing attacks:

Theorem 7. *Let N be a prime. Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a pair of generic algorithms for $[K] \times \mathbb{Z}_N$ on $\mathcal{L}$, such that $\mathcal{A}_0$ outputs an S-bit state, $\mathcal{A}_1$ makes at most T oracle queries, and*

$$\Pr_{\sigma,k,x,\mathcal{A}_1}\left[\mathcal{A}_1^{\mathcal{O}_{\sigma_{\text{key}}}}\left(\mathcal{A}_0^{\mathcal{O}_{\sigma_{\text{key}}}}(), k, \sigma(k,x)\right) = x\right] \geq \epsilon\,,$$

where the probability is taken over the uniformly random choice of the labeling σ_{key}, the key $k \in [K]$, the instance $x \in \mathbb{Z}_N$, and the coins of $\mathcal{A}_1$. Then $ST^2 = \widetilde{\Omega}(\epsilon K N)$.

The proof of Theorem 7 appears in the full version of this paper [25]. The structure of the proof follows that of Theorem 2, except that we need some extra care to handle the fact that an adversary may query the oracle at many different values of k in a single execution.

4 Lower Bound for Computing Many Discrete Logarithms

A natural extension of the standard discrete-log problem is the *multiple*-discrete-log problem [37, 48, 54, 78, 79], in which the adversary's task is to solve M discrete-log problems at once. This problem arises in the setting of multiple-instance security of discrete-log-based cryptosystems. If an adversary has a list of M public keys $(g^{x_1}, \ldots, g^{x_M})$ in some group $\mathbb{G} = \langle g \rangle$ of prime order N, we would like to understand the cost to the adversary of recovering all M secret keys $x_1, \ldots, x_M \in \mathbb{Z}_N$.

Solving the multiple-discrete-log problem cannot be harder than solving M instances of the standard discrete-log problem independently using $\widetilde{O}(M\sqrt{N})$ time overall. One can however do better: generic algorithms due to Kuhn and Struik [54] and Fouque, Joux, and Mavromati [37] solve it in time $\widetilde{O}(\sqrt{MN})$. These algorithms achieve a speed-up over solving M discrete-log instances in sequence by reusing some of the work between instances. Yun [79] showed that in the generic-group model, these algorithms are optimal up to logarithmic factors by proving an $\Omega(\sqrt{NM})$-time lower bound for online-only algorithms, subject to the natural restriction that $M = o(N)$.

Our methods give the more general $ST^2 = \widetilde{\Omega}(\epsilon^{1/M} NM)$ generic lower bound for the M-instance multiple-discrete-log problem with preprocessing. For the special case of algorithms without preprocessing, our bound gives $T = \widetilde{\Omega}(\sqrt{NM})$, which matches the above upper and lower bounds. An additional benefit of our

analysis it that it handles arbitrarily small success probabilities ϵ, whereas Yun's bound applies only to the $\epsilon = \Omega(1)$ case.

Let $\bar{x} = (x_1, \ldots, x_M) \in \mathbb{Z}_N^M$ and, for a labeling $\sigma : \mathbb{Z}_N \to \mathcal{L}$, define the vector $\sigma(\bar{x}) = (\sigma(x_1), \ldots, \sigma(x_M)) \in \mathcal{L}^M$. We restrict ourselves to the case of $M \leq T$, as otherwise the algorithm cannot even afford to perform a group operation on each of its inputs.

Theorem 8. *Let N be a prime. Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a pair of generic algorithms for $\mathbb{Z}_N$ on $\mathcal{L}$ such that $\mathcal{A}_0$ outputs an S-bit advice string, $\mathcal{A}_1$ makes at most T oracle queries,*

$$\Pr_{\sigma,\bar{x},\mathcal{A}_1}\left[\mathcal{A}_1^{\mathcal{O}_\sigma}\left(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)),\, \sigma(\bar{x})\right) = \bar{x}\right] \geq \epsilon,$$

where the probability is taken over the random choice of the labeling σ, a random input vector $\bar{x} \in \mathbb{Z}_N^M$ (for $M \leq T$), and the coins of $\mathcal{A}_1$. Then

$$ST^2/M + T^2 = \widetilde{\Omega}(\epsilon^{1/M} NM).$$

We prove this theorem in the full version of this paper [25].

The proof follows the proof of Theorem 2, except the encoder now runs $\mathcal{A}_1$ on M labels at a time. The encoder and decoder keep a table in M formal variables $(X_1, \ldots, X_M)$, representing the M discrete logs being sought. With every "collision event," we show that the number of formal variables in the table can decrease by one until either (a) $\mathcal{A}_1$ outputs the M discrete logs, or (b) the table has no more formal variables and the encoder halts $\mathcal{A}_1$.

5 The Decisional Diffie-Hellman Problem

The decisional Diffie-Hellman problem [14] (DDH) is to distinguish tuples of the form (g, g^x, g^y, g^{xy}) from tuples of the form (g, g^x, g^y, g^z), for random $x, y, z \in \mathbb{Z}_N$. In this section, we show that every generic distinguisher with preprocessing for the decisional Diffie-Hellman problem that achieves advantage ϵ must satisfy $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$. More formally:

Theorem 9. *Let N be a prime. Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a pair of generic algorithms for $\mathbb{Z}_N$ on $\mathcal{L}$, such that $\mathcal{A}_0$ outputs an S-bit state, $\mathcal{A}_1$ makes at most T oracle queries, and*

$$\begin{aligned}\Big|\Pr\left[\mathcal{A}_1^{\mathcal{O}_\sigma}\left(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x), \sigma(y), \sigma(xy)\right) = 1\right] \\ - \Pr\left[\mathcal{A}_1^{\mathcal{O}_\sigma}\left(\mathcal{A}_0^{\mathcal{O}_\sigma}(\sigma(1)), \sigma(x), \sigma(y), \sigma(z)\right) = 1\right]\Big| \geq \epsilon\,,\end{aligned}$$

where the probabilities are over the choice of the label σ, the values $x, y, z \in \mathbb{Z}_N$, and the randomness of $\mathcal{A}_1$. Then $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$.

