Breaking RSA Generically is Equivalent to Factoring, *with Preprocessing*

Dana Dachman-Soled[1] *, Julian Loss[2] **, Adam O'Neill[3] ***, and Nikki Sigurdson[3]

[1] University of Maryland
danadach@umd.edu
[2] CISPA Helmholtz Center for Information Security
loss@cispa.de
[3] Manning College of Information and Computer Science, University of Massachusetts Amherst
amoneill@gmail.com, nsigurdson@umass.edu

Abstract. We investigate the relationship between the classical RSA and factoring problems when *preprocessing* is considered. In such a model, adversaries can use an unbounded amount of precomputation to produce an "advice" string to then use during the online phase, when a problem instance becomes known. Previous work (*e.g.*, [Bernstein, Lange ASIACRYPT '13]) has shown that preprocessing attacks significantly improve the runtime of the best-known factoring algorithms. Due to these improvements, we ask whether the relationship between factoring and RSA fundamentally changes when preprocessing is allowed. Specifically, we investigate whether there is a superpolynomial gap between the runtime of the best attack on RSA with preprocessing and on factoring with preprocessing.

Our main result rules this out with respect to algorithms in a natural adaptation of the generic ring model to the preprocessing setting. In particular, in this setting we show the existence of a factoring algorithm (albeit in the random oracle model) with polynomially related parameters, for any setting of RSA parameters.

1 Introduction

1.1 Motivation and Main Results.

Background. Use of the RSA function [26] $f_{N,e}(x) = x^e \bmod N$ where $N = pq$ is ubiquitous in practice, and attacks against it have been the subject of intensive study, see *e.g.* [4]. A key question about its security is its relationship

* Supported in part by NSF grants #CNS-1933033, #CNS-1453045 (CAREER), and by financial assistance awards 70NANB15H328 and 70NANB19H126 from the U.S. Department of Commerce, National Institute of Standards and Technology.

** Part of this work was done while the author was a postdoc at the University of Maryland and Carnegie Mellon University.

*** Supported in part by a grant from Cisco.

to factoring N. While it is trivial to see that factoring N allows one to invert RSA, the converse is a major open problem. To make progress on this question, researchers have studied it in restricted (aka. idealized) models of computation. To our knowledge, this approach was initiated by Boneh and Venkatesan [5], who showed that a reduction from factoring to low-exponent RSA that is a *straight-line program* (SLP) gives rise to an efficient factoring algorithm. An SLP is simply an arithmetic program (performing only ring operations) that does not branch. A complementary approach, which we pursue in this work, is to consider RSA *adversaries* that are restricted. The best known result of this nature is due to Aggarwal and Maurer (which we abbreviate as AM) [1], who showed that breaking RSA and factoring are *equivalent* wrt. so-called "generic-ring algorithms" (GRAs), namely ones that treat the ring $\mathbb{Z}_N$ like a black-box, only performing ring operations and equality checks that allow branching. Put another way, GRAs work in any efficient ring isomorphic to $\mathbb{Z}_N$. Note that SLPs are a special case of GRAs.

In the context of any cryptographic problem or protocol it is valuable to consider *preprocessing attacks*, because an adversary may be willing to perform highly intensive computation to break many instances of the problem, if that computation only has to be performed once. To model this, one considers an *unbounded* algorithm that produces a short "advice" string that can be used to efficiently solve a problem instance once it becomes known (much more efficiently than without the advice string). Note that above-mentioned attacks on RSA from [4] do not take advantage of preprocessing. However, in the preprocessing setting, Bernstein and Lange [3] describe a Number Field Sieve (NFS) with preprocessing, based on work by Coppersmith [7], which significantly reduces the exponent in the running-time compared to the standard NFS factoring algorithm, and they use this to get an improved attack on RSA. Thus, a natural question is:

> Does the relationship between RSA and factoring fundamentally change in the preprocessing setting?

The Need for a New Model. To answer this question, we need to formalize a model of computation for this setting. First, we will briefly survey some related models in the literature. The generic ring model (GRM) of AM considers an algorithm (called a generic ring algorithm or GRA) to be a directed acyclic graph where nodes are labelled with constants (or the input indeterminate) in $\mathbb{Z}_N^*$ and operations $(+, \times, \div)$; execution corresponds to a walk in the graph according to suitable rules. One can contrast this with Shoup's generic group model (GGM) [29], where the group representation is random and accessible only via an oracle; otherwise, an algorithm is allowed *arbitrary* computation. A Shoup-style GRM has also been considered by Dodis *et al.* [14]. We consider a hybrid of this model and AM's, wherein the ring representation is random and accessible via an oracle, but an algorithm is restricted (though more general than in AM). To understand the rationale, it is instructive to see why AM's model, extended to the preprocessing setting in the obvious way, is not suitable. In this

model, after the preprocessing stage the adversary outputs a GRA to run in the online stage. But then observe that the best the adversary could do in the preprocessing stage is to pick a single GRA of size at most some T that obtains optimal advantage, where the advantage is computed with respect to the random choice of N with bitlength at most security parameter κ and random choice of $y = x^e \pmod N$. The description of this optimal GRA would then be passed to the online stage. This process does not capture our intuition of what can be done with preprocessing. For example, the following simple algorithm would not be captured: Create a table of many input/output pairs $((y = x^e \pmod N), N), x)$ in the preprocessing stage, then, in the online stage, perform a lookup on the challenge input (y^*, N^*). Output the trivial GRA that outputs the constant $x^* = y^{*1/e} \pmod N$ if (y^*, N^*) is found in the table, and output the aforementioned optimal GRA otherwise. This algorithm cannot be captured in AM's model since the table lookup (via a binary tree or hash table structure) requires use of the bit-representation of the input (y^*, N^*), while a GRA is agnostic of the particular representation of the ring. While this is a simple example, it captures the techniques originating from Hellman's tables [17], which are common strategies for preprocessing algorithms in practice.

Our New "GRM-with-Preprocessing" Model. To allow these types of representation-specific strategies, we must associate integers y of bitlength at most κ with *labels*. This is somewhat analogous to moving Shoup's model of the GGM to the GRM setting. While this approach was used in Dodis et al. [14] for the specific problem of RSA-FDH, a version of the GRM that is analogous to Shoup's GGM has not been previously considered in full generality to the best of our knowledge. Indeed, we find that allowing completely arbitrary computation on labels becomes extremely hard to analyze. We therefore consider an intermediate model that allows for *representation-specific* yet *structured* algorithms.

In our definition of the generic ring model, an injective mapping π takes every element in $\{0,1\}^\kappa$ to a unique random string in $\{0,1\}^m$, where $m > \kappa$. We let the unbounded preprocessing algorithm read the entire description π and perform arbitrary computation. It produces a short advice string st that is passed to the online phase. The online algorithm is split into two parts, an intermediate algorithm, and a GRA. The intermediate algorithm is *bounded but not generic* and gets the problem instance $(N, e, \pi(x^e))$, where $N = pq$ has bit-length κ, but does *not* get access to π. This intermediate algorithm is crucial to our model, since this is what allows computation that *depends on the input representation*, and therefore allows the online part of the algorithm to leverage the advice from the preprocessing stage. Finally, this intermediate algorithm outputs an *oracle-aided* GRA that computes relative to π, and which we then run on the RSA problem instance. For example, an addition step of the oracle-aided GRA takes as input two strings $y_1, y_2 \in \{0,1\}^m$ and outputs $\pi(\pi^{-1}(y_1) + \pi^{-1}(y_2) \pmod N)$. (Multiplication and division proceed analogously.) We call $S = |\mathsf{st}|$ the space of the adversary and its running-time is specified by the pair (T_1, T_2), where T_1 is the runtime of the *intermediate algorithm*, and T_2 is the run-time of the GRA

output by the intermediate algorithm. (Note that we require that $T_2 \leq T_1$.) We refer to this model as the "GRM-with-preprocessing" for simplicity.

Main Results. We show that in the GRM-with-preprocessing model, any RSA algorithm with preprocessing implies the existence of a factoring algorithm with preprocessing in the random oracle model, with polynomially related parameters. This essentially answers our question for RSA algorithms in the GRM-with-preprocessing model and shows that the relationship of RSA and factoring does not fundamentally change in this setting.

Theorem 1. *(Informal.) Suppose there is an RSA adverary in the GRM with preprocessing model with space S_r and running-time $(T_{1,r}, T_{2,r})$ that succeeds with probability ϵ_r. Then there is a factoring adversary in the random oracle model (ROM) with space $S_f = S_r + O(1)$ and running-time $T_f = \mathsf{poly}(\kappa, T_{1,r}, T_{2,r}, 1/\epsilon_r)$ that succeeds with probability $\epsilon_f = \mathsf{poly}(\epsilon_r)$.*

See Theorem 4 for the formal statement. Our result implies that any significant speed-up on attacking RSA vs. factoring even *with preprocessing* must use non-generic techniques. Note that the space complexity of our factoring algorithm is essentiallly the same as that of the RSA algorithm. In terms of time complexity and success probability, our bounds are similar to those achieved by AM, which is to be expected. We differ from AM in that the success probability of our factoring algorithm ϵ_f depends only on ϵ_r, and not on $T_{1,r}, T_{2,r}$. We discuss additional differences between the time complexity and success probability of our factoring algorithm and that of AM in Section 4.

On the Use of a Random Oracle. We note that our final factoring algorithm with preprocessing is in the random oracle (RO) model. We believe using a RO is reasonable since prior work on space/time tradeoffs (such as the seminal results of Hellman [17] and Fiat-Naor [15]) either required a random oracle or achieved simplified algorithms/improved parameters in the random oracle model. Further, achieving a polynomial time factoring algorithm with preprocessing in the random oracle model would still be considered a breakthrough result.

Note that since our model allows an inefficient preprocessing phase, the random oracle (RO) cannot easily be removed from our final factoring algorithm while maintaining the desired polynomially related space complexity and runtime from Theorem 1. The reason is that in the preprocessing phase of the factoring algorithm, the entire RO could be queried and global information about it could be stored in the preprocessing advice. In this case, it is no longer possible for the online part of the factoring algorithm to simulate the RO "on the fly" since the responses generated by the simulator need to be consistent with the global information learned in the preprocessing phase. One approach would be to show that the global information about the random oracle (which has length S_f) can be simulated by fixing the input/output of some set of some q queries to the random oracle, and showing that any remaining queries not in this set can still be chosen "on the fly." This "bit-fixing" technique has been studied in a number

of works [9, 13]. However, this line of work proved a lower bound that q must be larger than $S_f T_f/(\epsilon_f)^2$ in order for this simulation to be ϵ_f-indistinguishable to an adversary making T_f queries (note that we require $\approx \epsilon_f$-indistinguishability to guarantee that the factoring algorithm in the standard (RO-devoid) model still succeeds with probability $\mathsf{poly}(\epsilon_f) = \mathsf{poly}(\epsilon_r)$). For us, this would lead to trivial parameter settings.

On Using Bit-Fixing Instead of Compression. A different approach to our use of the compression technique is to use *bit-fixing* instead (cf. [8]). That is, one would first show that an RSA algorithm of the form (A_0, A_1, A_2) with advice of size S_r, making at most T_r number of queries, and achieving success probability ϵ_r, implies the existence of an RSA algorithm of the form (A'_1, A'_2) making at most T'_r number of queries, and achieving success probability ϵ'_r in the *bit fixing* model, which fixes the labeling function π in q locations. It is possible that the AM reduction could then be applied more directly to (A'_1, A'_2) to obtain a factoring algorithm without going through a compression argument.

Unfortunately, similarly to the discussion above, this approach requires the number of fixed locations q to be at least $S_r T_r/(\epsilon_r)^2$. Since A'_1 cannot itself make oracle queries, for it to be able to choose A'_2 adaptively in the bit-fixing model, the information about the q fixed locations would need to be given to A'_1 as non-uniform advice. This would mean that the space of the RSA algorithm, and hence the resulting factoring algorithm, be at least $S_r T_r/(\epsilon_r)^2$, leading to trivial parameter settings.

1.2 Technical Overview

Our main result shows that any generic attack on RSA with preprocessing gives rise to a factoring algorithm with preprocessing in the random oracle model with polynomially related parameters. We begin by recapping the subclass of RSA algorithms we consider, and then discuss the high level approach of our proof of equivalence.

The RSA algorithm. Recall that we consider RSA adversaries that are split into two 'fixed' parts (A_0^π, A_1) and a third part G^π that is adaptively chosen by A_1 upon seeing the RSA instance. In more detail, A_0^π gets oracle access to $\pi : \{0,1\}^\kappa \to \{0,1\}^m$ and is completely unbounded *both* in terms of computation and number of queries to π. A_0^π finally outputs a state st of size S_r (called A's *space*). A_1 takes as input st and the RSA instance $(N, e, \pi(y) = \pi(x^e))$, runs in time $T_{1,r}$, and outputs a GRA G^π of size (and hence running-time) $T_{2,r}$. The GRA G^π is an oracle-aided program that computes relative to π. In other words, each multiplication (resp. division, addition) step of G^π with inputs y_1, y_2 outputs $\pi(\pi^{-1}(y_1) \cdot \pi^{-1}(y_2) \pmod N)$ (resp. $\pi(\pi^{-1}(y_1) \cdot (\pi^{-1}(y_2))^{-1} \pmod N)$, $\pi(\pi^{-1}(y_1) + \pi^{-1}(y_2) \pmod N)$). A_1 is computationally bounded but may run for superpolynomial time; however, it may not make any queries to the oracle π. Finally, G^π takes as input $\pi(y)$ and evaluates $G^\pi(\pi(y))$. In the following, we fix π, a state st of some bounded size S_r output by A_0^π, as well as a modulus N and value e with $\gcd(e, \phi(N)) = 1$. We consider the success probability ϵ_r on input

$\pi(y)$ of A_1 relative to these fixed values in outputting G^π such that $G^\pi(\pi(y)) = \pi(x)$ and $x^e = y \pmod N$. Here, the success probability is taken over random choice of $y \leftarrow \mathbb{Z}_N$ and coins of A_1. Fixing π, st, N, e simplifies our discussion and can easily be justified by an averaging argument. Our final analysis, however, considers these values drawn from an appropriate distribution. Our goal is to construct a factoring algorithm in the random oracle model *with preprocessing* and with parameters S_f, T_f, ϵ_f (space, time, and success probability) that are polynomially related to $S_r, T_{1,r}, T_{2,r}, \epsilon_r$. Specifically, we require that $S_f = S_r + O(1)$, $T_f = \mathsf{poly}(\kappa, T_{1,r}, T_{2,r}, 1/\epsilon_r)$ and $\epsilon_f = \mathsf{poly}(\epsilon_r)$, where $\kappa = \log(N)$ is security parameter.

In the following, we first restrict our attention to the special case where A_1 outputs a straight-line program (SLP) with addition/multiplication *only* (i.e., without equality checks). This special case already requires most of the key ideas of our proof. We then explain how to extend our result to the case where A_1 may output a generic ring algorithm (GRA). Extending the model to allow division is straightforward and we therefore omit this part from the technical overview.

Both our model and main theorem are very general in the sense that they show existence of a factoring algorithm with polynomially related parameters for *any setting of RSA parameters* $T_{1,r}, T_{2,r}, S_r, \epsilon_r$ and for a general class of algorithms. Our result does not restrict the relationship between $(T_{1,r}, T_{2,r}, S_r)$ (other than the requirement that $T_{1,r} \geq T_{2,r}$, which is implied by the model) and we show that generic RSA with preprocessing implies factoring with preprocessing, even for unconventional parameter settings (such as setting S_r to be larger than the time complexity of the best online factoring algorithm). We believe it is important to cover all parameter regimes, as this ensures that our result actually suggests a mathematical connection between the factoring and RSA problems themselves, rather than just showing that for the typical parameter settings used in practice the best factoring and RSA algorithms happen to have the same complexity.

We consider algorithms with unbounded preprocessing. Moreover, the algorithm A_1 does not have access to π, but can perform arbitrary (and superpolynomially many) operations after learning the modulus N and the RSA instance $\pi(x^e)$. Only then does it hand over the remaining computation to the fully generic program G^π. In order for this to be possible, we must do several case analyses. To simplify this technical overview, we will henceforth conflate the online portion of algorithm's running times by setting $T_r = T_{r,1} + T_{r,2}$.

First case analysis: Fiat-Naor argument. In the case that $T_r \cdot S_r \geq \epsilon_r \cdot 2^\kappa/16$, we will completely ignore the RSA algorithm, and construct a different Factoring algorithm in the RO model "from scratch." The idea is to use a theorem of Fiat and Naor [15], which extends Hellman's seminal result on space/time tradeoffs for inversion of a *random* function [17], to obtain space/time tradeoffs for inversion of *any* function f. Specifically, Fiat and Naor consider an arbitrary function $f : D \to D$ and show that f can be inverted with probability $1 - 1/|D|$ in the random oracle (RO) model with space S and time T, as long as $S^2 \cdot T \geq |D|^3 \cdot q(f)$,

where $q(f)$ is the probability that two random elements in D collide under f.[4] We apply Fiat-Naor to the factoring problem by viewing f as the function that takes two $\kappa/2$ bit strings and multiplies them to obtain a κ-bit string, where $\kappa = \log(N)$. Using properties of the second moment of the divisor function, we bound $q(f) \in O(\frac{\kappa^3}{2^\kappa})$. Thus, we obtain a factoring algorithm with $S_{f'} := 1/\epsilon \cdot S_r$ and and $T_{f'} := O(\kappa^3) \cdot T_{2,r}^2$ that inverts with probability $1 - 1/2^\kappa$. Indeed, one can check that for the $S_{f'}, T_{f'}$ defined above, $S_{f'}^2 \cdot T_{f'} \geq 2^{2\kappa} \cdot \kappa^3$ as long as $T_r \cdot S_r \geq \epsilon_r \cdot 2^\kappa/16$. We then observe that since we only need $O(\epsilon)$ success probability, we can reduce the space from $S_{f'} = 1/\epsilon \cdot S_r$ to $S_f = S_r$, while $T_f = O(T_{f'})$. Thus, our final parameters are $S_f = S_r$, $T_f = \mathsf{poly}(\kappa) \cdot T_{2,r}^2$ and inversion probability $O(\epsilon)$. Note that all parameters are polynomial in the parameters of the RSA algorithm. Our goal is therefore achieved in the case that $T_r \cdot S_r \geq \epsilon_r \cdot 2^\kappa/16$. See Section 4.4 for more details.

Factoring from RSA. We now consider the main parameter regime of interest, where $T_r \cdot S_r < \epsilon_r \cdot 2^\kappa/16$. In this parameter regime, we will show how to use the RSA algorithm to construct a factoring algorithm. However, before we can do that, we need to eliminate a crucial case in which the RSA algorithm is unhelpful for constructing a factoring algorithm. Let us first consider when and why the RSA algorithm is useful for factoring. Then we will show how to eliminate the remaining case.

Note that if A is successful with probability ϵ_r, then with probability ϵ_r the SLP S output by A_1 is such that on a randomly chosen $y = x^e$, $S^\pi(\pi(y)) = \pi(x)$. We define an "inversion procedure" on SLP's that, given S such that $S^\pi(\pi(y)) = \pi(x)$ and oracle access to π, outputs an SLP $\tilde{S}$ with no oracle access such that $\tilde{S}(y) = x$. This, in turn, means that y is a root of the SLP $\tilde{S}(Y)^e - Y$, with respect to formal variable Y. In AM's analysis, they were able to conclude that if A is successful, then $\tilde{S}(Y)^e - Y$ must have many roots. Then, they showed an algorithm that successfully factors, given as input a non-zero SLP $\tilde{S}(Y)^e - Y$ with a sufficiently large fraction of roots. In our setting, however, we cannot necessarily conclude this.. This is because we allow A_1 to output a different SLP $S_{\pi(y)}$ *after seeing* input $\pi(y)$ (we use the notation $S_{\pi(y)}$ to emphasize that the chosen SLP may depend on $\pi(y)$). This means that the SLP $S_{\pi(y)}$ output by A_1 can be tailored to succeed on $\pi(y)$ and on only *few* other inputs, while A_1 maintains overall high success probability. So while w.h.p. y itself must still be a root of the "inverted SLP" $\tilde{S}_{\pi(y)}(Y)^e - Y$, we are not guaranteed that $\tilde{S}_{\pi(y)}(Y)^e - Y$ has many roots overall. In this case, factoring fails.

The above reasoning leads to the second and third cases considered in our proof: The second case is that w.h.p. y is a root of $\tilde{S}_{\pi(y)}(Y)^e - Y$, *and* $\tilde{S}_{\pi(y)}(Y)^e - Y$ has *at most* J roots. The third case is that w.h.p. the SLP $\tilde{S}_{\pi(y)}(Y)^e - Y$, has *at least* J roots. The second case will lead to contradiction due to a compression argument. We will therefore be left with the third case which will imply existence of a factoring algorithm.

[4] Their final algorithm actually requires only k-wise independent hash functions instead of a RO. For simplicity, we instead assume a RO with $O(1)$ evaluation time.

Second case analysis: Compression. Intuitively, if y is a root of $\tilde{S}_{\pi(y)}(Y)^e - Y$, *and* $\tilde{S}_{\pi(y)}(Y)^e - Y$ has *at most* J roots, then there is an efficient way to transmit y to a decoder who knows $\tilde{S}_{\pi(y)}$: Simply output the index of y among the J roots of $\tilde{S}_{\pi(y)}(Y)^e - Y$. Intuitively, we save space when $\log(J)$ is small compared to the trivial encoding of y, which specifies the index of y among all pre-images that are not yet mapped to an image. Making this intuition rigorous is quite challenging.

First, we must show how the encoder can efficiently transmit the description of $S_{\pi(y)}$ to the decoder. Although A_1 and st are known to the decoder, to obtain the correct SLP $S_{\pi(y)}$, the decoder must run A_1 on the correct random coins ρ and on the correct input $\pi(y)$. Furthermore, A_1 is only guaranteed to output an SLP $S_{\pi(y)}$ that is successful on $\pi(y)$ w.h.p., when $\pi(y) = \pi(x^e)$ and ρ are chosen *uniformly at random*. But we cannot afford to transmit the value of a random $\pi(y)$, nor the value of random coins ρ of A_1, while still achieving compression. To solve both of these problems, we rely, as prior work of Corrigan-Gibbs and Kogan [10] did, on a lemma of De, Trevisan, and Tulsiani [12]. This lemma proves incompressibility of an element x from a sufficiently large set $\mathcal{X}$ in a setting that allows the encoder and decoder to pre-share a random string of arbitrary length. For our purposes, this random string will allow us to both (1) select a random $\pi(y)$ from the set of images whose preimages are not yet known and (2) select the random tape ρ for A_1 to use together with input $\pi(y)$. Thus, the successful randomness can simply be encoded by its index within the shared random string, thus saving space. We mention that Corrigan-Gibbs and Kogan avoided encoding successful $\pi(y)$ values by using the random self-reducibility property of the discrete log problem to obtain an adversary that succeeds w.h.p. on *every* input. Unlike Corrigan-Gibbs and Kogan, our argument does *not* require random self-reducibility, and rather uses the random tape to select a random image $\pi(y)$ instead. Thus, while RSA also enjoys random self-reducibility, our proof does not make use of it, potentially making our techniques applicable to broader settings.

The third challenge is that in order to obtain $\tilde{S}_{\pi(y)}$ from $S_{\pi(y)}$, the decoder must run the SLP inversion procedure, which requires access to π. Therefore, we include all the responses of queries to π during evaluation of the SLP inversion procedure in the encoding, replacing any query to $\pi^{-1}(\pi(y))$ itself with the formal variable Y. The final challenge is the delicate setting of parameters needed for the result to go through. We must set the value J (the number of roots in the SLP $\tilde{S}_{\pi(y)}(Y)^e - Y$) such that compression is achieved when the number of roots is at most J and, looking ahead, such that efficient factoring (with parameters S_f, T_f, ϵ_f that are polynomially related to S_r, T_r, ϵ_r) is possible when the number of roots is at least J. We note that our techniques for analyzing the encoding length are significantly different from those used by Corrigan-Gibbs and Kogan and may be of independent interest. (See Section 4.2 for more details.)

Third case analysis: Factoring routine. When the SLP $S_{\pi(y)}(Y)^e - Y$, has *at least* J roots w.h.p., we can apply a theorem of AM to obtain a factoring algorithm. Unfortunately, the final factoring algorithm (with preprocessing) that

we obtain is in the random injection (RI) model, where the algorithm requires access to *both* π and π^{-1}. So our final step is to switch from the RI model to the more-standard RO model. In other words, we need to construct a RI from a RO. To construct a RI from κ-bits to m-bits for some $m \geq \kappa$, the idea is to construct a random permutation (RP) on m-bits and define the RI on subdomain $\{0, \ldots, 2^\kappa - 1\}$. To construct a RP, we use a result of Luby and Rackoff [21] showing that forward and backward oracle access to a 4-round Feistel network with $m/2$-bit ROs as the round functions is indistinguishable from forward and backward oracle access to a RP. (Note that the adversary does *not* have oracle access to the round functions.) However, the distinguishing probability is $\frac{q^2}{2^{m/2}}$, where q is the number of queries made by the adversary. So, for example, $m = \kappa$ will not lead to a sufficient security level since A_0 can query the RI on *all* inputs from $\{0, \ldots, 2^\kappa - 1\}$ and thus q can be as large as 2^κ, which renders the distinguishing probability of $\frac{q^2}{2^{m/2}} > 1$ meaningless. We therefore use a larger m such that $(2^\kappa)^2/2^{m/2} \ll \tilde{\epsilon}/2$, where $\tilde{\epsilon}$ is the success probability of the factoring algorithm in the RI model. Now the output of the random injective function on an input in $\{0, \ldots, 2^\kappa - 1\}$ is obtained by padding the input (e.g. with 0's) up to a length of m-bits, and then running the 4-round Feistel construction with $m/2$-bit ROs as the round functions. Now, although A_0 may query the entire truth table of the injective function, this corresponds to making only $q = 2^\kappa$ number of queries. And since $(2^\kappa)^2/2^{m/2} \ll \tilde{\epsilon}/2$, oracle access to the 4-round Feistel is $\tilde{\epsilon}/2$-indistinguishable from oracle access to random injective function. This means that the factoring algorithm must still succeed with probability $\tilde{\epsilon}/2$ relative to the 4-round Feistel in the RO model. This means that obtain our final factoring algorithm in the RO model and we *do not* require forwards and backwards access to a RI. See Sections 4.3 and 4.3 for additional details.

Extending to the GRA case. We first note that, in fact, compression can be achieved in a slightly broader setting. Specifically, we define a notion of "y being negatively oriented w.r.t. an SLP S" and show that if y is negatively oriented w.r.t. S then there is an efficient way to transmit y to a decoder who knows S. We say that "y is negatively oriented w.r.t. an SLP S" if either (1) S has *few* roots (δ-fraction or less) and y is one of the roots or (2) S has *many* roots ($(1-\delta)$-fraction or more) and y is a non-root. In this case, given S, one can efficiently encode y by giving the index of the root or non-root, as appropriate.

Now, given a GRA G we consider the *dominating path* of the GRA, which itself corresponds to an SLP $S_{\psi+1}$. As defined by AM, this means that whenever we reach a branching vertex, we go either left or right depending on whether most inputs satisfy the equality test or dissatisfy the equality test. We view every branching vertex as a comparison of two SLP's evaluated at the input. When comparing two SLPs, S_1, S_2, if most inputs satisfy the equality test it means that $S_1 - S_2$ has *many* roots, whereas if most inputs dissatisfy the equality test it means that $S_1 - S_2$ has *few* roots.

We now consider an A_1 that outputs w.h.p. a successful GRA G^π such that $G^\pi(\pi(y)) = \pi(x)$. Let $\tilde{G}$ be the "inverse" of G^π relative to π, and let $S_1, \ldots, S_{\psi+1}$

corresponding to the nodes along the *dominating path* of $\tilde{G}$. We show that one of the following must occur: (1) y is negatively oriented to some $S_i \in S_1, \ldots, S_{\psi+1}$. Intuitively, this corresponds to the case that either input y *does not* take the dominating path in $\tilde{G}$, or that y does take the dominating path but that $\tilde{G}$ succeeds on only a small fraction of inputs. In this case y can be efficiently encoded. (2) The SLP $S_{\psi+1}$ has a large fraction of roots. In this case we can run a factoring algorithm inspired by AM that is guaranteed to either find a factor of N or output the dominating path of $\tilde{G}$ with high probability. If the algorithm succeeds in finding the dominating path, then we can finally invoke the previous factoring algorithm for the SLP special case to complete the proof.

1.3 Related Work

There is an extensive body of literature on the hardness of the RSA problem and is relationship to factoring. Boneh and Venkatesan [5] gave the first among these results. Their result shows that reducing low-exponent RSA from factoring using a straight-line reduction is as hard as factoring itself. A similar result by Joux *et al.* [18] shows that when given access to an oracle computing eth roots modulo N of integers $x + c$ (where c is fixed and x varies), computing eth roots modulo N of arbitrary numbers becomes easier than factoring.

A more closely related line of work initiated by Brown [6] shows that for generic adversaries, computing RSA (or variants thereof), is as hard as factoring the modulus N. Brown's initial work considered only the case of SLPs without division and was subsequently extended by Leander and Rupp [20] to the case of GRAs without division.The work of Aggarwal and Maurer [1] finally showed that the problems are equivalent even for GRAs with division. A subsequent result of Jager and Schwenk showed that computing Jacobi symbols is equivalent to factoring for GRAs. Their result puts into question the soundness of the generic ring model (GRM), as it shows that there are problems which are hard in the GRM, but easy in the plain model. On the other hand, this result has no immediate implication for other computational problems like the RSA problems, which may still be meaningful to consider in the GRM. A recent work by Rotem and Segev also showed how the GRM can been used to analyze the security of verifiable delay functions [28].

The Generic Group Model (with Preprocessing). Starting with Nechaev [24], a long line of work has studied the complexity of group algorithm in the generic group model (GGM) [29, 22]. Algorithms in this model are restricted to accessing the group using handles and cannot compute on group elements directly. This makes it possible to prove information theoretic lower bounds on the running times and success probabilities of generic group algorithms for classic problems in cyclic groups (e.g., DLP, CDH, DDH). To the best of our knowledge, only two works have considered the RSA problem in idealized group models. The first of these work is due to Damgard and Koprowski [11] who ported Shoup's generic group model [29] to the setting of groups with unknown order and showed the generic hardness of computing eth roots in this model. The second work is that

of Dodis et al. [14] who considered the instantiability of the hash function in FDH-RSA when modelling $\mathbb{Z}_N^*$ as a generic group. Their techniques are similar to ours in the sense that they combine factoring and compression arguments. On the one hand, their techniques apply to a more general type of online adversary who can perform side computations over $\mathbb{Z}_N^*$ in any way that it likes. Recall that we do not allow such computations in our model once we have chosen the generic online algorithm to solve the RSA instance (although the generic algorithm itself can depend on the instance in non-adaptive and non-generic fashion). On the other hand, we face many additional technical issues due to considering a more general generic model of computation for the ring $\mathbb{Z}_N$ as well as preprocessing. Even more recently, the work of Corrigan-Gibbs and Kogan [10] initiated the study of preprocessing algorithms in the GGM. They considered generic upper and lower bounds for the discrete logarithm problem and associated problems. Their modelling approach is very similar to our own, in that the algorithm in the offline phase has access to the labelling oracle π and can pass an advice string of bounded size to the online phase of the algorithm. A key difference is that in their setting, the group is fixed throughout the offline and online phase, whereas in our setting, the group is fixed together with the RSA instance only in the online phase. Moreover, they can also consider adversaries who, in the online phase, may perform arbitrary side computations.

The Algebraic Group Model. More recently, a series of works has explored the algebraic group model [16] as a means to abstract the properties of the groups $\mathbb{QR}_N$ and $\mathbb{Z}_N^*$ more faithfully. The work of Katz et al. [19] introduced a quantitative version of the algebraic group model called the strong algebraic group model to relate the RSW assumption [27] over $\mathbb{QR}_N$ to the hardness of factoring (given that N is a product of safe primes p, q). Their model and ideas were extended to $\mathbb{Z}_N^*$ by Stevens and van Baarsen [30] who gave a general framework for computational reductions in the (strong) algebraic group model over $\mathbb{Z}_N^*$.

2 Preliminaries

2.1 Notation and Conventions

We denote the sampling of a uniformly random element x from a set S as $x \leftarrow S$. Similarly, we denote the output y of a randomized algorithm A on input x as $y \leftarrow \mathsf{A}(x)$. We sometimes also write $y := \mathsf{A}(x;\omega)$ to denote that A deterministically computes y on input x and random coins ω. To denote that an algorithm A has access to an oracle O during runtime, we write A^{O}. We denote as $\mathbb{Z}_N$ the ring of integers modulo N, and as $[N]$ the set $\{1, ..., N\}$. We write $\nu_N(f)$ to denote the fraction of roots of a polynomial f over $\mathbb{Z}_N$, i.e.,

$$\nu_N(f) := \frac{|\{a \in \mathbb{Z}_N \mid f(a) = 0\}|}{N}.$$

Throughout, we denote the security parameter as κ. For $k, m \in \mathbb{N}$ we denote by $\mathsf{Func}[k, m]$ the set of *functions* $F\colon \{0,1\}^k \to \{0,1\}^m$. Denote by $\mathsf{Perm}[m]$ the

set of *permutations* on $\{0,1\}^m$. We denote by $\mathsf{FuncInj}[k,m]$ the set of *injective functions* $I\colon \{0,1\}^k \to \{0,1\}^m$.

2.2 Incompressibility Lemma

We use the following lemma by De et al. [12].

Lemma 1. *(De, Trevisan, Tulsiani [12].) Let* $\mathsf{E} : \mathcal{X} \times \{0,1\}^\rho \to \{0,1\}^m$ *and* $\mathsf{D} : \{0,1\}^m \to \mathcal{X} \times \{0,1\}^\rho$ *be randomized encoding and decoding procedures such that, for every* $x \in \mathcal{X}, \Pr_{r \leftarrow \{0,1\}^\rho}[\mathsf{D}(\mathsf{E}(x,r),r) = x] \geq \gamma$*. Then,* $m \geq \log|\mathcal{X}| - \log 1/\gamma$.