The proof of Theorem 9 appears in the full version of this paper [25].

While the proof uses an incompressibility argument, extending the technique of Theorem 2 to give lower bounds for decisional-type problems requires overcoming

additional technical challenges. Consider a DDH distinguisher with preprocessing $(\mathcal{A}_0, \mathcal{A}_1)$ that achieves advantage ϵ. The difficulty with using such an algorithm to build a scheme for compressing σ is that each execution of $\mathcal{A}_1$ only produces a single bit of output. When $\epsilon < 1$, each execution of $\mathcal{A}_1$ produces even less— a fraction of a bit of useful information.

To explain why getting only a single bit of output from $\mathcal{A}_1$ is challenging: the encoder of Theorem 2 derandomized $\mathcal{A}_1$ by writing a pointer $r^* \in [R]$ to a "good" set of random coins for $\mathcal{A}_1$ into the encoding, thus turning a faulty randomized algorithm into a correct deterministic algorithm at the cost of slightly increasing the encoding length. This derandomization technique does not apply immediately here, since the $\log R$-bit value required to point to the "good" set of random coins eliminates any profit in encoding length that we would have gained from the fraction of a bit that $\mathcal{A}_1$ produces as output.

A straightforward amplification strategy—building an algorithm $\mathcal{A}'_1$ that calls $\mathcal{A}_1$ many times and takes the majority output—would circumvent this problem, but would yield an $ST^2 = \widetilde{\Omega}(\epsilon^4 N)$ lower bound that is loose in ϵ.

To achieve a tighter $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$ bound, our strategy is to use $\mathcal{A}_1$ to construct an algorithm $\mathcal{A}_1^{\times B}$ that executes $\mathcal{A}_1$ on a batch of B independent DDH problem instances (one at a time), for some batch size parameter $B \in \mathbb{Z}^+$. The algorithm $\mathcal{A}_1^{\times B}$ now produces B bits of output and succeeds with probability ϵ^B. If we now choose R such that $\log R < B$, we can now apply our prior derandomization technique, since each execution of $\mathcal{A}_1^{\times B}$ will yield some profit in our compression scheme.

Handling collisions in this case involves additional technicalities, since there might (or might not) be a collision in each of the B sub-executions of $\mathcal{A}_1^{\times B}$ and we need to be able to identify which execution encountered a collision without squandering the small profit that $\mathcal{A}_1^{\times B}$ yields.

Putting everything together, we achieve an $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$ lower bound for the DDH problem.

6 Lower Bounds with Limited Preprocessing

Up to this point, we have measured the cost of a discrete-log algorithm with preprocessing by (a) number of bits of preprocessed advice it requires and (b) its online running time. In this section, we explore the preprocessing cost—the time required to compute the advice string— and we prove tight lower bounds on the preprocessing cost of generic discrete-log algorithms.

Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a generic discrete-log algorithm with preprocessing, as defined in Sect. 2. For this section, we allow $\mathcal{A}_0$ to be randomized. We say that $(\mathcal{A}_0, \mathcal{A}_1)$ uses P preprocessing queries and T online queries if $\mathcal{A}_0$ makes P oracle queries and $\mathcal{A}_1$ makes T oracle queries. In this section, we do not put any restriction on the size of the state that $\mathcal{A}_0$ outputs—we are only interested in understanding the relationship between the preprocessing time P and the online time T.

Remark. When $P = \Theta(N)$, there is a trivial discrete-log algorithm with preprocessing $(\mathcal{A}_0, \mathcal{A}_1)$ that uses $T = 0$ online queries and succeeds with constant

probability. In the preprocessing step, $\mathcal{A}_0$ computes a table of $\Theta(N)$ distinct pairs of the form $(i, \sigma(i)) \in \mathbb{Z}_N \times \mathcal{L}$. On receiving a discrete-log instance $\sigma(x)$, the online algorithm $\mathcal{A}_1$ looks to see if $\sigma(x)$ is already stored in its precomputed table and outputs the discrete log x if so. This algorithm succeeds with probability $\epsilon = P/N = \Omega(1)$.

Remark. When $P = o(\sqrt{N})$, we can rule out algorithms that run in online time $T = o(\sqrt{N})$ and succeed with constant probability. To do so, we observe that every generic discrete-log algorithm that uses P preprocessing queries and T online queries can be converted into an algorithm that uses *no* preprocessing queries and $T' = (P + T)$ online queries, such that both algorithms achieve the same success probability.

Shoup's lower bound [71] states that every generic discrete-log algorithm *without preprocessing* that runs in time T' succeeds with probability at most $\epsilon = O(T'^2/N)$. This implies that any algorithm *with preprocessing* P and online time T succeeds with probability at most $\epsilon = O((T + P)^2/N)$.

Put another way: Shoup's result implies a lower bound of $(T + P)^2 = \Omega(\epsilon N)$. So any algorithm that makes only $P = o(\sqrt{N})$ preprocessing queries must use $T = \Omega(\sqrt{N})$ online queries to succeed with constant probability. Thus, an algorithm that uses $o(\sqrt{N})$ preprocessing queries cannot asymptotically outperform an online algorithm.

Given these two remarks, the remaining parameter regime of interest is when $\sqrt{N} < P < N$. We prove:

Theorem 10. *Let $(\mathcal{A}_0, \mathcal{A}_1)$ be a generic discrete-log algorithm with preprocessing for $\mathbb{Z}_N$ on $\mathcal{L}$ that makes at most P preprocessing queries and T online queries. If $x \in \mathbb{Z}_N$ and a labeling function σ are chosen at random, then $\mathcal{A}$ succeeds with probability $\epsilon = O((PT + T^2)/N)$.*

As a corollary, we find that every algorithm that succeeds with probability ϵ must satisfy $PT + T^2 = \Omega(\epsilon N)$. For example, an algorithm that uses $P = O(N^{2/3})$ preprocessing queries must use online time at least $T = \Omega(N^{1/3})$ to succeed with constant probability.

The full proof appears in the full version of this paper [25], and we sketch the proof idea here.

Proof Idea for Theorem 10. We prove the theorem using a pair of probabilistic experiments, following the general strategy of Shoup's now-classic proof technique [71].

In both experiments, the adversary interacts with a challenger, who plays the role of the generic group oracle $\mathcal{O}_\sigma$. The challenger defines the labeling function $\sigma(\cdot)$ lazily in response to the adversary's queries. Both experiments follow similar steps:

1. The challenger sends a label $s_1 \in \mathcal{L}$, representing $\sigma(1)$, to the adversary.
2. The adversary makes P preprocessing group-oracle queries to the challenger.

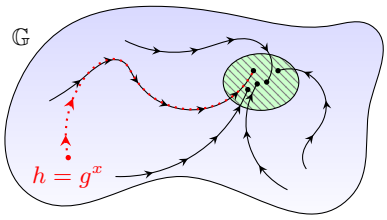

Fig. 1. The discrete-log algorithm with preprocessing of Sect. 7.1 uses a random function F to define a walk on the elements of $\mathbb{G}$. The preprocessed advice consists of the discrete logs of S points that lie at the end of length-$\Theta(T)$ disjoint paths on the walk. In the online phase, the algorithm walks from the input point until hitting a stored endpoint, which occurs with good probability.

3. The challenger sends the discrete-log instance $s_x \in \mathcal{L}$, representing $\sigma(x)$, to the adversary.
4. The adversary makes T online queries and outputs a guess x' of x.

The difference between the two experiments is in how the challenger defines the discrete log of the instance $s_x \in \mathcal{L}$.

In Experiment 0, the challenger chooses the discrete log $x \in \mathbb{Z}_N$ of s_x *before* the adversary makes any online queries. The challenger in Experiment 0 is thus a faithful (or honest) oracle.

In Experiment 1, the challenger chooses the discrete log x of σ_x *after* the adversary has made all of its online queries. In this latter case, the challenger is essentially "cheating" the adversary, since all of the challenger's query responses are independent of x and the adversary cannot recover x with probability better than random guessing. To complete the argument, we show that unless the adversary makes many queries, it can only rarely distinguish between the two experiments.

A detailed description of the experiments and their analysis appears in the full version of this paper [25]. □

The Lower Bound is Tight. From Theorem 2, we know that a discrete-log algorithm that succeeds with constant probability must use advice S and online time T such that $ST^2 = \widetilde{\Omega}(N)$. From Theorem 10, we know that any such algorithm must also use preprocessing P such that $PT + T^2 = \Omega(N)$. The best tradeoff we could hope for, ignoring the constants and logarithmic factors, is $PT + T^2 = ST^2$, or $P = ST$. Indeed, the known upper bound with preprocessing (see Sect. 7.1) matches this lower bound, disregarding low-order terms.

7 Preprocessing Attacks on Discrete-Log Problems

In this section, we recall the known generic discrete-log algorithm with preprocessing and we introduce two new generic attacks with preprocessing. Specifically,

we show an attack on the multiple-discrete-log problem that matches the lower bound of Theorem 8, and we show an attack on certain decisional problems in groups that matches the lower bound of Theorem 9.

These attacks are all generic, so they apply to every group, including popular elliptic-curve groups. Our preprocessing attacks are *not* polynomial-time attacks—indeed our lower bounds rule out such attacks—but they yield better-than-known exponential-time attacks on these problems.

The analysis of the algorithms in these sections rely on the attacker having access to a random function (i.e., a random oracle [10]), which the attacker could instantiate with a standard cryptographic hash function, such as SHA-256. Removing the attacks' reliance on a truly random function remains a useful task for future work.

7.1 The Existing Discrete-Log Algorithm with Preprocessing

For the reader's reference, we describe a variation of the discrete-log algorithm with preprocessing, introduced by Mihalcik [59] and Bernstein and Lange [13], with a slightly more detailed analysis. This discrete-log algorithm shows that the lower bound of Theorem 2 is tight. Our algorithms for the multiple-discrete-log problem (Sect. 7.2) and for distinguishing pseudo-random generators (Sect. 7.3) use ideas from this algorithm.

The algorithm computes discrete logs in a group $\mathbb{G}$ of prime order N with generator g. The algorithm takes as input parameters $S, T \in \mathbb{Z}^+$ such that $ST^2 \leq N$. The algorithm uses $\widetilde{O}(S)$ bits of precomputed advice about the group $\mathbb{G}$, uses $\widetilde{O}(T)$ group operations in the online phase, and succeeds with probability $\epsilon = \Omega(ST^2/N)$.

Let $F : \mathbb{G} \to \mathbb{Z}_N$ be a random function, which we can instantiate in practice using a standard hash function. We use the function F to define a walk on the elements of $\mathbb{G}$. Given a point $h \in \mathbb{G}$, the walk computes $\alpha \leftarrow F(h)$ and moves to the point $g^\alpha h \in \mathbb{G}$.

Given these preliminaries, the algorithm works as follows:

- *Preprocessing phase.* Repeat S times: pick $r \xleftarrow{R} \mathbb{Z}_N$ and, starting at $g^r \in \mathbb{G}$, take the walk defined by F for $T/2$ steps. Store the endpoint of the walk $g^{r'}$ and its discrete log r' in a table: $(r', g^{r'})$.