Remark 1. As noted by [10], this lemma also holds when the encoding and decoding algorithms have access to a common random oracle.

2.3 Relevant Problems

In this subsection, we introduce the main relevant problems: the RSA Problem, the Factoring Problem, and the general Function Inversion Problem (all with preprocessing). Algorithm RSAGen on input 1^κ generates (N,e,d,p,q) where $N = pq$ and p,q are primes of bit-length $\kappa/2$ with leading bit 1. Finally, $ed = 1 \bmod \phi(N)$.

Definition 1 (Factoring with Preprocessing). *Let* $\mathsf{F} = (\mathsf{F}_0, \mathsf{F}_1)$ *be an algorithm and* RSAGen *be an RSA generator. Consider the factoring-with-preprocessing game* $\mathbf{fac}^{\mathsf{F}}_{\mathsf{RSAGen}}$*:*

- **Offline Phase.** *Run* F_0 *on input* 1^κ *to obtain an* advice string st.
- **Online Phase.** *Run* RSAGen *on input* 1^κ *to obtain*(N,e,d,p,q)*. Then run* F_1 *on input* (N,st).
- **Output Determination.** *When* F_1 *returns* p'*, the experiment returns* 1 *if* $p = p'$ *or* $q = p'$*. It returns* 0 *otherwise.*

Define F*'s advantage in the above experiment as*

$$\mathsf{Adv}^{\mathbf{fac}}_{\mathsf{RSAGen}}(\mathsf{F}) = \Pr[\mathbf{fac}^{\mathsf{F}}_{\mathsf{RSAGen}} = 1] \ .$$

We call F *an* (S,T)-factoring algorithm relative to RSAGen *if* F_0 *outputs advice strings of size at most* S *and* F_1 *runs in time at most* T*.*

Definition 2 (RSA with Preprocessing). *Let* $\mathsf{A} = (\mathsf{A}_0, \mathsf{A}_1)$ *be an adversary. Consider the RSA-with-preprocessing game* $\mathbf{rsa}^{A}_{\mathsf{RSAGen}}$*:*

- **Offline Phase.** *Run* A_0 *on input* 1^κ *to obtain an* advice string st.
- **Online Phase.** *Run* RSAGen *on input* 1^κ *to obtain*(N,e,d,p,q)*. Sample* $x \leftarrow \mathbb{Z}_N^*$ *and run* A_1 *on input* $(N,e,\mathsf{st},x^e \bmod N)$.
- **Output Determination.** *When* A_1 *returns* x'*, the experiment returns* 1 *if* $x = x' \pmod N$*. It returns* 0 *otherwise.*

Define A's advantage in the above experiment as

$$\mathsf{Adv}^{\mathbf{rsa}}_{\mathsf{RSAGen}}(\mathsf{A}) = \Pr[\mathbf{rsa}^{\mathsf{A}}_{\mathsf{RSAGen}} = 1] \,.$$

We call A *an* (S,T)-RSA algorithm relative to RSAGen *if* A_0 *outputs advice strings of size at most* S *and* A_1 *runs in time at most* T*.*

In the following, we consider a domain D of finite size along with a randomized point generator G that outputs points in D.

Definition 3 (Function Inversion with Preprocessing). *Let* D *be a finite set and let* $f\colon D \to D$ *be a function. Let* $\mathsf{I} = (\mathsf{I}_0, \mathsf{I}_1)$ *be an adversary and* Gen *a point generator. Consider the function-inversion-with-preprocessing game* $\mathbf{func}^{\mathsf{I}}_{f,\mathsf{Gen}}$*:*

- **Offline Phase.** *Run* I_0 *on input* 1^κ *to obtain an* advice string st*.*
- **Online Phase.** *Run* Gen *on input* 1^κ *to obtain a point* $y \in D$*. Run* I_1 *on input* (y, st)
- **Output Determination.** *When* I_1 *returns* x'*, the experiment returns* 1 *if* $f(x') = y$*. It returns* 0 *otherwise.*

Define I*'s advantage in the above experiment as*

$$\mathsf{Adv}^{\mathbf{func}}_{f,\mathsf{Gen}}(\mathsf{I}) = \Pr[\mathbf{func}^{\mathsf{I}}_{f,\mathsf{Gen}} = 1] \,.$$

We call I *an* (S,T)-function-inversion algorithm relative to Gen *if* I_0 *outputs advice strings of size at most* S *and* I_1 *runs in time at most* T*.*

Definition 4 (Collision Probability). *Let* D *be a finite set and let* $f\colon D \to D$ *be a function. For* $z \in D$*,* $I_f(z)$ *denotes the number of preimages for* z *under* f*, i.e.*

$$I_f(z) := |\{u \in D : f(u) = z\}| \,.$$

The collision probability of $f\colon D \to D$*, denoted by* $q(f)$ *is defined as follows:*

$$q(f) := \frac{\sum_{z \in D} I_f^2(z)}{|D|^2}.$$

Theorem 2 (Fiat-Naor [15]). *For any* D, f, Gen *as in Definition 3 and any* S, T *such that* $T \cdot S^2 = |D|^3 \cdot q(f)$*, there is a RO-model* (S,T)*-function-inversion algorithm* I *such that* $\mathsf{Adv}^{\mathbf{func}}_{f,\mathsf{Gen}}(\mathsf{I}) \geq 1 - 1/|D|$*.*[5]

3 Computational Models

In this section, we review some idealized models that will be relevant in our analyses and discuss their relationships to each other.

[5] This statement is weaker than the one proven in [15] but is sufficient for our purpose.

Random Oracle Model (ROM). In the random oracle model [2] all algorithms have oracle access to a uniformly random function from $\mathsf{Func}[m_1, m_2]$ for some $m_1, m_2 \in \mathbb{N}$ specified by the model.

Random Injection Model (RIM). In the random injection model all algorithms have *forwards and backwards* oracle access to a uniformly random function from $\mathsf{FuncInj}[n, m]$ for some $n \leq m$ specified by the model.

Random Permutation Model (RPM). In the random permutation model all algorithms have *forwards and backwards* oracle access to a uniformly random function from $\mathsf{Perm}[m]$ for some $m \in \mathbb{N}$ specified by the model.

3.1 Switching from RIM to ROM

To switch from the RIM to the ROM, we need to show how to simulate oracle access to a random injection (forward and backward), given oracle access to a random function. We implement the random injection by padding the input and using Luby-Rackoff's strong pseudorandom permutation construction [21].

Luby-Rackoff. We first recall the Luby-Rackoff construction [21], which we view as a construction of a random permutation oracle from a random oracle. Formally, suppose ρ is a RO from $\{0,1\}^{m/2}$ to $\{0,1\}^{m/2}$ for $m \in \mathbb{N}$. Define oracle $\mathsf{LubRac}[\rho]$ on $\{0,1\}^m$ as follows:

- Parse x as $x_1 \| x_2$ with $|x_1| = |x_2| = m/2$ and apply a 4-round balanced Feistel network with h as the round function to obtain y. Output y.

Oracle $\mathsf{LubRack}^{-1}[\rho]$ is defined accordingly.

Theorem 3 (Luby-Rackoff [21]). *For any (even unbounded) adversary* A *making at most* q *queries it holds that*

$$\Big|\Pr_{\rho \leftarrow \mathsf{Func}[m/2,m/2]}[\mathsf{A}^{\mathsf{LubRack}[\rho](\cdot),\mathsf{LubRack}^{-1}[\rho](\cdot)} \text{ outputs } 1] - \Pr_{\pi \leftarrow \mathsf{Perm}[m]}[\mathsf{A}^{\pi(\cdot),\pi^{-1}(\cdot)} \text{ outputs } 1]\Big| \in \mathcal{O}(q^2/2^{m/2}).$$

Random Injection from Random Permutation. We next show a construction of a random injection oracle π from a random permutation oracle ψ. Suppose ψ is a random permutation oracle on m bits and ψ^{-1} is its inverse. For $n \leq m$, define $\pi[\psi] \colon \{0,1\}^n \to \{0,1\}^m$ as $\pi[\psi](x) := \psi(\mathsf{pad}(x))$ where $\mathsf{pad}(x)$ is the function that pads the LSBs of x with $m - n$ zeros. Define $\pi[\psi]^{-1}$ accordingly. It should be clear that $\pi[\psi]$ is a random injection oracle.

Now, composing the above constructions gives a construction of a random injection oracle from a random oracle. Namely, suppose $\rho \colon \{0,1\}^{m/2} \to \{0,1\}^{m/2}$ is a RO. Define the random injection oracle $\pi[\rho] \colon \{0,1\}^n \to \{0,1\}^m$ as $\pi[\rho](x) = \mathsf{LubRac}[\rho](\mathsf{pad}(x))$ and $\pi[\rho]^{-1}$ accordingly. By a simple hybrid argument we have:

Proposition 1. *(RIM-to-ROM.) For any (even unbounded) adversary* A *making at most* q *queries it holds that*

$$\Big|\Pr_{\rho\leftarrow\mathsf{Func}[m/2,m/2]}[A^{\pi[\rho](\cdot),\pi[\rho]^{-1}(\cdot)} \text{ outputs } 1]-$$
$$\Pr_{\pi\leftarrow\mathsf{FuncInj}[n,m]}[A^{\pi(\cdot),\pi^{-1}(\cdot)} \text{ outputs } 1]\Big| \in \mathcal{O}(q^2/2^{m/2}).$$

3.2 Straight-Line Programs and Generic Ring Algorithms

Let $N \in \mathbb{N}$ and assume that $m \geq \kappa$, where κ is the bit length of N. Below, we define two types of programs (aka. algorithms) that use oracles, namely generic-ring algorithms (GRAs) and straight-line programs (SLPs).

Program Graphs and Their Execution. The below is based on [1]. We consider deterministic programs that perform arithmetic operations (mod N) on indeterminate Y.

We associate a program on a single input with its *program graph over* $\mathbb{Z}_N$, a labelled graph where a label of a node represents a (binary) operation and the program implicitly stores all intermediate results. We only consider programs whose graphs are binary trees. Vertices can be either *branching* or *non-branching*.

Execution of a program corresponds to traversing a labelled path in its program graph over $\mathbb{Z}_N$. *Non-branching vertices* are used to execute arithmetic operations (mod N) or to load inputs and constants into the program. They are accordingly labelled with elements $a \in \mathbb{Z}_N$ corresponding to constants in the program, with a (unique) indeterminate Y corresponding the programs input, or with an arithmetic operation label $(i, j, \circ, b)$ which applies the arithmetic ring operation $\circ$ (mod N) to operands at indices i and j that the program previously stored. (The flag $b \in \{-1, 1\}$ indicates inversion of the second operand.) *Branching vertices* are used to test two values i, j previously computed by the program for equality (mod N). A branching vertex has two outgoing edges that are labelled 0 (for left) and 1 (for right).

The program applies the operations indicated by the labels of the vertices and edges it encounters in the order of traversal as follows:

- The first three vertices are a path and are always labelled 0, 1, and Y. That is, they are used to load the constants 0 and 1, and the single input y of the program. The program stores the intermediate results $y_0 = 0, y_1 = 1, y_2 = y$ for these vertices, respectively, and continues execution along this path.
- For $k \geq 4$:
  - If the kth vertex v_k is labelled with $a \in \mathbb{Z}_N$, the program stores $y_k \leftarrow a$ as the intermediate result for this vertex. It continues execution along this path.
  - If the kth vertex v_k is labelled with $(i, j, \circ, b)$ then the program does as follows. Here $\circ \in \{\cdot, +\}, b \in \{-1, 1\}$, and $i, j < k$ correspond to the ith and jth vertices on the path of traversal, which must be non-branching. The program computes $y_k := y_i \circ y_j^b$ (mod N) and stores

the intermediate result y_k for vertex v. In case $\circ = +$ and $b = -1$, then $y_j^b = -y_j \pmod N$. In case $\circ = \cdot$, $b = -1$, and $y_j = 0 \pmod N$, $y_k := \perp$. In case $y_i = \perp$ or $y_j = \perp$, $y_k := \perp$. It continues execution along this path.

- If the kth vertex v_k is labelled (i, j) where $i, j < k$ correspond to the ith and jth vertices on the path of traversal, which must be non-branching, the program makes an *equality test* whether $y_i = y_j \pmod N$. If the result is 1, the program continues its execution along the right edge; otherwise, along the left.

– Whenever v_k is the last vertex on the path, the program computes y_k and outputs it, terminating execution.

Oracle-Aided Programs. Apart from the types of programs we have discussed above, we are also interested in programs that can perform arithmetic operations via oracle access (as opposed to directly).

Hence, we define oracles π, eq, and op_π as follows. Oracle π initially samples a random function $\pi \in \mathsf{FuncInj}[\kappa, m]$ and on query $x \in \mathbb{Z}_N$ returns $y = \pi(x) \in \{0,1\}^m$. Here we refer to $y \in \{0,1\}^m$ as a *label*. We slightly abuse notation by referring to the oracle π and the internally sampled function indiscriminately. We also make the convention of parsing $x \in \mathbb{Z}_N$ as a κ-bit binary string. Given $\pi \in \mathsf{FuncInj}[\kappa, m]$, we first consider an oracle eq for testing equality. On input $y_1, y_2 \in \{0,1\}^m$, eq returns 1 iff $\pi^{-1}(y_1) = \pi^{-1}(y_2) \pmod N$, and 0 otherwise. Now, we define the behavior of the *ring oracle* op_π on input as $y_1, y_2 \in \{0,1\}^m$ as

$$\mathsf{op}_\pi(y_1, y_2, \circ, b) := \pi\left(\pi^{-1}(y_1) \circ \left(\pi^{-1}(y_2)\right)^b \bmod N\right)$$

for all $y_1, y_2 \in \{0,1\}^m$, $\circ \in \{+, \cdot\}$, $b \in \{1, -1\}$, where the inverse is additive in case $\circ = +, b = -1$. We implicitly assume that in case $\circ = \cdot, b = -1$, op_π internally queries $\mathsf{eq}(y_2, 0)$. op_π returns $\perp$ in case either of the operands is $\perp$ or the call to eq returns 1, i.e., if $\pi^{-1}(y_2) = 0 \pmod N$.

Oracle-aided program graphs over $\mathbb{Z}_N$ are labelled very similarly to plain program graphs over $\mathbb{Z}_N$. Roughly speaking, all values in $\mathbb{Z}_N$ are now replaced with their labels, according to π. Thus, a non-branching vertex is now labelled in one of two ways. Either it is labelled with $(i, j, \circ, b)$ where i and j correspond to the ith and jth non-branching vertex among the vertices previously encountered on the path and $\circ \in \{+, \cdot\}, b \in \{1, -1\}$. Otherwise, it is labelled with some m-bit label σ in the image of π.

As before, a branching vertex is labeled with (i, j), where i and j correspond to the ith and jth non-branching vertex among the vertices previously encountered on the path. It has two outgoing edges labelled 0 (for left edge) and 1 (for right edge). The only difference is that the program now has to invoke eq on the intermediate values y_i and y_j so as to test their equality (rather simply testing whether they are equal).

Execution of an oracle-aided program corresponds to its program graph by adapting the above correspondence in the straight forward manner:

- The first two nodes on a path are always labelled as $\pi(0), \pi(1)$, respectively; that is, they are used to load the constants 0 and 1. The third node on a path is used to load the (single) input $\pi(y)$ to the program. It is labelled with a special label ϕ. The program stores the intermediate results $y_0 = 0, y_1 = 1, y_3 = \pi(y)$ for these vertices, respectively, and continues execution along this path.
- When the program encounters a non-branching vertex v:
  - If v is labelled with $(i, j, \circ, b)$, where i, j are indices and $b \in \{0, 1\}$, and this is the kth non-branching vertex on the path of traversal for some $k \geq 4$, and, then the program invokes the oracle op_π on input $(y_i, y_j, \circ, b)$. It stores the output of op_π as y_k.
  - If v is labelled with σ and this is the kth non-branching index on the path of traversal for some $k \geq 4$, store $y_k \leftarrow \sigma$ and continue the execution of the program along this path.
- If the program encounters a branching vertex v: if v is labelled (i, j), the program invokes the oracle eq on input (y_i, y_j). If the result is 1, the program continues its execution along the right edge; otherwise, along the left.
- If k is the last vertex on the path, the program outputs y_k and terminates.

Types of Programs. We define two types of programs:

Definition 5. *A T-step (possibly oracle-aided)* straight line program (SLP) S over $\mathbb{Z}_N$ *is a program whose program graph over $\mathbb{Z}_N$ is a labelled path $v_0, \ldots, v_{T+3}$.*

A deterministic generic ring algorithm (GRA) is a generalization of SLPs that allows equality tests. As explained above, such queries are represented as branching vertices in our graph representation of a GRA. Thus, an SLP can be seen as special case GRA, where an SLP is a GRA that contains no branching vertices.

Definition 6. *A T-depth deterministic (possibly oracle-aided)* generic ring algorithm (GRA) G over $\mathbb{Z}_N$ *is a program whose program graph over $\mathbb{Z}_N$ is a depth-$(T+3)$ vertex-labelled and partially edge-labelled binary tree.*

To keep the distinction between oracle aided vs. regular programs clear, we will always make the dependency on π explicit by superscripting oracle-aided programs with π, i.e., G^π.