  At the end of the preprocessing phase, the algorithm stores this table of S group elements along with their discrete logs, using $O(S \log N)$ bits.
- *Online phase.* Given a discrete-log instance $h = g^x$, the algorithm takes T steps along the random walk defined by F, starting from the point h (see Fig. 1). If the walk hits one of the S points stored in the precomputed table, this collision yields a linear relation on x in the exponent: $g^{r'} = g^{x+\alpha_1+\cdots+\alpha_k} \in \mathbb{G}$. Solving this linear relation for $x \in \mathbb{Z}_N$ reveals the desired discrete log.

The algorithm uses $\widetilde{O}(S)$ bits of group-specific advice and runs in online time $\widetilde{O}(T)$. The remaining task is to analyze its success probability.

We first claim that, with good probability, the S walks in the preprocessing phase touch at least $ST/4$ distinct points. To this end, observe that for every

walk in the preprocessing phase, the probability that it touches $T/2$ new points is at least $(1 - ST/(2N))^{T/2} \geq 1 - ST^2/(4N)$, by Bernoulli's inequality. Since $ST^2 \leq N$, we have that $1 - ST^2/(4N) \geq 1 - 1/4 = 3/4$. Therefore, in expectation, each walk touches at least $3T/8$ new points and by linearity of expectation, the overall expected number of touched points is at least $3ST/8$. The number of touched points is at most $ST/2$ and is at least $3ST/8$, in expectation. We can apply Markov's inequality to an auxiliary random variable to conclude that the number of touched points is greater than $ST/4$ with probability at least $1/2$.

Next, observe that if at any of its first $T/2$ steps, the online walk hits any of the points touched by one of the preprocessed walks, in the remaining $T/2$ steps it will hit the stored endpoint of that preprocessed walk. It will then successfully compute the discrete log. Moreover, as long as the online walk does not hit any of these points, its steps are independent random points in $\mathbb{G}$. If the number points touched during preprocessing is at least $ST/4$, then the online walk succeeds with probability at least $1-(1-(ST/(4N))^{T/2} \geq 1-\exp(-ST^2/(8N)) \geq ST^2/(16N)$. Overall, the probability of success ϵ is at least $1/2 \cdot ST^2/(16N) = \Omega(ST^2/N)$.

7.2 Multiple Discrete Logarithms with Preprocessing

We now demonstrate that a similar technique allows solving the multiple-discrete-log problem more quickly using preprocessing. The algorithm is a modification to the attack of Fouque et al. [37] to allow for precomputation, in the spirit of the algorithm of Sect. 7.1.

This upper bound matches the lower bound of Theorem 8 for a constant ϵ, up to logarithmic factors, which shows that the lower bound is tight for constant ϵ. To recall, an instance of the multiple-discrete-log problem is a vector $(g^{x_1}, \ldots, g^{x_M})$ for random $x_i \in \mathbb{Z}_N$. The solution is the vector $(x_1, \ldots, x_M)$. Then we have the following theorem:

Theorem 11. *There exists a generic algorithm with preprocessing for the M-instance multiple-discrete-log problem in a group of prime order N that makes use of a random function, uses $\widetilde{O}(S)$ bits of group-specific advice, runs in time $\widetilde{O}(T)$, succeeds with constant probability, and satisfies $ST^2/M + T^2 = O(MN)$.*

We prove the theorem in the full version of this paper [25].

7.3 Distinguishers with Preprocessing

In this section, we give a new distinguishing algorithm for certain decisional problems in groups.

For concreteness, we first demonstrate how to use preprocessing to attack the *square decisional Diffie-Hellman problem* (sqDDH) [50], which is the problem of distinguishing tuples of the form (g, g^x, g^y) from tuples of the form $(g, g^x, g^{(x^2)})$ for random $x, y \in \mathbb{Z}_N$. In groups for which DDH is hard, the best known attack against this assumption requires solving the discrete-log problem. Later on, we show how to generalize the attack to a larger family of natural decisional assumptions in groups.

Definition 12. We say that an oracle algorithm $\mathcal{A}^{\mathcal{O}}$ has *advantage* ϵ at distinguishing distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ if $\left|\Pr[\mathcal{A}^{\mathcal{O}}(d_1) = 1] - \Pr[\mathcal{A}^{\mathcal{O}}(d_2) = 1]\right| = \epsilon$, where the probability is over the randomness of the oracle and samples $d_1 \sim \mathcal{D}_1$ and $d_2 \sim \mathcal{D}_2$.

Theorem 13. *There is a sqDDH distinguisher with preprocessing that makes use of a random function, uses $\widetilde{O}(S)$ bits of group-specific advice, runs in time $\widetilde{O}(T)$, and achieves distinguishing advantage ϵ whenever $ST^2 = \Omega(\epsilon^2 N)$.*

Remark. A simple sqDDH distinguisher takes as input a sample $(h_0, h_1) \in \mathbb{G}^2$, computes the discrete logarithm $x = \log_g(h_0)$ of the first group element and checks whether $h_1 = g^{(x^2)} \in \mathbb{G}$. Theorem 2 indicates that such a distinguisher using advice S and time T and achieving advantage ϵ must satisfy $ST^2 = \widetilde{\Omega}(\epsilon N)$. So, this attack allows the parameter setting $S = T = 1/\epsilon = N^{1/4}$. In contrast, the distinguisher of Theorem 13 allows the better running time and advice complexity roughly $S = T = 1/\epsilon = N^{1/5}$.

Remark. To see the cryptographic significance of Theorem 13, consider the pseudo-random generator $P(x) \stackrel{\text{def}}{=} (g^x, g^{(x^2)})$ that maps $\mathbb{Z}_N$ to $\mathbb{G}^2$. Theorem 13 shows that, for generic algorithms with preprocessing, it is significantly easier to distinguish this PRG from random than it is to compute discrete logs.

Proof Sketch of Theorem 13. The attack that proves the theorem combines two technical tools. The first tool is a general method for using preprocessing to distinguish PRG outputs from random, which we adopt from Bernstein and Lange [13]. (De et al. [28] rigorously analyze a more nuanced PRG distinguisher with preprocessing.) The second tool, adopted from the attack of Sect. 7.1, is the idea of taking a walk on the elements of the group, and applying the PRG distinguisher only to the set of points that lie at the end of long walks.

The attack works because a walk that begins at a point of the form $(g^x, g^{(x^2)})$ is likely to hit one of the precomputed endpoints quickly and applying the PRG distinguisher yields an ϵ-biased output value. In contrast, an attack that begins at a point of the form (g^x, g^y) will never hit a precomputed point and applying the distinguisher yields a relatively unbiased output.

The algorithm (illustrated in Fig. 2) takes as input parameters $S, T \in \mathbb{Z}^+$.

As in the attack of Sect. 7.1, we use a random function to define a walk on a graph. In this case, the vertices of the graph are *pairs* of group elements—so every vertex is an element of $\mathbb{G}^2$. We also define the subset of vertices $\mathcal{Y} = \{(g^x, g^{(x^2)}) \mid x \in \mathbb{Z}_N\} \subset \mathbb{G}^2$ that correspond to "yes" instances of the sqDDH problem. The subset $\mathcal{Y}$ is very small relative to the set of all vertices $\mathbb{G}^2$, since $|\mathbb{G}^2| = N^2$, while $|\mathcal{Y}| = N$.

To define the walk on the vertices of this graph, we use a random function F that maps $\mathbb{G}^2 \to \mathbb{Z}_N$. Given a point $(h_0, h_1) \in \mathbb{G}^2$, the walk computes $\alpha \leftarrow F(h_0, h_1)$ and moves to the point $(h_0^\alpha, h_1^{(\alpha^2)}) \in \mathbb{G}^2$. Observe that if the walk starts in $\mathcal{Y}$ (i.e., at a "yes" point), the walk remains inside of $\mathcal{Y}$. If the walk starts at a point outside of $\mathcal{Y}$, the walk remains outside of $\mathcal{Y}$.

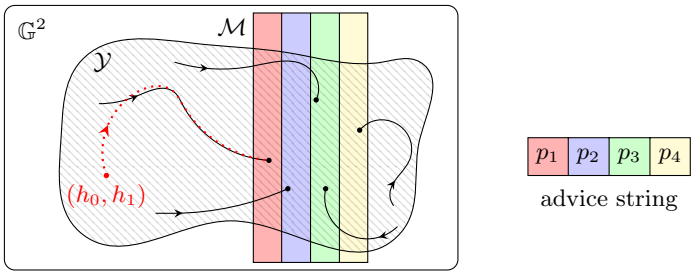

Fig. 2. The preprocessing phase of the sqDDH distinguisher takes walks on the elements of $\mathcal{Y} \subset \mathbb{G}^2$. Each walk terminates upon hitting the set of *marked points* $\mathcal{M}$, which we further partition into S "colors". The advice consists of a string p_c for each of the colors, such that the sum $\sum H(p_c, m)$ is maximized over all the endpoints of color c. In the online phase (in red), the algorithm walks from the input point until hitting a marked point.

Out of the N^2 total vertices in the graph, we choose a set of distinguished or "marked" points $\mathcal{M}$, by marking each point independently at random with probability $1/T$. (In practice, we can choose the set of marked points using a hash function.) To each point in $\mathcal{M}$, we assign one of S different "colors," again using a hash function. So there are roughly $N^2/(ST)$ points each with color $1, 2, \ldots, S$.

Given these preliminaries, the algorithm works as follows:

– *Preprocessing phase.* Choose $N/3T^2$ random points in $\mathcal{Y}$. From each of these points, take $2T$ steps of the walk on $\mathbb{G}^2$ that F defines. Halt the walk upon reaching a marked point $m \in \mathcal{M}$. If the walk hits a marked point, store the marked point along with its color c in a table.

  Group the endpoints of the walks by color. For each of the colors $c \in [S]$, find the prefix string $p_c \in \{0,1\}^{\log N}$ that maximizes the sum $\sum H(p_c, m)$, where $H : \{0,1\}^{\log N} \times \mathbb{G}^2 \to \{0,1\}$ is a random function and the sum is taken over the stored marked points m of color c.

  Store the prefix strings $(p_1, \ldots, p_S)$ as the distinguisher's advice.

– *Online phase.* Given a sqDDH challenge $(h_0, h_1) \in \mathbb{G}^2$ as input, perform at most $10T$ steps of the walk on $\mathbb{G}^2$ that the function F defines. As soon as the walk hits a marked point $m \in \mathcal{M}$ of color c, return the value $H(p_c, m)$ as output. If the walk never hits a marked point, output "0" or "1" with probability $1/2$ each.

The distinguisher uses $\widetilde{O}(S)$ bits of group-specific advice and runs in time $\widetilde{O}(T)$ as desired. So all we must argue is that the algorithm achieves distinguishing advantage $\epsilon = \Omega(\sqrt{ST^2/N})$. We argue this last step in the full version of this paper [25]. □

Attacking More-General Problems. The distinguishing attack of Theorem 13 applies to a general class of decisional problems in cyclic groups. Let $(f_1, \ldots, f_\ell)$

be k-variate polynomials and let $\bar{x} = (x_1, \ldots, x_k) \in \mathbb{Z}_N^k$. Then we can define the problem of distinguishing tuples of the form

$$(g^{x_1}, \ldots, g^{x_k}, g^{f_1(\bar{x})}, \ldots, g^{f_\ell(\bar{x})}) \quad \text{from} \quad (g^{x_1}, \ldots, g^{x_k}, g^{r_1}, \ldots, g^{r_\ell}),$$

for uniformly random $x_1, \ldots, x_k, r_1, \ldots, r_\ell \in \mathbb{Z}_N$.