The following definition applies only to non-oracle aided programs. It inductively defines the polynomial corresponding to an execution of a program on input $x \in \mathbb{Z}_N$. Essentially, if the program encounters a non-branching vertex v and v corresponds to an arithmetic operation, then we associate the resulting tuple $(P_v^G(x), Q_v^G(x))$ with vertex v. Here, $P_v^G(x)$ and $Q_v^G(x)$ are interpreted as the numerator and denominator of a rational function $P_v^G(x)/Q_v^G(x)$ that is the result of applying the arithmetic operation to the rational functions associated with prior vertices w, u.

Definition 7. *For a GRA G (or SLP S) over $\mathbb{Z}_N$ of size T and non-branching vertex v in its execution graph, the pair $(P_v^G(x), Q_v^G(x))$ of polynomials in $\mathbb{Z}_N[x]$* associated with v *is defined inductively, as follows:*

1. *The root has associated the pair* $(0,1)$*, the child of the root the pair* $(1,1)$*, and the child of that child has the pair* $(x,1)$*.*
2. *A vertex* v *labelled with* $a \in \mathbb{Z}_N$ *is associated with* $(a,1)$*.*
3. *For each non-branching vertex* v*, labelled with operation* $(u,w,+,b)$*, we have:*

$$(P_v^G(x), Q_v^G(x)) := \begin{cases} (P_u^G(x) \cdot Q_w^G(x) + P_w^G(x) \cdot Q_u^G(x), Q_u^G(x) \cdot Q_w^G(x)) & b = 1 \\ (P_u^G(x) \cdot Q_w^G(x) - P_w^G(x) \cdot Q_u^G(x), Q_u^G(x) \cdot Q_w^G(x)) & b = -1 \end{cases}$$

4. *For each non-branching vertex* v*, labelled with operation* $(u,w,\cdot,b)$*, we have:*

$$(P_v^G(x), Q_v^G(x)) := \begin{cases} (P_u^G(x) \cdot P_w^G(x), Q_u^G(x) \cdot Q_w^G(x)) & b = 1 \\ (P_u^G(x) \cdot Q_w^G(x), Q_u^G(x) \cdot P_w^G(x)) & b = -1, Q_u^G(x) \neq 0 \pmod N \\ \perp & b = -1, Q_u^G(x) = 0 \pmod N \end{cases}$$

Note that $P_v^G(x)$ *and* $Q_v^G(x))$ *can each be represented as an SLP of size at most* T*.*

Definition 8. *For an SLP* S *over* $\mathbb{Z}_N$ *of size* T*, we denote by* $(P^S(x), Q^S(x))$ *the pair of polynomials in* $\mathbb{Z}_N[x]$ associated with the final vertex on the evaluation path. *If* $Q^S(x) \equiv 1$*, we denote by* f_S *the polynomial* $P^S(x)$*. Note that* $P^S(x)$ *and* $Q^S(x))$ *can each be represented as an SLP of size at most* T*.*

Definition 9. *For each non-branching vertex* v *in the program graph over* $\mathbb{Z}_N$ *of an* ℓ*-step GRA* G *with corresponding pair of polynomials* $(P_v^G(a), Q_v^G(a))$*, we associate the function*

$$f_v^G : \mathbb{Z}_N \to \mathbb{Z}_N \cup \{\perp\} : a \mapsto \frac{P_v^G(a)}{Q_v^G(a)}$$

where the function is undefined if $Q_v^G(a) = 0$*, which is denoted as* $f_v^G(a) = \perp$*, and where* $P_v^G(a)$ *and* $Q_v^G(a)$ *are evaluated over* $\mathbb{Z}_N$*. Moreover, for an argument* $a \in \mathbb{Z}_N$*, the computation path from the root* v_0 *to a leaf* $v_{\ell+3}(a)$ *is defined by taking, for each equality test of the form* (u,w)*, the edge labeled* 0 *if* $f_u^G(a) = f_w^G(a)$*, and the edge labeled* 1 *if* $f_u^G(a) \neq f_w^G(a)$*. The partial function* f^G *computed by* G *is defined as*

$$f^G : \mathbb{Z}_N \to \mathbb{Z}_N \cup \{\perp\} : a \mapsto f_{v_{\ell+3}(a)}^G.$$

We define the output of G *on input* $x \in \mathbb{Z}_N$ *as* $G(x) := f^G(x)$*.*

3.3 Model Specific Versions of the RSA Assumption

We introduce a new variant of the RSA game with preprocessing model specifically tailored to the oracle-aided computational models from the previous section. In the following, we fix the security parameter κ and an integer $m \in \mathbb{Z}, m \geq \kappa$.

Definition 10 (Generic RSA Problem with Preprocessing). *For a tuple of algorithms* $\mathsf{A} = (\mathsf{A}_0^\pi, \mathsf{A}_1)$ *and an RSA instance generator* RSAGen*, define experiment* $\mathbf{crsa}^{\mathsf{A}}_{\mathsf{RSAGen}}$ *as follows:*

- **Offline Phase.** *Sample* $\pi \leftarrow \mathsf{FuncInj}[\kappa, m]$*. Run* A_0^π *on input* 1^κ*. Let* st *denote the return value of* A_0^π*.*
- **Online Phase.** *Compute* $(N, e, d) \leftarrow \mathsf{RSAGen}(1^\kappa)$ *and sample* $x \leftarrow \mathbb{Z}_N$*. Run* A_1 *on input* $(N, e, \pi(x^e), \mathsf{st})$ *and let* G^π *denote the output. If* G^π *does not correspond to the description of a GRA, abort. Note that* A_1 *does* not *get access to oracle* π*.*
- **Output Determination.** *Run* G^π *on input* $(N, e, \pi(x^e))$*. When* G^π *outputs* $z \in \{0,1\}^m$*, the experiment evaluates to* 1 *iff* $z = \pi(x)$*.*

Define A*'s advantage relative to* RSAGen *as*

$$\mathsf{Adv}^{\mathbf{crsa}}_{\mathsf{RSAGen}}(\mathsf{A}) = \Pr[\mathbf{crsa}^{\mathsf{A}}_{\mathsf{RSAGen}} = 1] \ .$$

We call $\mathsf{A} = (\mathsf{A}_0^\pi, \mathsf{A}_1)$ *an* (S, T_1, T_2)-generic-RSA-with-preprocessing algorithm (GP-RSA) relative to RSAGen *if* A_0^π *outputs advice strings* st *of size at most* S*,* A_1 *runs in time at most* T_1*, and and any program* G^π *in the output of* A_1 *runs in time at most* T_2*. Note that we require that* $T_1 \geq T_2$*.*

We also give an alternative version of this game in which $\pi \in \mathsf{FuncInj}[\kappa, m]$ and $(N, e, d) \in \mathsf{RSAGen}(1^\kappa)$ are a fixed.

Definition 11 (Fixed Generic RSA Problem with Preprocessing). *Fix integers* $(N, e, d) \in \mathsf{RSAGen}(1^\kappa)$*, let* $\pi \in \mathsf{FuncInj}[\kappa, m]$*, and let* st *be of size at most* S*. Define experiment* $\mathbf{fcrsa}^{\mathsf{A}}_{(N,e,d,\mathsf{st},\pi)}$ *as follows:*

- **Online Phase.** *Sample* $x \leftarrow \mathbb{Z}_N$*. Run* A *on input* $(N, e, \pi(x^e), \mathsf{st})$ *and let* G^π *denote the output. If* G^π *does not correspond to the description of a GRA, abort. Note that* A *does not have oracle access to* π*.*
- **Output Determination.** *Run* G^π *on input* $(N, e, \pi(x^e))$*. When* G^π *outputs* $z \in \{0,1\}^m$*, the experiment evaluates to* 1 *iff* $z = \pi(x)$*.*

Define A*'s advantage relative to* $(N, e, d, \mathsf{st}, \pi)$ *as*

$$\mathsf{Adv}^{\mathbf{fcrsa}}_{(N,e,d,\mathsf{st},\pi)}(\mathsf{A}) = \Pr[\mathbf{fcrsa}^{\mathsf{A}}_{(N,e,d,\mathsf{st},\pi)}(1^\kappa) = 1],$$

We call A *an* (S, T_1, T_2)-fixed-generic-RSA-with-preprocessing (FGP-RSA) algorithm relative to $(N, e, d, \mathsf{st}, \pi)$ *if* A *runs in time at most* T_1*, and and any program* G^π *in the output of* A *runs in time at most* T_2*.*

Note that in the above definition we do not require the advice string st to be output by a preprocessor A_0^π. However, by a standard averaging argument, we obtain the following lemma:

Lemma 2. *Let* $\mathsf{A} = (\mathsf{A}_0, \mathsf{A}_1)$ *be an* (S, T_1, T_2)*-GP-RSA algorithm and suppose that* $\mathsf{Adv}^{\mathbf{crsa}}_{\mathsf{RSAGen}}(\mathsf{A}) \geq \epsilon$*. Then with probability at least* $\epsilon/2$ *over the coins of* RSAGen*, the choice of* π*, and coins of* A_0^π*,* A_0^π *outputs* st *and* RSAGen *outputs* (N, e, d) *s.t.* $\mathsf{Adv}^{\mathbf{fcrsa}}_{(N,e,d,\mathsf{st},\pi)}(\mathsf{A}_1) \geq \epsilon/2$*.*

4 Main Result

Theorem 4. *Let* $\mathsf{A} = (\mathsf{A}_0^{\pi}, \mathsf{A}_1)$ *be an* $(S_r, T_{1,r}, T_{2,r})$*-GP-RSA algorithm relative to* RSAGen*, and let* $\epsilon := \mathsf{Adv}^{\mathbf{crsa}}_{\mathsf{RSAGen}}(\mathsf{A})$*.*

Then there exists a (S_f, T_f)*-factoring algorithm* B *in the random oracle model relative to* RSAGen *such that*

$$\mathsf{Adv}^{\mathbf{fac}}_{\mathsf{RSAGen}}(\mathsf{B}) \in \Omega(\epsilon^3),$$

such that $T_f := \mathsf{poly}(\kappa) \cdot (T_{1,r} + T_{2,r}^5 + \frac{T_{2,r}^{7/2}}{\epsilon^{3/2}})$*, and such that* $S_f := S_r + O(1)$*.*

Remark 2. In the following, we set $\epsilon' := \epsilon/4$.

Remark 3. We slightly abuse notation in the proof and denote by $S_r, T_{1,r}, T_{2,r}, \epsilon'$ *both* of the following: (1) constants $S_r := S_r(\kappa)$, $T_{1,r} := T_{1,r}(\kappa)$, $T_{2,r} := T_{2,r}(\kappa)$, $\epsilon' = \epsilon'(\kappa)$ for a *fixed* setting of security parameter κ and (2) functions $S_r := S_r(\kappa), T_{1,r} := T_{1,r}(\kappa), T_{2,r} := T_{2,r}(\kappa), \epsilon' = \epsilon'(\kappa)$ with respect to an indeterminate κ. For simplicity, in our case analysis below, we assume that one of the following *mutually exclusive* scenarios occur: (1) for sufficiently large κ, $S_r(\kappa) \cdot T_{2,r}(\kappa) \geq \epsilon'(\kappa) \cdot 2^{\kappa}/4$ or (2) for sufficiently large κ, $S_r(\kappa) \cdot T_{2,r}(\kappa) < \epsilon'(\kappa) \cdot 2^{\kappa}/4$. We note that it is also possible that for *infinitely many* κ, $S_r(\kappa) \cdot T_{2,r}(\kappa) \geq \epsilon'(\kappa) \cdot 2^{\kappa}/4$, and *simultaneously*, for *infinitely many* κ, $S_r(\kappa) \cdot T_{2,r}(\kappa) < \epsilon'(\kappa) \cdot 2^{\kappa}/4$. We can still handle this case and show existence of a factoring algorithm with parameters as above. If the above occurs, then the unbounded pre-processing stage of the factoring algorithm will do the following: On fixed input κ, it will run the GP-RSA algorithm exhaustively on all possible random coins and inputs to determine the exact constants $S_r(\kappa), T_{r,2}(\kappa), \epsilon'(\kappa)$. It will then check whether $S_r(\kappa) \cdot T_{r,2}(\kappa) \geq \epsilon'(\kappa) \cdot 2^{\kappa}/4$ or $S_r(\kappa) \cdot T_{r,2}(\kappa) < \epsilon'(\kappa) \cdot 2^{\kappa}/4$. If the former is true, it will append a "0" bit to the preprocessing advice st to tell the online portion of the factoring algorithm to run the factoring algorithm from Section 4.3. If the latter is true, it will append a "1" bit to the preprocessing advice to tell the online portion of the factoring algorithm to run the factoring algorithm from Section 4.4. Thus, the preprocessing advice increases by a single bit (so it still satisfies $S_f = S_r + O(1)$) and the other parameters $T_f, \mathsf{Adv}^{\mathbf{fac}}_{\mathsf{RSAGen}}(\mathsf{B})$ remain the same and therefore satisfy the required constraints.

Remark 4. Note that achieving the desired factoring algorithm when $T_{r,2} \geq 2^{\kappa/10}$ or $\epsilon' \leq 1/2^{\kappa/6}$ is trivial since there is a trivial factoring algorithm that runs in time $T_f = O\left((2^{\kappa/10})^5\right) = O\left(2^{\kappa/2}\right)$, with zero pre-processing and success probability 1, as well as a trivial factoring algorithm that achieves success probability $\Omega\left((2^{-\kappa/6})^3\right) = \Omega\left(2^{-\kappa/2}\right)$ with zero pre-processing and $\mathsf{poly}(\kappa)$ time (which just guesses a random number in $[2^{\kappa/2}]$ as one of the factors of N). We therefore assume WLOG that $T_{r,2} < 2^{\kappa/10}$ and $\epsilon' > 2^{-\kappa/6}$.

Remark 5. We give a comparison here of the bounds we achieve versus those achieved by AM's factoring algorithm. First, we consider our runtime of $T_f := \mathsf{poly}(\kappa) \cdot (T_{1,r} + T_{2,r}^5 + \frac{T_{2,r}^{7/2}}{\epsilon^{3/2}})$, and focus on the $(T_{1,r} + T_{2,r}^5 + \frac{T_{2,r}^{7/2}}{\epsilon^{3/2}})$ part. The first

term's dependence on $T_{1,r}$ is unavoidable, since the factoring algorithm must run the RSA algorithm at least once. The second term of $T_{2,r}^5$ comes from running the $\mathsf{SLPFactoring}^\pi$ algorithm with $M' := \mathsf{poly}(\kappa) \cdot (T_{2,r})^2$. This corresponds exactly to running AM's Algorithm 1 M' number of times, whereas they only run it once. The reason for one of the $T_{2,r}$ factors in M' is that the success probability of AM's Algorithm 1 depended linearly on $1/T_{2,r}$ (the size of the SLP) and we wanted to remove the dependence on $T_{2,r}$ from our factoring algorithm's success probability. The reason for the second $T_{2,r}$ factor is that the success probability of AM's Algorithm 1 also depends linearly on the fraction of roots in the SLP. For them, this is essentially equivalent to the RSA algorithm's success probability. But for us, due to our compression argument, the fraction of roots in the SLP is only guaranteed to be at least $J/N \approx \epsilon/T_{2,r}$. Since we want to remove the dependence on $T_{2,r}$ from the success probability of the factoring algorithm, this accounts for the second factor of $T_{2,r}$ in our runtime. Moving to the third term of $\frac{T_{2,r}^{7/2}}{\epsilon^{3/2}}$, this comes from the runtime of $\mathsf{Alg2AM}$ which is essentially the same as Algorithm 2 of AM. We are able to reduce from $\epsilon^{3/2}$ to $\epsilon^{5/2}$ in the denominator, since we assume that $\epsilon > 1/N$ and since we ignore $\mathsf{polylog}(N) = \mathsf{poly}(\kappa)$ factors in our analysis.

Next we move on to our success probability. We have ϵ^3 compared to linear dependence on ϵ in AM because we only provide a factoring algorithm when a certain event occurs. The event that we consider is only guaranteed to occur with probability ϵ with respect to ϵ-fraction of oracles.

4.1 Notation and Algorithms

We begin by introducing some additional notation, terminology, and useful algorithms.

Recall that at a branching vertex v with label (u, w) in the program graph of a GRA G, the program performs an equality test on rational functions P_u^G/Q_u^G and P_w^G/Q_w^G, evaluated at the input y of the program. As in AM, we refer to such a branching index as *extreme* if the test consistently yields either 0 or 1 for most possible inputs y of the program.