The attack of Theorem 13 applies whenever there exists an index i, a linear function $L : \mathbb{G}^{k+\ell} \to \mathbb{G}$, and a constant $c > 1$ such that $L(\bar{x}, f_1(\bar{x}), \ldots, f_\ell(\bar{x})) = x_i^c$. To apply the attack, first apply $L(\cdot)$ "in the exponent" to the challenge to get a pair $(g^{x_i}, g^{x_i^c}) \in \mathbb{G}^2$ and then run the distinguisher on this pair of elements.

As an example, this attack can distinguish tuples of the form $(g^{x_1}, g^{x_2}, g^{(x_1^2)}, g^{x_1 x_2}, g^{(x_2^2)})$ from random. The attack uses $i = 1$, $L(z_1, z_2, z_3, z_4, z_5) = z_3$, and $c = 2$. Note that this assumption is very closely related to the standard DDH assumption, except that the challenge tuple includes the extra elements $g^{(x_1^2)}$ and $g^{(x_2^2)}$.

Remark. Somewhat surprising is that the distinguishing attack of Theorem 13 *does not* translate to an equivalently strong attack for the DDH problem. The immediate techical obstacle for this is the fact that the distinguishing advantage of the generic PRG distinguisher reduces as the size of the seed space of the PRG grows. That space is of size N in the sqDDH problem, but of size N^2 in the DDH case, which results in a weaker distinguisher.

8 Conclusion

We studied the limits of generic group algorithms with preprocessing for the discrete-logarithm problem and related computational tasks.

In almost all cases, our lower bounds match the best known attacks up to logarithmic factors in group order. The one exception is our lower bound for the decisional Diffie-Hellman problem, in which our lower bound is $ST^2 = \widetilde{\Omega}(\epsilon^2 N)$, but the attack requires computing a discrete logarithm with $ST^2 = \widetilde{O}(\epsilon N)$. When the success probability ϵ is constant, these bounds match. For intermediate values of ϵ, such as $\epsilon = N^{-1/4}$, it is not clear which bound is correct.

One useful task for future work would be to generalize our lower bounds to more complex assumptions, such as Diffie-Hellman assumptions on pairing-equipped groups [17], q-type assumptions [15], or the "uber" assumptions [16, 19].

In addition, our upper bounds of Sect. 7 make use of a public random function. Making the attacks fully constructive by removing this heuristic, in the spirit of Fiat and Naor [35] and De et al. [28], would be valuable as well.

Acknowledgements. We would like to thank Dan Boneh for encouraging us to undertake this project and for his advice along the way. We thank Omer Reingold, David Wu, and Benedikt Bünz for fruitful discussions during the early stages of this work. Saba Eskandarian, Steven Galbraith, Sam Kim, and Florian Tramèr gave suggestions that improved the presentation. This work was supported by NSF, DARPA, the Stanford Cyber Initiative, the Simons foundation, a grant from ONR, and an NDSEG Fellowship.

References

1. Abadi, M., Feigenbaum, J., Kilian, J.: On hiding information from an oracle. In: STOC (1987), https://doi.org/10.1145/28395.28417
2. Abusalah, H., Alwen, J., Cohen, B., Khilko, D., Pietrzak, K., Reyzin, L.: Beyond Hellman's time-memory trade-offs with applications to proofs of space. In: ASIACRYPT (2017), https://doi.org/10.1007/978-3-319-70697-9_13
3. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., et al.: Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: CCS (2015), https://doi.org/10.1145/2810103.2813707
4. Aggarwal, D., Maurer, U.: Breaking RSA generically is equivalent to factoring. In: EUROCRYPT (2009), https://doi.org/10.1007/978-3-642-01001-9_2
5. Arora, S., Barak, B.: Computational complexity: a modern approach. Cambridge University Press (2009)
6. Babai, L., Szemeredi, E.: On the complexity of matrix group problems I. In: FOCS (1984), https://doi.org/10.1109/sfcs.1984.715919
7. Bărbulescu, R.: Improvements on the Discrete Logarithm Problem in GF(p). Master's thesis, Ècole Normale Supérieure de Lyon (2011), https://hal.inria.fr/inria-00588713
8. Barbulescu, R., Gaudry, P., Joux, A., Thomé, E.: A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In: EUROCRYPT (2014), https://doi.org/10.1007/978-3-642-55220-5_1
9. Barkan, E., Biham, E., Shamir, A.: Rigorous bounds on cryptanalytic time/memory tradeoffs. In: CRYPTO (2006), https://doi.org/10.1007/11818175_1
10. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: CCS (1993), https://doi.org/10.1145/168588.168596
11. Bernstein, D., Lange, T.: Two grumpy giants and a baby. The Open Book Series 1(1), 87–111 (2013), https://doi.org/10.2140/obs.2013.1.87
12. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: PKC (2006), https://doi.org/10.1007/11745853_14
13. Bernstein, D.J., Lange, T.: Non-uniform cracks in the concrete: the power of free precomputation. In: ASIACRYPT (2013), https://doi.org/10.1007/978-3-642-42045-0_17
14. Boneh, D.: The decision Diffie-Hellman problem. In: ANTS (1998), https://doi.org/10.1007/BFb0054851
15. Boneh, D., Boyen, X.: Short signatures without random oracles. In: EUROCRYPT (2004), https://doi.org/10.1007/978-3-540-24676-3_4
16. Boneh, D., Boyen, X., Goh, E.J.: Hierarchical identity based encryption with constant size ciphertext. In: EUROCRYPT (2005), https://doi.org/10.1007/11426639_26
17. Boneh, D., Franklin, M.K.: Identity-based encryption from the weil pairing. In: CRYPTO (2001), https://doi.org/10.1007/3-540-44647-8_13
18. Boneh, D., Goh, E.J., Nissim, K.: Evaluating 2-DNF formulas on ciphertexts. In: TCC (2005), https://doi.org/10.1007/978-3-540-30576-7_18
19. Boyen, X.: The uber-assumption family. In: Pairing-based Cryptography (2008), https://doi.org/10.1007/978-3-540-85538-5_3
20. Brown, D.: On the provable security of ECDSA. In: Advances in Elliptic Curve Cryptography, pp. 21–40. Cambridge University Press (2005), https://doi.org/10.1017/cbo9780511546570.004