Definition 12 (Extreme Branching Vertex). *Let $\delta \in [0,1]$ and $N \in \mathbb{Z}$. A* (δ, N)-extreme branching vertex *of a GRA G is a branching vertex v labeled with (u, w) such that $\nu_N(P_u^G \cdot Q_w^G - P_w^G \cdot Q_u^G) \in [0, \delta] \cup [1-\delta, 1]$.*

Our next definition defines a property of an element $y \in \mathbb{Z}_N$ with respect to a polynomial f over $\mathbb{Z}_N$. We refer to this property as *negative orientation.*

Definition 13 (Negative Orientation). *Let $\delta \in [0,1]$, $N \in \mathbb{Z}$, and let f be a polynomial with $\nu_N(f) \in [0, \delta]$ (resp. $\nu_N(f) \in [1-\delta, 1]$). We say that y is* (δ, N)-negatively oriented *with respect to f if $f(y) = 0 \pmod N$ (resp. $f(y) \neq 0 \pmod N$).*

We now define algorithm PreGRA^π in Figure 1. Intuitively, the purpose of this algorithm is to turn an oracle-aided GRA G^π into a GRA $\tilde{G}$ that succesfully

computes the eth root x of $y \pmod N$, whenever G^π successfully computes $\pi(x)$ on input $\pi(y)$. We prove this property of PreGRA^π in Lemma 3. A crucial property of PreGRA^π is that it never queries $\pi^{-1}(\sigma_y)$ when run on input σ_y.

Algorithm PreGRA^π

Input: A GRA G^π of size at most T, a label $\sigma_y \in \{0,1\}^\kappa$.
Output: A GRA $\tilde{G}$.

- Traverse the nodes of the execution graph of G^π using a topological ordering. Let v_i be the i-th node.
  - If v_i corresponds to a non-branching vertex with label σ_y, then set the i-th node $\tilde{v}_i$ of $\tilde{G}$ with value Y, where Y is the indeterminant of the GRA $\tilde{G}$.
  - If v_i corresponds to a non-branching vertex with label σ_z, where $\sigma_z \neq \sigma_y$, then label the i-th node $\tilde{v}_i$ of the GRA $\tilde{G}$ with $\pi^{-1}(\sigma_z)$.
  - If v_i is a non-branching vertex labelled with $(v_u, v_w, \circ, b)$, where v_u and v_w are predecessor nodes of v_i (according to the topological order), label $\tilde{v}_i$ with $(\tilde{v}_u, \tilde{v}_w, \circ, b)$.
  - If v_i is a branching vertex corresponding to the equality check of two nodes v_u, v_w, where v_u and v_w are predecessor nodes of v_i (according to the topological order), set $\tilde{v}_i$ as the branching vertex labelled with (v_u, v_w)
- Output $\tilde{G}$.

Fig. 1: Inversion algorithm for GRAs.

Lemma 3. *Let G^π be a T-depth oracle-aided GRA over $\mathbb{Z}_N$, and let $y = x^e \pmod N$. Suppose that $\tilde{G} := \mathsf{PreGRA}^\pi(G^\pi, \pi(y))$ and that $G^\pi(\pi(y)) = \pi(x)$. Then $\tilde{G}$ is a T-depth GRA over $\mathbb{Z}_N$ and $\tilde{G}(y) = x \pmod N$.*

Proof. The claim on the number of steps is immediate. For the second claim, observe that for every intermediate value stored at a v node labeled with $\sigma_z = \pi(z)$ in the program graph of G^π, PreGRA^π stores the value $z \in \mathbb{Z}_N$, unless $z = y \pmod N$. In the latter case, PreGRA^π stores Y. Hence, for every operation $(\circ, b) \in \{+, \cdot\} \times \{-1, 1\}$ or equality check performed by G^π on two labels $\pi(a)$ and $\pi(b)$ at some node v_i of its program graph, $\tilde{G}$ performs the analogous operation on a and b. It follows that if $G^\pi(\pi(y)) = \pi(x)$ then $\tilde{G}(y) = x$. □

We next define the algorithm $\mathsf{DomPath}$ from AM in Figure 2. This algorithm extracts the path most likely to be taken from a GRA G over $\mathbb{Z}_N$, when G is run on a random input y, i.e., it extracts the 'dominating path' in G. Note that when viewing a GRA as its program graph, then the dominating path can be seen as corresponding to an SLP that corresponds to the nodes in this path. $\mathsf{DomPath}$ takes a sensitivity parameter $\delta \in [0, 1]$ that determines how accurate its guess of the dominating path will be. On top of the SLP corresponding to

this path, DomPath also outputs all the SLPs induced by branching vertices encountered along the dominating path. That is, if a node along the dominating path is labeled (u, w), then DomPath, on input G, will include the SLP $P_u^G - P_w^G$ in its output.

Algorithm DomPath

Input: A GRA G over $\mathbb{Z}_N$ of depth at most T, $\delta \in [0, 1]$.
Output: A list of Polynomials $R_1, \ldots, R_{\psi+1}$ over $\mathbb{Z}_N$, where $\psi \leq T - 1$.

- Let G_{ex} be the tree obtained from G by truncating the sub-tree rooted at v for all non-(δ, N)-extreme branching verteces v (in particular, v must be branching). Thus the leaf vertices of G_{ex} are either non-branching or non-extreme branching vertices.
- Consider a traversal of G_{ex} starting from the root and, at the i-th extreme vertex v labeled with (u, w), going to the edge labeled 1 if $\nu_N(f_u^G - f_w^G) \in [0, \delta]$ and to the edge labeled 0 if $\nu_N(f_u^G - f_w^G) \in [1-\delta, 1]$. Let ψ be the number of extreme vertices encountered in the path from the root to a leaf. Note that $\psi \leq T - 1$. For each extreme vertex labeled with (u, w), let R_i denote the polynomial $(P_u^G \cdot Q_w^G - P_w^G \cdot Q_u^G)$. Let $\hat{R}_{\psi+1}$ denote the final SLP formed by the path from the root to the leaf (note that $f^{\hat{R}_{\psi+1}}$ is different from R_ψ, since the leaf is not an extreme branching vertex).
- If the last vertex of $\hat{R}_{\psi+1}$ is a non-branching vertex with $(P^{\hat{R}_{\psi+1}}, Q^{\hat{R}_{\psi+1}})$, then let $R_{\psi+1}(Y) := [P^{\hat{R}_{\psi+1}}(Y)]^e - Y \cdot [Q^{\hat{R}_{\psi+1}}(Y)]^e$.
- If the last vertex of $\hat{R}_{\psi+1}$ is a non-extreme branching vertex labeled with (u, w), then let $R_{\psi+1} := P_u^G \cdot Q_w^G - P_w^G \cdot Q_u^G$.

Fig. 2: Inversion algorithm for GRAs.

Note that for all the SLP's $R_1, \ldots, R_{\psi+1}$ with corresponding pairs $(P^{R_1}(Y), Q^{R_1}(Y)), \ldots, (P^{R_{\psi+1}}(Y), Q^{R_{\psi+1}}(Y))$ returned by DomPath, we have that $Q^{R_1}(Y) \equiv \cdots \equiv Q^{R_{\psi+1}}(Y) \equiv 1$. We therefore denote by $f_{R_1}, \ldots, f_{R_{\psi+1}}$ the polynomials $P^{R_1}(Y), \ldots, P^{R_{\psi+1}}(Y)$.

Finally, we define the algorithm ComGRA^π (see Figure 3). This algorithm internally runs the online phase of an FGP-RSA algorithm to obtain the oracle aided GRA G^π. It then runs PreGRA on its input $\pi(y)$ to obtain a GRA $\tilde{G}$, from which it extracts the dominating path via DomPath.

We prove the following Lemma:

Lemma 4. *Let* A *be an* $(S_r, T_{1,r}, T_{2,r})$*-FGP RSA algorithm relative to* N, e*,* π *and* st*. Suppose that* $\mathsf{Adv}^{\mathbf{fcrsa}}_{N,e,d,\mathsf{st},\pi}(\mathsf{A}) \geq \epsilon/2$ *and let* $\delta \in [0, 1]$*. Then with probability at least* $\epsilon/2$ *over the coins of* $\mathsf{ComGRA}[\mathsf{A}]^\pi$ *and* $y \leftarrow \mathbb{Z}_N$*, at least one of the following two events occurs when running* $\mathsf{ComGRA}[\mathsf{A}]^\pi$ *on input* $\pi(y)$*, and* $\delta \in [0, 1]$*.*

1. $\mathsf{ComGRA}[\mathsf{A}]^\pi$ *returns* $R_1, ..., R_{\psi+1}$ *such that* y *is* (δ, N)*-negatively oriented with respect to* R_i *for some* $i \in [\psi + 1]$*.*

Algorithm $\mathsf{ComGRA}[\mathsf{A}]^{\pi}$

Input: A label $\pi(y)$, a sensitivity parameter $\delta \in [0,1]$. Denote the random coins as ρ.
Output: A list of polynomials $\{R_1, \ldots, R_{\psi+1}\}$ over $\mathbb{Z}_N$.

- Run A on input $(N, e, \pi(y), \mathsf{st})$ and randomness ρ. Let G^{π} denote the output. If G^{π} does not correspond to the description of a T_2-depth GRA over $\mathbb{Z}_N$, abort. Let $\tilde{G} := \mathsf{PreGRA}(G^{\pi}, \pi(y))$.
- Return $\{R_1, \ldots, R_{\psi+1}\} \leftarrow \mathsf{DomPath}(\tilde{G}, \delta)$

Fig. 3: Combined Algorithm. A is an (S, T_1, T_2) FGP-RSA algorithm relative to N, e, st, π.

2. $\mathsf{ComGRA}[\mathsf{A}]^{\pi}$ *returns* $R_1, ..., R_{\psi+1}$ *such that* $\nu_N(R_{\psi+1}) \geq \delta$ *and* $R_{\psi+1} \not\equiv 0 \pmod N$.

Proof. Let G^{π} be the oracle aided GRA output by A on input $N, e, \pi(y), \mathsf{st}$, where $y = x^e \pmod N$. We will show that whenever the algorithm is successful (i.e. $G^{\pi}(\pi(y)) = \pi(x)$) then the conclusion of Lemma 4 must hold. This is sufficient to prove the lemma.

Let $\tilde{G} = \mathsf{PreGRA}(G^{\pi}, \pi(y))$. Consider the execution of $\mathsf{DomPath}(\tilde{G})$ and recall that for each extreme vertex labeled with (u, w), R_i denotes the SLP

$$(P_u^{\tilde{G}} \cdot Q_w^{\tilde{G}} - P_w^{\tilde{G}} \cdot Q_u^{\tilde{G}}).$$

Finally, recall that $\hat{R}_{\psi+1}$ denotes the final SLP formed by the path from the root to the leaf.

If the last vertex of $\hat{R}_{\psi+1}$ is a non-extreme branching vertex labeled with (u, w), then $R_{\psi+1}$ is defined as $R_{\psi+1} = P_u^{\tilde{G}} \cdot Q_w^{\tilde{G}} - P_w^{\tilde{G}} \cdot Q_u^{\tilde{G}}$ and by definition of non-extreme branching vertex, $\nu_N(R_{\psi+1}) \in (\delta, 1-\delta)$. This implies that $\nu_N(R_{\psi+1}) \geq \delta$ and $R_{\psi+1} \not\equiv 0$. (If $R_{\psi+1} \equiv 0$ then $\nu_N(R_{\psi+1}) = 1$.). So the conclusion of Lemma 4 holds.

We therefore assume w.l.o.g. that the last vertex of $\hat{R}_{\psi+1}$ is a non-branching vertex — in which case $R_{\psi+1}$ is defined as $R_{\psi+1} = [P^{\hat{R}_{\psi+1}}(Y)]^e - Y \cdot [Q^{\hat{R}_{\psi+1}}(Y)]^e$ — *and* that the algorithm is successful (i.e. $G^{\pi}(\pi(y)) = \pi(x)$).

Case 1: $\hat{R}_{\psi+1}(y) = x$**.** If $\nu_N(R_{\psi+1}) \geq \delta$ then the conclusion of Lemma 4 holds. Otherwise, $\nu_N(R_{\psi+1}) \in (0, \delta)$. In this case, y is a root of $R_{\psi+1}$, since by Lemma 3, $P^{\hat{R}_{\psi+1}}(y)/Q^{\hat{R}_{\psi+1}}(y) = x$ and $R_{\psi+1}(Y) = [P^{\hat{R}_{\psi+1}}(Y)]^e - Y \cdot [Q^{\hat{R}_{\psi+1}}(Y)]^e$. Moreover, $R_{\psi+1}(Y) \not\equiv 0$, which was shown by Aggarwal and Maurer [1] as part of the proof for their Corollary 1. Hence, y is negatively oriented with respect to $R_{\psi+1}$, and again the conclusion of Lemma 4 holds.

Case 2: $\hat{R}_{\psi+1}(y) \neq x$**.** Since the algorithm is successful, $G^{\pi}(\pi(y)) = \pi(x)$ and due to Lemma 3, we have that $\tilde{G}(y) = x$. Note that $\tilde{G}(y) = x$ but $\hat{R}_{\psi+1}(y) \neq x$ and the last vertex of $\hat{R}_{\psi+1}$ is a non-branching vertex. Hence,

the execution of $\tilde{G}(y)$ takes a different branch than the execution of $\hat{R}_{\psi+1}(y)$ at some extreme branching vertex on the path from the root to the leaf corresponding to $\hat{R}_{\psi+1}$. But then this means that for some $i \in [\psi]$, either $\nu_N(R_i) \in (0, \delta)$ and $R_i(y) = 0$ or $\nu_N(R_i) \in (1-\delta, 1)$ and $R_i(y) \neq 0$. By definition, this implies that for some $i \in [\psi]$, y is *negatively oriented* with respect to R_i, so again the conclusion of Lemma 4 holds.

This concludes the proof of Lemma 4. □

Two Events. Fix A, N, e, π and st as in Lemma 4. We consider the probability of two events over the randomness of ComGRA and choice of $y \leftarrow \mathbb{Z}_N$. Set $J := \frac{(1-\epsilon'/2)\epsilon' \cdot N}{8 \log N T_{2,r}} = \frac{(1-\epsilon'/2) \cdot N}{4 R_1 T_{2,r}} = N \cdot \delta$, where $\delta := J/N$.

- Event $E[N, e, \mathsf{st}, \pi]_1$: $\mathsf{ComGRA}[\mathsf{A}]^\pi$ on input $\pi(y)$ returns a list of polynomials $\{R_1, \ldots, R_{\psi+1}\}$ s.t. y is *negatively oriented* with respect to one of $\{R_1, \ldots, R_{\psi+1}\}$.
- Event $E[N, e, \mathsf{st}, \pi]_2$: $\mathsf{ComGRA}[\mathsf{A}]^\pi$ on input $\pi(y)$ returns a list of polynomials $\{R_1, \ldots, R_{\psi+1}\}$ s.t. $\nu_N(R_{\psi+1}) \in (\delta, 1-\delta)$.

By Lemma 4, we obtain the following corollary:

Corollary 1. *Suppose that the conditions of Lemma 4 hold. Then at least one of the events $E[N, e, \mathsf{st}, \pi]_1$ or $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least $\epsilon/4$.*

Looking ahead, if $E[N, e, \mathsf{st}, \pi]_1$ occurs, then A will be useless for factoring. Our task, therefore, is to prove that $E[N, e, \mathsf{st}, \pi]_1$ occurs with probability less than $\epsilon' = \epsilon/4$ (which we do in the following Section 4.2 via a compression argument). We can therefore conclude that $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least $\epsilon' = \epsilon/4$.

4.2 Bounding the Probability of Event $E[N, e, \mathsf{st}, \pi]_1$ when $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa / 4$

In this section, we upper bound the probability of the event $E[N, e, \mathsf{st}, \pi]_1$ as defined in the previous section, given that $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa / 4$, where $\epsilon' := \epsilon/4$. In particular, we fix the values of N, e, st, π throughout most of this section. We also fix the length of the labels and interpret our labelling function as an injective mapping $\pi : \mathbb{Z}_N \to \mathbb{Z}_L$, where $L \geq N$ is chosen of appropriate size.

To achieve our upper bound, we will construct an encoding routine $\widetilde{\mathsf{Enc}^\pi}$ (which is itself a "wrapped" version of Enc^π that includes the RSA instance generation and preprocessing steps)[6] that compresses the function table of π

[6] Separating them is convenient because otherwise, we couldn't keep N fixed while arguing about Enc^π. We also comment that $\widetilde{\mathsf{Enc}^\pi}$ uses more randomness than Enc^π and Dec. So when the non-wrapped routines read their shared random string ρ, they should start at the position corresponding to the number of random bits used by $\widetilde{\mathsf{Enc}^\pi}$.

whenever the event $E[N, e, \mathsf{st}, \pi]_1$ is likely to happen. We also present a corresponding decoding routine Dec. Together, $\widetilde{\mathsf{Enc}^{\pi}}, \mathsf{Dec}$ will lead to a contradiction of Lemma 1, given $E[N, e, \mathsf{st}, \pi]_1$ happens with too large of a probability. We first present our encoding routine $\widetilde{\mathsf{Enc}^{\pi}}$ and decoding routine Dec in Figures 4, 5, and 6 and argue their correctness. Here we set $r_1 := \lceil 2 \log N/\epsilon' \rceil$, $r_2 := \lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor$, and J to be the maximum integer that satisfies $J \leq \frac{\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}) \rfloor}{8R_1}$ and $r_1 J T_{2,r}$ is a power of two. Note that we can lower bound J by $J \geq \frac{(1-\epsilon'/2)N}{32 r_1 T_{2,r}}$.

Algorithm Enc^{π}

Interface: Enc^{π} takes as input (N, e, st). Denote Enc^{π}'s random coins as ρ. Enc^{π} outputs an encoding E of the labelling function $\pi : \mathbb{Z}_N \to \mathbb{Z}_L$.
Initialization: Initialize a set $\mathsf{E} = \{\mathsf{st}, N, e\}$ and an (empty) table Table that stores rows of the form $(x, \pi(x))$ for $x \in \mathbb{Z}_N$. Split random tape ρ into $\rho[1]||\rho[2]$ of appropriate size. Let $\mathcal{I} \subseteq \mathbb{Z}_L$ denote the image of $\pi : \mathbb{Z}_N \to \mathbb{Z}_L$. Add to E an encoding of $\mathcal{I}$ (of length $\log \binom{L}{N}$).

Set $j := 0$. Repeat the following steps while $j < r_2$:

- Set $\mathsf{good} := \mathsf{false}$. Parse $\rho[1]$ as $\rho[1] = \rho[1,1], ..., \rho[1, r_1 \cdot r_2]$ and $\rho[2]$ as $\rho[2] = \rho[2,1], \ldots, \rho[2, r_1 \cdot r_2]$.
- Set $k := 0$ and repeat the following steps while $k \leq r_1$ and $\neg\mathsf{good}$:
  - Set $k := k + 1$.
  - Use $\rho[1, j \cdot r_1 + k]$ to select a random image $\pi(y_k)$ from the images that are not yet contained in Table.
  - Run $\mathsf{ComGRA}^{\pi}[\mathsf{A}_1]$ on random coins $\rho[2, j \cdot r_1 + k]$ and inputs $\delta, \pi(y_k)$.
  - If $\mathsf{ComGRA}^{\pi}[\mathsf{A}_1]$ returns $\{R_1, \ldots, R_{\psi+1}\}$ such that y_k is negatively oriented with respect to R_ζ for some $\zeta \in [\psi + 1]$, set $\mathsf{good} := \mathsf{true}$.
- If $\neg\mathsf{good}$, abort. (Call this Failure Event 2.1; we will show it occurs with probability at most 1/2.)
- Denote by $1 \leq \ell \leq J$ the index of y_k among the J roots or non-roots of R_ζ, depending on which case of Definition 13 R_ζ, y_k falls into.
- Add to E the entry (ζ, ℓ, k). Re-run $\mathsf{ComGRA}^{\pi}[\mathsf{A}_1]$ on random coins $\rho[2]_{j \cdot r_1 + k}$ and input $\delta, \pi(y_k)$. If $\mathsf{ComGRA}^{\pi}[\mathsf{A}_1]$ internally queries (during the execution of PreGRA^{π}) $\pi^{-1}(z)$ s.t. $(\pi^{-1}(z), z)$ is not yet stored in Table add to E the trivial encoding of $\pi^{-1}(z)$ (of length $\log(N - |\mathsf{Table}|)$) and add $(\pi^{-1}(z), z)$ to Table. Add $(y_k, \pi(y_k))$ to Table.
- Set $j := j + 1$.

At this point there is a set of pre-images S and images S' stored in Table. Add an encoding to E of π restricted to $(\mathbb{Z}_N \setminus S) \to (\mathcal{I} \setminus S')$.

Fig. 4: Non-wrapped encoding routine.

Algorithm Dec

Interface: Dec takes as input an encoding E and random coins ρ. It outputs the function table $\mathsf{Table}[\pi]$ of a function $\pi \in \mathsf{FuncInj}$.

- Initialize $\mathsf{Table}[\pi] = \perp$ and $\mathsf{done} := \mathsf{false}$.
- Recover from E the image of π and add it to $\mathsf{Table}[\pi]$.
- Interpret random coins ρ as $\rho_{\mathsf{RSAGen}}, \rho_{A_0}, \rho^{\pi}_{\mathsf{Enc}}$.
- Split $\rho^{\pi}_{\mathsf{Enc}}$ into two parts $\rho^{\pi}_{\mathsf{Enc}}[1]||\rho^{\pi}_{\mathsf{Enc}}[2]$ and parse these parts as $\rho^{\pi}_{\mathsf{Enc}}[1] = \rho^{\pi}_{\mathsf{Enc}}[1,1], ..., \rho^{\pi}_{\mathsf{Enc}}[1, r_1 \cdot r_2]$ and $\rho^{\pi}_{\mathsf{Enc}}[2] = \rho^{\pi}_{\mathsf{Enc}}[2,1], ..., \rho^{\pi}_{\mathsf{Enc}}[2, r_1 \cdot r_2]$.
- Compute $\mathsf{st} = A_0^{\pi}(1^{\kappa}; \rho_{A_0})$ and $(N, e, d) = \mathsf{RSAGen}(1^{\kappa}; \rho_{\mathsf{RSAGen}})$. Let $\delta := \frac{(1-\epsilon'/2)}{4r_1 T_{2,r}}$.
- While $\neg\mathsf{done}$ do:
  - Find the next tuple $t = (\zeta, \ell, k)$ in E. If this is the jth tuple of this form, set $m \leftarrow r_1 \cdot j + k$.
  - Use random coins $\rho^{\pi}_{\mathsf{Enc}}[1, m]$ to select a point $\pi(y)$ in the image of π.
  - Run $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$ on input $(\delta, \pi(y))$ with random coins $\rho^{\pi}_{\mathsf{Enc}}[2, m]$.
  - If $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$ queries π on input x, find $\pi(x)$ in E and return it to $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$.
  - When $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$ returns $\{R_1, \ldots, R_{\psi+1}\}$, find the ℓth root y of R_{ζ}.
  - Add to $\mathsf{Table}[\pi]$ the entry $(y, \pi(y))$.
  - Remove t from E. If no further tuple of the above form exists in E, set $\mathsf{done} := \mathsf{true}$.
- Add all remaining preimages stored in E to the appropriate positions in $\mathsf{Table}[\pi]$.
- Return $\mathsf{Table}[\pi]$

Fig. 5: Our decoding routine.

Lemma 5. *Suppose that* $\widetilde{\mathsf{Enc}^{\pi}}$ *with access to* π *and on random coins* ρ *outputs* E. *Then* Dec *on input* E *and on random coins* ρ *outputs the function table* $\mathsf{Table}[\pi]$ *of* π.

Proof. The lemma follows by construction of Enc^{π} and Dec. Specifically, Enc^{π} stores one tuple of the form (ζ, ℓ, k) per iteration of the outer loop. As Enc^{π} stores the tuples E in the order in which they are found, it follows that Dec can deterministically recover the tuple corresponding to the jth iteration of the outer loop. Since both algorithms parse the random tape $\rho^{\pi}_{\mathsf{Enc}}[2]$ in the same manner, Dec can also recover the proper index $m = r_1 \cdot j + k$ and (via $\rho^{\pi}_{\mathsf{Enc}}[1, m]$) the image $\pi(y)$ in order to run $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$. Moreover, Enc^{π} ensures that any call x to π that $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$ makes during its run can be answered by looking up the pair $(x, \pi(x))$ in E. Hence, Dec obtains from $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$ the same list of polynomials $\{R_1, \ldots, R_{\psi+1}\}$ as Enc^{π} does. It can now identify the correct polynomial R_{ζ} among them and use the ℓ root y as the preimage y of $\pi(y)$ in order to complete the pair $(y, \pi(y))$ in $\mathsf{Table}[\pi]$. Once Dec finds no further points, it can easily recover the remaining points of the function table by using the trivial encoding provided by Enc^{π}.

We next present the main technical lemma of this subsection. The proof of this lemma is deferred to Appendix A. Combining it with Lemma 7 below, we get that the encoding routine is *compressing* with high probability.

Lemma 6. *Let* $S_r \leq \epsilon' 2^{\kappa}/(4T_{2,r})$ *and fix some* N, e, π *and* st *of size at most* S_r. *Let* A *be an* $(S_r, T_{1,r}, T_{2,r})$*-FGP RSA algorithm. Suppose that* $\mathsf{Adv}^{\mathbf{fcrsa}}_{(N,e,\mathsf{st},\pi)}(\mathsf{A}) \geq \epsilon/2$ *and* $\Pr[E[N, e, \mathsf{st}, \pi]_1] \geq \epsilon'$ *(over the random coins of* $\mathsf{ComGRA}[\mathsf{A}_1]^{\pi}$ *and random choice of* $y \sim \mathbb{Z}_N$*). Then with probability at least* $1/2$ *over the coins of* Enc^{π}, Enc^{π}, *on input* (N, e, st), *returns* E *of size at most* $\log\binom{L}{N} + \log(N!) + 2\log(N) - \epsilon' N/(2T_{2,r}) + 2$.

Algorithm $\widetilde{\mathsf{Enc}^{\pi}}$

Interface: $\widetilde{\mathsf{Enc}^{\pi}}$ takes in random coins ρ. It outputs an encoding E of π.

- Parse ρ as $\rho = (\rho_{\mathsf{RSAGen}}, \rho_{A_0}, \rho_{\mathsf{Enc}})$.
- Compute $\mathsf{st} = A_0^{\pi}(1^{\kappa}; \rho_{A_0})$ and $(N, e, d) = \mathsf{RSAGen}(1^{\kappa}; \rho_{\mathsf{RSAGen}})$.
- Obtain an encoding E as $\mathsf{E} \leftarrow \mathsf{Enc}^{\pi}(\mathsf{st}, N, e; \rho_{\mathsf{Enc}})$. If Enc^{π} aborts, abort.
- Output E.

Fig. 6: Our final, wrapped encoding routine.

Lemma 7. *Let* RSAGen *be an RSA generator and let* $\mathsf{A} = (\mathsf{A}_0, \mathsf{A}_1)$ *be an AG-RSA algorithm with* $\mathsf{Adv}^{\mathbf{crsa}}_{\mathsf{RSAGen}}(\mathsf{A}) \geq \epsilon$ *s.t.* $S_r \cdot T_{2,r} < 2^{\kappa} \cdot \epsilon/16$. *Then the following happens with probability less than* $\epsilon/2$ *over the choice of* π, *the random coins of*

A_0, *and the random coins of* RSAGen*:* A_0^π *outputs* st *of size at most* S_r *and* RSAGen *outputs* (N, e, d) *s.t.* $E[N, e, \mathsf{st}, \pi]_1$ *occurs with probability at most* $\epsilon/4$ *(over the random coins of* $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ *and random choice of* $y \sim \mathbb{Z}_N$*).*

Proof. Let A be as in the lemma statement and assume toward a contradiction that with probability at least $\epsilon/2$ (over their internal coins and choice of π), A_0^π and RSAGen output st and (N, e, d) s.t. $E[N, e, \mathsf{st}, \pi]_1$ occurs with probability at least $\epsilon' = \epsilon/4$ over the randomness of $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ and random choice of $y \sim \mathbb{Z}_N$. Then by Lemma 6, given that π, st, and N, e are such that $E[N, e, \mathsf{st}, \pi]_1$ occurs with probability at least $\epsilon' = \epsilon/4$ over the internal coins of $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ and random choice of $y \sim \mathbb{Z}_N$, we have that, for sufficiently large κ, Enc^π outputs an encoding E of size at most $\log\binom{L}{N} + \log(N!) + 2\log(N) - \epsilon' N/(2T_{2,r}) + 2$ with probability at least $1/2$ over its choice of random coins. Moreover if Enc^π returns E, then running Dec on E reproduces the function table of π with probability 1 (when run with the same coins). Now consider the algorithm $\widetilde{\mathsf{Enc}^\pi}$ depicted in Figure 5. $\widetilde{\mathsf{Enc}^\pi}$ gets access to π and internally runs A_0 and RSAGen on input 1^κ to obtain st of size at most S_r and (N, e, d), respectively. It then runs Enc^π on input (N, e, st) with access to π. It returns the encoding E returned by Enc^π and aborts in case Enc^π aborts. We can view $(\widetilde{\mathsf{Enc}^\pi}, \mathsf{Dec})$ as a pair of encoding/decoding routines that produce, for any input π, an encoding E which successfully decodes to π with probability at least ϵ' over the random coins of $\widetilde{\mathsf{Enc}^\pi}$. We now use Lemma 1 where we set $E = \mathsf{Enc}^\pi$ (here we give π as an oracle, but this is equivalent to giving it as input as in the Lemma since Enc^π can make an unbounded number of queries), $D = \mathsf{Dec}$, and $\mathcal{X} = \{0,1\}^{\log(\frac{L!}{(L-N)!})}$, $m = \log(N!) + 2\log(N) - \epsilon' N/(2T_{2,r}) + 2$, and $\gamma = \epsilon'$. Note that our choice of $\mathcal{X}$ is large enough to store the function table of π. Lemma 1 says that $\log\binom{L}{N} + \log(N!) + 2\log(N) - \epsilon' N/(2T_{2,r}) + 2 = m \geq \log|\mathcal{X}| - \log 1/\gamma = \log(N!) - 2 + \log(\epsilon')$. Since $\log\binom{L}{N} + \log(N!) = \log(\frac{L!}{(L-N)!}) = \log|\mathcal{X}|$, we arrive at a contradiction whenever $\epsilon' > 16N^2/2^{\epsilon' N/(2T_{2,r})}$. Since we have assumed that $\epsilon' > 2^{-\kappa/6} \geq (1/(2N)^{1/6})$ and $T_{2,r} < 2^{\kappa/10} \leq (2N)^{1/10}$, we indeed have that for sufficiently large κ, $\epsilon' > 16N^2/2^{\epsilon' N/(2T_{2,r})} \geq 16N^2/2^{N^{3/5}/2^{19/15}} > 16N^2/2^{\epsilon' N/(2T_{2,r})}$, which yields the desired contradiction.

4.3 Constructing a Factoring Algorithm in the RO Model when $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa/4$

Recall that in Lemma 4 we showed that, for properly generated N, e, st, π, at least one of the events $E[N, e, \mathsf{st}, \pi]_1$, $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least $\epsilon/4$. Further, in Lemma 7 we showed that for a large fraction $\epsilon/2$ of N, e, st, π, the event $E[N, e, \mathsf{st}, \pi]_1$ occurs with probability at most $\epsilon/4$, when $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa/4$. This means that (for $\epsilon/2$-fraction of N, e, st, π) event $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least $\epsilon/4 = \epsilon'$, when $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa/4$. In this subsection, we will first present a factoring algorithm in the RI model that succeeds when event $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least ϵ'. Put together with Lemmas 4

and 7, this means that the factoring algorithm presented in this section is guaranteed to succeed with high probability when $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa/4$. Looking ahead, at the end of this section, we will show how to switch the algorithm from the RI model (with backwards and forwards access to the random injective function) to the RO model. In Section 4.4, we will present a completely different factoring algorithm in the RO model that succeeds when $S_r \cdot T_{2,r} > \epsilon' 2^\kappa/4$.

We begin by recalling Algorithm 2 (denoted Alg2AM) from Aggarwal and Maurer in Figure 7.

Algorithm Alg2AM

Input: A GRA G over $\mathbb{Z}_N$.
Output: A factor of N or an SLP S over $\mathbb{Z}_N$.

- Let $v_1, ..., v_4$ be the first 4 nodes on a path from the root of G.
- Initialize S to be a path of length 2 with v_1, v_2, v_3. Let $v = v_4$.
- While $v.right$ is defined do:
  - If v is a non-branching vertex, then append v with its label to S^π.
  - Else, let the label of v be (u, w).
  - For $i \leftarrow 1$ to $M := \log(N) \cdot \delta^{-3/2}$ do:
    - Generate a uniformly random element $x \in \mathbb{Z}_N$
    - Compute g as the gcd of $P_u^G(x) \cdot Q_w^G(x) - P_w^G(x) \cdot Q_u^G(x)$ and N
    - If $g \notin \{1, N\}$, then return g
  - Generate a uniformly random element $x' \in Z_N$
  - If $P_u^G(x') \cdot Q_w^G(x') - P_w^G(x') \cdot Q_u^G(x') = 0$ then $v := v.left$
  - Else, $v := v.right$
  - If the final vertex of S originated from a non-branching vertex v, let (P^S, Q^S) denote the SLP's computed by the path S from the root to v. Let $S_{\psi+1}(Y) := [P^S(Y)]^e - Y \cdot [Q^S(Y)]^e$. I.e. $S_{\psi+1}(Y)$ is the SLP obtained by exponentiating the outputs of SLP's $P^S(Y)$ and $Q^S(Y)$ (which can be done in $O(\log(e))$ steps, mutliplying the second by indeterminate Y (which can be done in a single step) and subtracting the two (which can be done in a single step).
  - If the final vertex of S originated from a non-extreme branching vertex labeled with (u, w), set $S_{\psi+1} := P_u^G \cdot Q_w^G - P_w^G \cdot Q_u^G$. I.e. $S_{\psi+1}$ is the SLP obtained by multiplying the outputs of SLP's P_u^G, Q_w^G and P_w^G, Q_u^G (which can be done in two steps) and subtracting the two (which can be done in a single step).
  - Else, set $S_{\psi+1} := \perp$.
- Return $S_{\psi+1}$.