21. Brown, D.R.L.: Generic groups, collision resistance, and ECDSA. Designs, Codes and Cryptography 35(1), 119–152 (2005), https://doi.org/10.1007/s10623-003-6154-z
22. Chung, K.M., Lin, H., Mahmoody, M., Pass, R.: On the power of nonuniformity in proofs of security. In: ITCS (2013), http://doi.acm.org/10.1145/2422436.2422480
23. Coppersmith, D.: Modifications to the number field sieve. Journal of Cryptology 6(3), 169–180 (1993)
24. Coretti, S., Dodis, Y., Guo, S., Steinberger, J.: Random oracles and non-uniformity. Cryptology ePrint Archive, Report 2017/937 (2017), https://eprint.iacr.org/2017/937
25. Corrigan-Gibbs, H., Kogan, D.: The discrete-logarithm problem with preprocessing. Cryptology ePrint Archive, Report 2017/1113 (2017), https://eprint.iacr.org/2017/1113
26. Cramer, R., Shoup, V.: A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In: CRYPTO (1998), https://doi.org/10.1007/bfb0055717
27. Damgård, I., Koprowski, M.: Generic lower bounds for root extraction and signature schemes in general groups. In: EUROCRYPT (2002), https://doi.org/10.1007/3-540-46035-7_17
28. De, A., Trevisan, L., Tulsiani, M.: Time space tradeoffs for attacks against one-way functions and PRGs. In: CRYPTO (2010), https://doi.org/10.1007/978-3-642-14623-7_35
29. Dent, A.W.: Adapting the weaknesses of the random oracle model to the generic group model. In: ASIACRYPT (2002), https://doi.org/10.1007/3-540-36178-2_6
30. Dent, A.W.: The hardness of the DHK problem in the generic group model. Cryptology ePrint Archive, Report 2006/156 (2006), https://eprint.iacr.org/2006/156
31. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Trans. Inf. Theory 22(6), 644–654 (1976), https://doi.org/10.1109/tit.1976.1055638
32. Dodis, Y., Guo, S., Katz, J.: Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In: EUROCRYPT (2017), https://doi.org/10.1007/978-3-319-56614-6_16
33. Dodis, Y., Haitner, I., Tentes, A.: On the instantiability of hash-and-sign RSA signatures. In: TCC (2012), https://doi.org/10.1007/978-3-642-28914-9_7
34. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In: CRYPTO (1984), https://doi.org/10.1007/3-540-39568-7_2
35. Fiat, A., Naor, M.: Rigorous time/space tradeoffs for inverting functions. In: STOC (1991), http://doi.acm.org/10.1145/103418.103473
36. Fischlin, M.: A note on security proofs in the generic model. In: ASIACRYPT (2000), https://doi.org/10.1007/3-540-44448-3_35
37. Fouque, P.A., Joux, A., Mavromati, C.: Multi-user collisions: Applications to Discrete Logarithm, Even-Mansour and PRINCE. In: ASIACRYPT (2013), https://doi.org/10.1007/978-3-662-45611-8_22
38. Freeman, D., Scott, M., Teske, E.: A taxonomy of pairing-friendly elliptic curves. Journal of Cryptology 23(2), 224–280 (2010), https://doi.org/10.1007/s00145-009-9048-z
39. Galbraith, S.D., Gaudry, P.: Recent progress on the elliptic curve discrete logarithm problem. Designs, Codes and Cryptography 78(1), 51–72 (Jan 2016), https://doi.org/10.1007/s10623-015-0146-7
40. Galbraith, S.D., Ruprai, R.S.: Using equivalence classes to accelerate solving the discrete logarithm problem in a short interval. In: PKC (2010), https://doi.org/10.1007/978-3-642-13013-7_22