Fig. 7: Algorithm 2 from Aggarwal Maurer

This algorithm runs in polynomial time and takes as input a GRA G and an integer N. It outputs either a non-trivial factor of N or an SLP S with many roots. In the former case, we are done. In the latter case, the idea is to run

Algorithm $\mathsf{SLPFactoring}^{\pi}$ (see Appendix B) on input S and N, which similarly produces a non-trivial factor of N in polynomial running time. $\mathsf{SLPFactoring}^{\pi}$ corresponds to Algorithm 1 of Aggarwal and Maurer [1]. The only difference is that we repeat their algorithm M' times with independent random coins in order to improve the success probability. For our purposes, we will set the parameter M' as $M' := \log N \cdot (T_{2,r})^2$. We next consider a simple augmentation of $\mathsf{Alg2AM}$ in Figure 7. For sake of simplicity, in the following, we refer to an SLP S as *functionally equivalent* to a polynomial R if for all $x \in \mathbb{Z}_N$, $S(x) = R(x) \pmod N$. Our main reason for distinguishing SLPs from polynomials is because an SLP has an efficient representation. Note that this need not be true for a polynomial in general.

Lemma 8. *Let G^{π} be a GRA, let $y = x^e$, let $\tilde{G} = \mathsf{PreGRA}^{\pi}(G^{\pi}, \pi(y))$, and let $\{R_1, \ldots, R_{\psi+1}\}$ be the list of polynomials returned by $\mathsf{DomPath}(\tilde{G})$. Then with probability $1 - T_{2,r} \cdot \delta = 1 - T_{2,r} \cdot \frac{(1-\epsilon'/2)\epsilon'}{32\kappa T_{2,r}} \geq 1 - \epsilon/8$ over the random coins of $\mathsf{Alg2AM}$, $\mathsf{Alg2AM}$ on input $\tilde{G}$ outputs a non-trivial factor g of N or outputs an SLP $S_{\psi+1}$ s.t. $S_{\psi+1}$ is functionally equivalent to $R_{\psi+1}$.*

Proof. We consider the set $\mathcal{V}$ of all vertices v encountered during a run of $\mathsf{Alg2AM}$. Let p_1 denote the probability that there is some $v \in \mathcal{V}$ that is a non-extreme branching vertex. Let $p_2 = (1 - p_1)$ denote the probability that all $v \in \mathcal{V}$ are either non-branching vertices or extreme branching vertices. Let p_3 denote the probability that $\mathsf{Alg2AM}$ outputs a factor g conditioned on all $v \in \mathcal{V}$ being either non-branching vertices or extreme branching vertices. If there is some $v \in \mathcal{V}$ that is a non-extreme branching vertex then (invoking Lemma 2 of Aggarwal and Maurer [1]) $\mathsf{Alg2AM}$ outputs a factor g with probability at least $1-(1-\delta^{3/2})^M$. Further, conditioned on (1) all $v \in \mathcal{V}$ being either non-branching vertices or extreme branching vertices, and (2) $\mathsf{Alg2AM}$ not outputting a factor g, we have that $S_{\psi+1}$ is functionally equivalent to $R_{\psi+1}$ with probability at least $1 - T_{2,r}\delta$. This follows directly from the fact that $\mathsf{Alg2AM}$ performs an identical traversal of the nodes in $\tilde{G}$'s execution graph as does $\mathsf{DomPath}(\tilde{G})$.

Thus, the overall success probability is at least

$$\begin{aligned}
&p_1 \cdot (1 - (1 - \delta^{3/2})^M) + p_2 \cdot (p_3 + (1 - p_3)(1 - T_{2,r}\delta)) \\
&\geq p_1 \cdot (1 - (1 - \delta^{3/2})^M) + p_2(1 - T_{2,r}\delta) \\
&= p_1 \cdot (1 - (1 - \delta^{3/2})^M) + (1 - p_1)(1 - T_{2,r}\delta) \\
&\geq (1 - T_{2,r}\delta),
\end{aligned}$$

where the final inequality follows due to setting parameter $M = \log(N) \cdot \delta^{-3/2}$ so that $1 - (1 - \delta^{3/2})^M \geq 1 - e^{-M\delta^{3/2}} \geq (1 - 1/N) \geq (1 - T_{2,r}\delta)$.

Lemma 9. *Let $A = (A_0, A_1)$ be an $(S_r, T_{1,r}, T_{2,r})-$ FGP RSA algorithm. Fix (N, e, π), let $\mathsf{st} \in A_0^{\pi}(1^{\kappa})$ and suppose that $S_r \cdot T_{2,r} \leq 2^{\kappa} \cdot \epsilon'/4$. If $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least ϵ' over the randomness of ComGRA and random $y \sim \mathbb{Z}_N$, then Algorithm $\mathsf{Factoring}^{\pi}$ on input $(N, e), \mathsf{st}$ runs in time*

$O((\kappa^2 \frac{\kappa T_{2,r}}{(1-\epsilon'/2)\epsilon'})^{3/2} + T_{2,r}^5 \kappa^3 + T_{1,r})$ *and outputs a non-trivial factor of* N *with probability* $\Omega((\epsilon')^2(1-\epsilon'/2))$ *over its choice of random coins.*

Algorithm $\mathsf{Factoring}^{\pi}$

Input: A tuple (N, e) and a state st.
Output: A factor g of N.

1. Sample y at random.
2. Run A_1 on input $(N, \pi(y), \mathsf{st})$ and let G^{π} denote the output. If G^{π} does not correspond to the description of a GRA, abort.
3. Run $\tilde{G} := \mathsf{PreGRA}^{\pi}(G)$
4. Run $\mathsf{Alg2AM}(\tilde{G})$. If it returns a factor, return the factor and abort. Otherwise let $S_{\psi+1}$ be the returned SLP.
5. Return the output of $\mathsf{SLPFactoring}^{\pi}$ on input $(S_{\psi+1}, N)$ with $M' := \log N \cdot (T_{2,r})^2$.

Fig. 8: Factoring Algorithm

Algorithm $\widetilde{\mathsf{Factoring}}^{\mathsf{H}}$

Offline Phase: Run A_0 on input 1^{κ} to obtain state st of size S_r. Simulate the oracle for A_0 by using random oracle H as the labelling function.

Online Phase:

- Choose a prime e uniformly at random.
- Return the output of $\mathsf{Factoring}^{\pi}$ run on input $(N, e), \mathsf{st}$. Simulate the oracle π using oracle H.

Fig. 9: Wrapped Factoring Algorithm

Proof. We now analyze the success probability of the above algorithm in the case that $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability at least ϵ'.

First, Lemma 8 implies that if $E[N, e, \mathsf{st}, \pi]_2$ occurs with probability ϵ', then with probability at least $\epsilon'/2$, $E[N, e, \mathsf{st}, \pi]_2$ occurs and $\mathsf{Alg2AM}$ either returns a factor of N or $S_{\psi+1}$ that is functionally equivalent to $R_{\psi+1}$.

Let p_1 denote the probability that $E[N, e, \mathsf{st}, \pi]_2$ occurs and $\mathsf{Alg2AM}$ returns a factor of N Let p_2 denote the probability that $E[N, e, \mathsf{st}, \pi]_2$ occurs and $S_{\psi+1}$ is functionally equivalent to $R_{\psi+1}$. Note that, by the above, $p_1 + p_2 \geq \epsilon/8$.

If $E[N, e, \mathsf{st}, \pi]_2$ occurs and $S_{\psi+1}$ is functionally equivalent to $R_{\psi+1}$, then it means that $\nu_N(S_{\psi+1}) \geq \delta$. Using Lemma 10 we have that $\mathsf{SLPFactoring}^{\pi}$

factors successfully on input $S_{\psi+1}$ with probability $p_3 \in \Omega(M' \cdot \nu_N(f)/T_{2,r}) = \Omega(M' \cdot \frac{\epsilon'(1-\epsilon'/2)}{\log N(T_{2,r})^2}) = \Omega(\epsilon'(1-\epsilon'/2))$.

Thus, in total, the probability of factoring successfully is at least $p_1 + p_2 \cdot p_3 \geq p_3(p_1+p_2) \geq p_3 \cdot \epsilon'/2 \in \Omega((\epsilon')^2(1-\epsilon'/2))$. This concludes the proof.

Corollary 2. *Let $A = (A_0^\pi, A_1^\pi)$ be an $(S_r, T_{2,r})-$ GP-RSA algorithm with advantage ϵ and let $S_r \cdot T_{2,r} < 2^\kappa/(\widetilde{16 \cdot \epsilon})$. Let $S_f = S_r$ and $T_f = O(\kappa^2 \frac{(\kappa T_{2,r})^{7/2}}{(1-\epsilon'/2)\epsilon')^{3/2}} + T_{2,r}^5 \kappa^3 + T_{1,r})$. Then $\mathsf{Factoring}^{\mathsf{H}}$ is an (S_f, T_f)-factoring algorithm and $\mathsf{Adv}^{\mathbf{fac}}_{\mathsf{RSAGen}}(\widetilde{\mathsf{Factoring}^{\mathsf{H}}}) \in \Omega(\epsilon^3)$ in the random invertible permutation model.*

Proof. By Lemma 7, with probability at least $\epsilon/2$ over the random coins of A_0, choice of π, and coins of RSAGen, A_0^π outputs st and RSAGen outputs (N, e, d) s.t.

$$\Pr_{\mathsf{ComGRA}, y \leftarrow \mathbb{Z}_N}(E[N, e, \mathsf{st}, \pi]_2) \geq 1 - \epsilon/4 \geq \epsilon/2 = \epsilon'$$

By Lemma 9, we see that running Algorithm $\mathsf{Factoring}^\pi$ on input N, e, st takes time $O((\kappa^2 \frac{\kappa T_{2,r}}{(1-\epsilon'/2)\epsilon'})^{3/2} + T_{2,r}^5 \kappa^3 + T_{1,r})$ and returns a non-trivial factor of N with probability at least $\widetilde{\Omega((\epsilon')^2(1-\epsilon'/2))}$ over its choice of random coins. Overall, this implies that $\mathsf{Factoring}^{\mathsf{H}}$ runs in online time $O((\kappa^2 \frac{\kappa T_{2,r}}{(1-\epsilon'/2)\epsilon'})^{3/2} + T_{2,r}^5 \kappa^3 + T_{1,r})$ and returns a factor of N with probability at least $\Omega(\epsilon \cdot (\epsilon')^2(1-\epsilon'/2)) \in \Omega(\epsilon^3)$.

Switching to the RO Model. We now show that the factoring algorithm from above, which was presented in the Random Injective Function model (with backwards and forwards access to the function), can be converted into a factoring algrotihm in the Random Oracle model.

In Proposition 1 we take A to be our factoring algorithm $\mathsf{Factoring}^\pi$ from Lemma 9 and $q = 2^\kappa$. Now set L such that

$$2^{2\kappa}/L \in O(N^2/L) \leq 1/(2N)\ .$$

As $\epsilon_f \in \Omega(1/N)$ where ϵ_f is the advantage $\mathsf{Factoring}^\pi$ relative to a random injection on $[L]$, we have

$$\epsilon'_f \geq \epsilon_f/2$$

where ϵ'_f is the advantage of the factoring algorithm in RO model that runs $\mathsf{Factoring}^\pi$, answering its queries via Luby-Rackoff.

4.4 Constructing a Factoring Algorithm in the RO model when $S_r \cdot T_{2,r} \geq \epsilon' 2^\kappa/4$

We begin by recapping Hellman's construction [17] for inverting a function $f : \{0,1\}^\kappa \to \{0,1\}^\kappa$ before presenting the main result of this section.

Hellman's Inversion Algorithm in the RO model. Hellman's algorithm is parameterized by (ℓ, m, t) and achieves space $S = \ell \cdot m$ and time $T = \ell \cdot t$.

Preprocessing Phase: The preprocessing phase outputs a table Table that consists of ℓ smaller tables $\mathsf{Table} = \mathsf{Table}_1, \ldots, \mathsf{Table}_\ell$. For $i \in [\ell]$, each Table_i consists of m entries entries $[(sp_i^j, ep_i^j)]_{j\in[m]}$, where sp_i^j is chosen at random from $\{0,1\}^\kappa$, $ep_i^j = g_i^t(sp_i^j)$, and $g_i = h_i \circ f$, where each h_i is an independent random oracle.

Online Phase: To invert an input $z \in \{0,1\}^\kappa$, for $i \in [\ell]$ do the following:

1. Set $u_i = h_i(z)$.
2. Repeat for $k \in [t]$: (a) Search for u_i in Table_i. If found (i.e. $u_i = ep_i^j$ for some j), compute $g_i^{t-k}(sp_i^j)$. If $f(g_i^{t-k}(sp_i^j)) = z$, then we have found a pre-image of z and we say that (i, j) is *useful* for z. Return $g_i^{t-k}(sp_i^j)$. (b) Set $u_i = g_i(u_i)$.

Proposition 2. *Let $A = (A_0, A_1)$ be a $(S_r, T_{2,r})$-RSA algorithm with advantage at least $\epsilon' = \epsilon/4$ and assume $S_r \cdot T_{2,r} \geq \epsilon' 2^\kappa/4$. Then there exists a RO-model (S_f, T_f)-factoring-with-preprocessing algorithm such that for $\kappa \in \mathbb{N}$, we have*

$$\mathsf{Adv}^{\mathbf{fac}}_{\mathsf{RSAGen}}(A) \in \Omega(\epsilon),$$

we have that $S_f = S_r$, and $T_f = \mathsf{poly}(\kappa) \cdot T_{2,r}^2$.

Proof. We first construct a factoring algorithm that succeeds with high probability and uses more space. We will then show how to reduce the space at the cost of reducing the success probability (but still achieving the required bounds).

We will invert the multiplication function $f := \mathsf{Mult}_\kappa(x, y) := \langle x \rangle \cdot \langle y \rangle$, where the brackets indicate encodings of x, y as $\kappa/2$-bit unsigned binary integers, for $x, y \in \{0,1\}^{\kappa/2}$. Our point generator for the function inversion problem with respect to $f = \mathsf{Mult}_\kappa(x, y)$ will be $G(1^\kappa)$ which outputs $N = pq$ where p, q are random $\kappa/2$-bit primes. Note that N outputted by RSAGen is identically distributed to N outputted by $G(1^\kappa)$. Further, for $N = pq$ where N has length κ, and p, q are $\kappa/2$-bit primes, $\mathsf{Mult}_\kappa^{-1}(N) = \{(p, q), (q, p)\}$; there are no other inverses. Therefore inverting $f = \mathsf{Mult}_\kappa$ reveals its correct factorization.

For $z \in \{0,1\}^\kappa$, recall that $I_f(z)$ denotes the number of preimages for z under $f = \mathsf{Mult}_\kappa$. Note that $I_f(z) \leq d(z)$, where d is the divisor function–i.e. the function that returns the number of divisors of an integer (including 1 and the number itself). We upperbound the collision probability $q(f)$ of f as follows:

$$q(f) := \frac{\sum_{z=0}^{2^\kappa - 1} I_f^2(z)}{2^{2\kappa}} \leq \frac{I_f^2(0)}{2^{2\kappa}} + \frac{\sum_{z=1}^{2^\kappa} d^2(z)}{2^{2\kappa}}.$$

An important line of work [23, 25, 31] proved that

$$\sum_{z=1}^{2^\kappa} d^2(z) = O(2^\kappa \cdot \kappa^3).$$

Combining the above two equations and using the fact that $I(0) \leq 2^{\kappa/2+1}$ yields

$$q(f) \leq \frac{4}{2^\kappa} + O\left(\frac{\kappa^3}{2^\kappa}\right) \in O\left(\frac{\kappa^3}{2^\kappa}\right). \tag{1}$$

Applying a theorem of Fiat and Naor (see Theorem 2), we have that for any choice of S'_f, T'_f such that $T'_f \cdot (S'_f)^2 \geq 2^{3\kappa} \cdot q(f)$, there exist settings of parameters (ℓ', m', t') such that Hellman's technique instantiated with these parameters yields an RO model algorithm that uses space $S'_f = \ell' \cdot m'$, time $T'_f = \ell' \cdot t'$, and achieves an inversion probability of $1 - 1/2^\kappa$. Recall that by assumption, $S_r \cdot T_{2,r} \geq \epsilon' 2^\kappa/4$, and that by (1) there is a constant c such that for sufficiently large κ, $q(f) \leq c \cdot \frac{\kappa^3}{2^\kappa}$. We set $S_{f'} := 1/\epsilon' \cdot S_r$ and and $T_{f'} := 16 \cdot c \cdot \kappa^3 \cdot T^2_{2,r}$. This setting satisfies the requirement $T'_f \cdot (S'_f)^2 \geq 2^{3\kappa} \cdot q(f)$ when $S_r \cdot T_{2,r} \geq \epsilon' 2^\kappa/4$, therefore yielding an inversion algorithm that succeeds with probability $1 - 1/2^\kappa$.