41. Gaudry, P., Hess, F., Smart, N.P.: Constructive and destructive facets of Weil descent on elliptic curves. Journal of Cryptology 15(1), 19–46 (2002), https://doi.org/10.1007/s00145-001-0011-x
42. Gennaro, R., Trevisan, L.: Lower bounds on the efficiency of generic cryptographic constructions. In: FOCS (2000), https://doi.org/10.1109/SFCS.2000.892119
43. Gennaro, R.: An improved pseudo-random generator based on discrete log. In: CRYPTO (2000), https://doi.org/10.1007/3-540-44598-6_29
44. Gordon, D.M.: Discrete logarithms in GF(P) using the number field sieve. SIAM Journal on Discrete Mathematics 6(1), 124–138 (1993), https://doi.org/10.1137/0406010
45. Hellman, M.: A cryptanalytic time-memory trade-off. IEEE Trans. Inf. Theor. 26(4), 401–406 (1980), http://dx.doi.org/10.1109/TIT.1980.1056220
46. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security 1(1), 36–63 (2001), https://doi.org/10.1007/s102070100002
47. Joux, A., Odlyzko, A., Pierrot, C.: The past, evolving present, and future of the discrete logarithm. In: Open Problems in Mathematics and Computational Science, pp. 5–36. Springer (2014), https://doi.org/10.1007/978-3-319-10683-0_2
48. Kim, T.: Multiple discrete logarithm problems with auxiliary inputs. In: ASIACRYPT (2015), https://doi.org/10.1007/978-3-662-48797-6_8
49. Koblitz, N., Menezes, A.: Another look at generic groups. Adv. Math. Commun. 1(1), 13–28 (2007), https://doi.org/10.3934/amc.2007.1.13
50. Koblitz, N., Menezes, A.: Intractable problems in cryptography. In: Conference on Finite Fields and Their Applications (2010), https://doi.org/10.1090/conm/518/10212
51. Koblitz, N., Menezes, A., Vanstone, S.: The state of elliptic curve cryptography. Des. Codes Cryptography 19(2-3), 173–193 (2000), http://dx.doi.org/10.1023/A:1008354106356
52. Koshiba, T., Kurosawa, K.: Short exponent Diffie-Hellman problems. In: PKC (2004), https://doi.org/10.1007/978-3-540-24632-9_13
53. Kravitz, D.W.: Digital signature algorithm (1993), US Patent 5,231,668
54. Kuhn, F., Struik, R.: Random walks revisited: Extensions of Pollard's rho algorithm for computing multiple discrete logarithms. In: International Workshop on Selected Areas in Cryptography (2001), https://doi.org/10.1007/3-540-45537-x_17
55. Leander, G., Rupp, A.: On the equivalence of RSA and factoring regarding generic ring algorithms. In: ASIACRYPT (2006), https://doi.org/10.1007/11935230_16
56. Matyukhin, D.V.: On asymptotic complexity of computing discrete logarithms over GF(p). Discrete Mathematics and Applications 13(1), 27–50 (2003), https://doi.org/10.1515/156939203321669546
57. Maurer, U.: Abstract models of computation in cryptography. In: Cryptography and Coding (2005), https://doi.org/10.1007/11586821_1
58. Menezes, A.J., Okamoto, T., Vanstone, S.A.: Reducing elliptic curve logarithms to logarithms in a finite field. IEEE Transactions on information Theory 39(5), 1639–1646 (1993), https://doi.org/10.1109/18.259647
59. Mihalcik, J.P.: An analysis of algorithms for solving discrete logarithms in fixed groups. Master's thesis, Naval Postgraduate School (2010), https://calhoun.nps.edu/bitstream/handle/10945/5395/10Mar_Mihalcik.pdf
60. Morris, R., Thompson, K.: Password security: a case history. Communications of the ACM 22(11), 594–597 (1979), https://doi.org/10.1145/359168.359172
61. Nechaev, V.I.: Complexity of a determinate algorithm for the discrete logarithm. Mathematical Notes 55(2), 165–172 (1994), https://doi.org/10.1007/bf02113297

62. Odlyzko, A.: Discrete logarithms: The past and the future. Designs, Codes and Cryptography 19(2), 129–145 (2000), https://doi.org/10.1023/A:1008350005447
63. Oechslin, P.: Making a faster cryptanalytic time-memory trade-off. In: CRYPTO (2003), https://doi.org/10.1007/978-3-540-45146-4_36
64. van Oorschot, P.C., Wiener, M.J.: On Diffie-Hellman key agreement with short exponents. In: EUROCRYPT (1996), https://doi.org/10.1007/3-540-68339-9_29
65. Patel, S., Sundaram, G.S.: An efficient discrete log pseudo random generator. In: CRYPTO (1998), https://doi.org/10.1007/bfb0055737
66. Pohlig, S., Hellman, M.: An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (corresp.). IEEE Trans. Inf. Theory 24(1), 106–110 (1978), https://doi.org/10.1109/tit.1978.1055817
67. Pollard, J.M.: Monte Carlo methods for index computation (mod p). Mathematics of computation 32(143), 918–924 (1978), https://doi.org/10.2307/2006496
68. Schnorr, C.P.: Efficient identification and signatures for smart cards. In: CRYPTO (1989), https://doi.org/10.1007/0-387-34805-0_22
69. Schnorr, C.P., Jakobsson, M.: Security of signed ElGamal encryption. In: ASIACRYPT (2000), https://doi.org/10.1007/3-540-44448-3_7
70. Shanks, D.: Class number, a theory of factorization, and genera (1971), https://doi.org/10.1090/pspum/020/0316385
71. Shoup, V.: Lower bounds for discrete logarithms and related problems. In: EUROCRYPT (1997), https://doi.org/10.1007/3-540-69053-0_18
72. Smart, N.P.: The exact security of ECIES in the generic group model. In: Cryptography and Coding (2001), https://doi.org/10.1007/3-540-45325-3_8
73. Smart, N.P.: The discrete logarithm problem on elliptic curves of trace one. Journal of Cryptology 12(3), 193–196 (1999), https://doi.org/10.1007/s001459900052
74. Unruh, D.: Random oracles and auxiliary input. In: CRYPTO (2007), https://doi.org/10.1007/978-3-540-74143-5_12
75. Wang, P., Zhang, F.: Computing elliptic curve discrete logarithms with the negation map. Information Sciences 195, 277–286 (2012), https://doi.org/10.1016/j.ins.2012.01.044
76. Wee, H.: On obfuscating point functions. In: STOC (2005), http://doi.acm.org/10.1145/1060590.1060669
77. Yao, A.C.C.: Coherent functions and program checkers. In: STOC (1990), http://doi.acm.org/10.1145/100216.100226
78. Ying, J.H., Kunihiro, N.: Bounds in various generalized settings of the discrete logarithm problem. In: ACNS (2017), https://doi.org/10.1007/978-3-319-61204-1_25
79. Yun, A.: Generic hardness of the multiple discrete logarithm problem. In: EUROCRYPT (2015), https://doi.org/10.1007/978-3-662-46803-6_27
80. Zhang, F., Wang, P., Galbraith, S.: Computing elliptic curve discrete logarithms with improved baby-step giant-step algorithm. Adv. Math. Commun. 11(3), 453–469 (2017), https://doi.org/10.3934/amc.2017038