We now modify the output of the preprocessing stage to reduce the space requirements, at the cost of lowering the success probability. Let $\mathcal{S}$ be the subset of $\{0,1\}^\kappa$ which consists of strings N of the form $N = pq$, where p, q are primes of length $\kappa/2$. Note that the algorithm described above inverts with some constant probability, p, on the set $\mathcal{S}$ (actually it can be made to succeed with higher advantage, but this is sufficient for our purposes). For $N \in \mathcal{S}$, consider the set $\mathcal{U}_N$ of pairs (i, j) such that (i, j) is *useful* for N (see Step 2(a) of Hellman's algorithm at the beginning of the subsection for the definition of useful). Define $\mathsf{entry}(N) = (i, j)$, to be the lexicographically first pair in the set $\mathcal{U}_N$, if the set is non-empty, and define $\mathsf{entry}(N) = \perp$ otherwise. Let indicator variable $I_{\mathsf{entry}(N)=(i,j)}$ be equal to 1 if $\mathsf{entry}(N) = (i, j)$. Note that $\sum_{N\in\mathcal{S}} \sum_{(i,j)\in[\ell']\times[m']} I_{\mathsf{entry}(N)=(i,j)} = \sum_{(i,j)\in[\ell']\times[m']} \sum_{N\in\mathcal{S}} I_{\mathsf{entry}(N)=(i,j)} \geq p|\mathcal{S}|$. This implies that there must exist a set $\mathcal{R}$ of $\epsilon' \cdot \ell' \cdot m'$ number of entries (i, j) such that $\sum_{(i,j)\in\mathcal{R}} \sum_{N\in\mathcal{S}} I_{\mathsf{entry}(N)=(i,j)} \geq \epsilon' p|\mathcal{S}|$. Consider a modified preprocessing algorithm that generates the table as before, selects this set $\mathcal{R}$ of entries, and then outputs the table consisting only of entries (sp^j_i, ep^j_i) such that $(i, j) \in \mathcal{R}$ to the online stage. Now, the online stage of the new algorithm is guaranteed to succeed with probability $p\epsilon'$, where p is constant. Further the new running time T_f is equal to $T_{f'}$. However, $S_f = \epsilon' \cdot \ell' \cdot m' = \epsilon' \cdot S_{f'} = S_r$. Note that, as desired, $S_f = S_r$, $T_f \in \mathsf{poly}(\kappa) \cdot T^2_{2,r}$, and $\epsilon_f = p \cdot \epsilon' \in \Omega(\epsilon)$.

References

1. D. Aggarwal and U. Maurer. Breaking RSA generically is equivalent to factoring. In A. Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 36–53. Springer, Heidelberg, Apr. 2009.
2. M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In D. E. Denning, R. Pyle, R. Ganesan, R. S. Sandhu, and V. Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, Nov. 1993.
3. D. J. Bernstein and T. Lange. Non-uniform cracks in the concrete: The power of free precomputation. In K. Sako and P. Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 321–340. Springer, Heidelberg, Dec. 2013.

4. D. Boneh. Twenty years of attacks on the rsa cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2):203–213, 1999.
5. D. Boneh and R. Venkatesan. Breaking RSA may not be equivalent to factoring. In K. Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 59–71. Springer, Heidelberg, May / June 1998.
6. D. R. L. Brown. Breaking rsa may be as difficult as factoring. *Eprint Cryptology Archive*, 2006.
7. D. Coppersmith. Modifications to the number field sieve. *J. Cryptol.*, 6(3):169–180, 1993.
8. S. Coretti, Y. Dodis, and S. Guo. Non-uniform bounds in the random-permutation, ideal-cipher, and generic-group models. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 693–721. Springer, Heidelberg, Aug. 2018.
9. S. Coretti, Y. Dodis, S. Guo, and J. P. Steinberger. Random oracles and non-uniformity. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 227–258. Springer, Heidelberg, Apr. / May 2018.
10. H. Corrigan-Gibbs and D. Kogan. The discrete-logarithm problem with preprocessing. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 415–447. Springer, Heidelberg, Apr. / May 2018.
11. I. Damgård and M. Koprowski. Generic lower bounds for root extraction and signature schemes in general groups. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 256–271. Springer, Heidelberg, Apr. / May 2002.
12. A. De, L. Trevisan, and M. Tulsiani. Time space tradeoffs for attacks against one-way functions and prgs. In T. Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 649–665, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
13. Y. Dodis, S. Guo, and J. Katz. Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In J.-S. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 473–495. Springer, Heidelberg, Apr. / May 2017.
14. Y. Dodis, I. Haitner, and A. Tentes. On the instantiability of hash-and-sign RSA signatures. In R. Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 112–132. Springer, Heidelberg, Mar. 2012.
15. A. Fiat and M. Naor. Rigorous time/space tradeoffs for inverting functions. pages 534–541, 01 1991.
16. G. Fuchsbauer, E. Kiltz, and J. Loss. The algebraic group model and its applications. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, Aug. 2018.
17. M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theory*, 26(4):401–406, 1980.
18. A. Joux, D. Naccache, and E. Thomé. When e-th roots become easier than factoring. In K. Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 13–28. Springer, Heidelberg, Dec. 2007.
19. J. Katz, J. Loss, and J. Xu. On the security of time-lock puzzles and timed commitments. In R. Pass and K. Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 390–413. Springer, Heidelberg, Nov. 2020.
20. G. Leander and A. Rupp. On the equivalence of RSA and factoring regarding generic ring algorithms. In X. Lai and K. Chen, editors, *ASIACRYPT 2006*, volume 4284 of *LNCS*, pages 241–251. Springer, Heidelberg, Dec. 2006.
21. M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2), 1988.

22. U. M. Maurer. Abstract models of computation in cryptography (invited paper). In N. P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, Dec. 2005.
23. M. B. Nathanson. *Elementary methods in number theory*, volume 195. Springer Science & Business Media, 2008.
24. V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.
25. S. Ramanujan. Some formulae in the analytic theory of numbers. *Messenger of Math*, 45:81–84, 1916.
26. R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the Association for Computing Machinery*, 21(2):120–126, 1978.
27. R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, MIT, 1996.
28. L. Rotem and G. Segev. Generically speeding-up repeated squaring is equivalent to factoring: Sharp thresholds for all generic-ring delay functions. In D. Micciancio and T. Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 481–509. Springer, Heidelberg, Aug. 2020.
29. V. Shoup. Lower bounds for discrete logarithms and related problems. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.
30. A. van Baarsen and M. Stevens. On time-lock cryptographic assumptions in abelian hidden-order groups. In *ASIACRYPT*, pages 367–397, 2021.
31. B. Wilson. Proofs of some formulae enunciated by ramanujan. *Proceedings of the London Mathematical Society*, 2(1):235–255, 1923.

Appendix

A Proof of Lemma 6

Proof. Let $\mathsf{A} = (\mathsf{A}_0^\pi, \mathsf{A}_1)$ be as in the lemma statement. We set $r_1 := \lceil 2 \log N/\epsilon' \rceil$ and $r_2 := \lfloor \frac{\epsilon N}{2T_{2,r}} \rfloor$. We will now use A_1 to construct a compression algorithm Enc^π.

Analysis of Enc^π. The length of the encoding E can be calculated as follows. Enc^π initially stores st, N in E, which are of size at most S_r and $\log N$, respectively. It then stores an encoding of $\mathcal{I}$ of length $\log \binom{L}{N}$. In each of the $r_2 = \lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor$ runs of the outer loop, Enc^π stores a $\log(r_1)$ bit encoding of the index among the r_1 runs of the inner loop that sets the condition good to true. It also stores the index ℓ among the roots of the polynomial f s.t. $f(y) = 0$, where y is the value being encoded in that repetition of the outer loop. This takes another $\log(J)$ bits. It also stores the index $\zeta \in [\psi{+}1]$ of the polynomial with respect to which y is negatively oriented, and where $\psi{+}1 \leq T_{2,r}$. This takes another $\log(T_{2,r})$ bits. So, overall, the outer loop adds at most $r_2 \cdot (\log(r_1){+}\log(J){+}\log(T_{2,r})) = r_2 \log(r_1 \cdot J \cdot T_{2,r})$ bits to E in this manner. Note that we choose J in such a way to ensure that $r_1 \cdot J \cdot T_{2,r}$ is a power of 2.

Next, we consider the number of bits added to E via Table. Note that values are added to Table in the order of iterating through the loops and so we can encode them slightly more efficiently than using the trivial encoding by using values already stored in Table. More concretely, suppose that Table has size $|\mathsf{Table}|$ at the time when a new value is to be added to Table. Then we can store this value using only $\log(N - |\mathsf{Table}|)$ many (amortized) bits, (rather than N many) by excluding all of the values already in Table.

For $i \in [1, \ldots, r_2]$, let t_i be the number of table entries modified in the i-th run; define $t_0 := 0$. Note that $t_i \leq T_{2,r}$. During the ith run of the outer loop, Table will be of size $t_1 + \ldots + t_i + \ell - 1$ when adding the ℓth value of run i to Table. This means that we can encode this value using $\log(N - (\ell + t_1 + \ldots + t_i - 1))$ many bits. Overall, we get at most $r_2 + \sum_{m=2}^{r_2} \sum_{\ell=0}^{t_m - 1} \log(N - (\ell + t_1 + \cdots + t_{m-1}))$ for the size of Table, where the additive r_2 comes from packing the t_i entries into a single encoding with an integer number of bits.

Finally, we can add the remaining points of the mapping π to E using $\log(N - (t_1 + \cdots + t_{r_2}))$ number of bits to specify the mapping, giving a final term of

$\sum_{k=(t_1+\cdots+t_{r_2})}^{N-1} \log(N-k)$. Hence, we obtain overall:

$$
\begin{aligned}
&|\mathsf{E}| \leq \\
&S_r + \log N + \log \binom{L}{N} + r_2(\log(r_1 J T_{2,r}) + 1) + \sum_{k=t_1+\cdots+t_{r_2}}^{N-1} \log(N-k) \\
&+ \sum_{m=1}^{r_2} \sum_{\ell=0}^{t_m-1} \log(N-(\ell+t_1+\cdots+t_{m-1})) \\
&= S_r + \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{k=t_1+\cdots+t_{\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor}}^{N-1} \log(N-k). \\
&+ \sum_{m=1}^{\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor} \sum_{\ell=0}^{t_m-1} \log(N-(\ell+t_1+\cdots+t_{m-1}))
\end{aligned}
$$

To upper bound this encoding length in the worst case, we examine the size of an element added to E as a result of the ith run of the outer loop. We distinguish two types of elements. The first type are entries of Table; these are added after the outer loop has terminated and they are size at least $\log(N-(t_1+\cdots+t_{r_2}))$ (amortized). The second type of element that is added are the pointers k, ℓ, and ζ; they take up a combined space of at most $\log(r_1 J T_{2,r})$. Since elements of the second type are larger than the size of the first type, it is clear that we want to add as few of the latter type as possible, while as adding as many of the former type as possible in order to maximize the size of E. Hence, we want to maximize the number of added elements t_i in each of these repetitions. We therefore set for $i \in [d], t_i = T_{2,r}$.

Thus, we have:

$$
\begin{aligned}
&S_r + \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) \\
&+ \sum_{m=1}^{\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor} \sum_{\ell=1}^{t_m - 1} \log(N - (\ell - 1 + t_1 + \cdots + t_{m-1})) + \sum_{k=t_1+\cdots+t_{\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor}}^{N-1} \log(N-k) \\
&\leq S_r + \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \\
&\sum_{m=1}^{\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor} \sum_{\ell=1}^{T_{2,r} - 1} \log(N - (\ell - 1 + (m-1)T_{2,r})) + \sum_{k=\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}}^{N-1} \log(N-k) \\
&= S_r + \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) \\
&+ \sum_{m=1}^{\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor} \sum_{\ell=1}^{T_{2,r} - 1} \log(N - (\ell - 1 + (m-1)T_{2,r})) + \sum_{k=1}^{N - \lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}} \log(k).
\end{aligned}
$$

Note that if $m = 1$, then $\log(N-(\ell+(m-1)T_{2,r})) = \log(N-\ell)$ takes values from $\log(N-1), ..., \log(N-T_{2,r}+1)$, as ℓ varies. Similarly, if $m = 2$, then $\log(N-(\ell+T_{2,r})) = \log(N-\ell-T_{2,r})$ takes values $\log(N-1-T_{2,r}), \ldots, \log(N-2T_{2,r}+1)$, as ℓ varies. More generally, $\log(N-(\ell+(m-1)T_{2,r}))$ takes all values $\log(N-j)$

s.t. $j \in [\epsilon' N/2]$ and $T_{2,r} \nmid j$. Hence, we continue with

$$
\begin{aligned}
&S_r + \log N + \log \binom{L}{N} + \\
&\left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{\substack{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}] \\ T_{2,r} \nmid j}} \log(N - j + 1) + \sum_{k=1}^{N - \lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}} \log(k) \\
&= S_r + \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}]} \log(N - j + 1) \\
&- \sum_{\substack{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}] \\ T_{2,r} \mid j}} \log(N - j + 1) + \sum_{k=1}^{N - \lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}} \log(k) \\
&= S_r + \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) \\
&+ \sum_{j \in [N]} \log(N - j + 1) - \sum_{\substack{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}] \\ T_{2,r} \mid j}} \log(N - j + 1) \\
&\leq S_r + 2 \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) \\
&+ \sum_{j \in [N]} \log(N - j + 1) - \sum_{\substack{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}] \\ T_{2,r} \mid j}} \log(N - j) \\
&= S_r + 2 \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) \\
&+ \sum_{j \in [N]} \log(N - j + 1) - \log \left(\prod_{\substack{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}] \\ T_{2,r} \mid j}} (N - j) \right) \\
&= S_r + 2 \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{j \in [N]} \log(N - j + 1) \\
&- \log \left(\prod_{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor]} (N - j \cdot T_{2,r}) \right) \\
&\leq S_r + 2 \log N + \log \binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{j \in [N]} \log(N - j + 1) \\
&- \log \left(\prod_{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor]} \left(T_{2,r} \left(\left\lfloor \frac{N}{T_{2,r}} \right\rfloor - j \right) \right) \right)
\end{aligned}
$$

The second to last step in the above derivation follows from the fact that we can write any $m \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor \cdot T_{2,r}]$ s.t. $T_{2,r} \mid m$ as $m = j \cdot T_{2,r}$, where $j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor]$. We continue with

$$
\begin{aligned}
=& S_r + 2\log N + \log\binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{j \in [N]} \log(N - j + 1) \\
& - \log\left((T_{2,r})^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor} \prod_{j \in [\lfloor \frac{\epsilon' N}{2T_{2,r}} \rfloor]} \left(\left\lfloor \frac{N}{T_{2,r}} \right\rfloor - j \right) \right) \\
=& S_r + 2\log N + \log\binom{L}{N} + \left\lfloor \frac{\epsilon' N}{2T_{2,r}} \right\rfloor (\log(r_1 J T_{2,r}) + 1) + \sum_{j \in [N]} \log(N - j + 1) \\
& - \log\left(\frac{\lfloor N/T_{2,r} \rfloor!}{(\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}\rfloor))!} \cdot (T_{2,r})^{\lfloor \epsilon' N/(2T_{2,r}\rfloor)} \right) \\
=& S_r + 2\log N + \log\binom{L}{N} + (\log((r_1 J T_{2,r}) + 1)^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}) + \log(N!) \\
& - \log\left(\frac{\lfloor (N/T_{2,r}) \rfloor!}{(\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}) \rfloor)!} \cdot (T_{2,r})^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor} \right) \\
=& S_r + \log\binom{L}{N} + \log\left(\frac{N! N^2 (\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}) \rfloor)! (2r_1 J T_{2,r})^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}}{\lfloor (N/T_{2,r}) \rfloor! \cdot (T_{2,r})^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}} \right) \\
\leq& \log\left(\frac{2^{S_r} \cdot N! \cdot N^2 (2r_1 J)^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}}{(\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}) \rfloor)^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}} \right).
\end{aligned}
$$

Next, we analyze Enc^π's failure probability.

Claim. Failure Event 2.1 occurs with probability at most $1/2$ over the randomness of Enc^π.

Proof. Recall that we have set $r_1 = 2\log N/\epsilon'$. Moreover, we have assumed that $\Pr_{\mathsf{ComGRA}[\mathsf{A}_1], y \leftarrow \mathbb{Z}_N}[E[N, e, d, \mathsf{st}, \pi]_1] \geq \epsilon'$. To bound the probability of Event 2.1, we consider the binary matrix Q defined for fixed π, N, and the randomized algorithm $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ in the following manner:

- Row i of Q is labelled with $\pi(y)$, where $y \in \mathbb{Z}_N$ is such that $\pi(y)$ is the ith value in lexicographical order in the range of π.
- Column j of Q is labelled with the jth bitstring $\rho \in \{0,1\}^t$ in lexicographical order, where t denotes the number of random coins $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ takes in.
- Let $\pi(y)$ and ρ correspond to rows i and column j. $Q_{i,j} = 1$ if $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ on input $\pi(y)$ and random coins ρ outputs a set of SLPs over $\mathbb{Z}_N$ such that y is (δ, N)-negatively oriented with respect to at least one of them. $Q_{i,j} = 0$ otherwise.

If $\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ succeeds with probability ϵ' over random choice of $\pi(y), \rho$ then at least ϵ' fraction of Q's entries are 1. Furthermore, a random run of

$\mathsf{ComGRA}[\mathsf{A}_1]^\pi$ on input $\pi(y)$ for random $y \in \mathbb{Z}_N$ corresponds to choosing a point in Q uniformly at random.

Consider a randomized procedure that in each step adds an entry $(y, \pi(y))$ to Table, where Table is initialized as empty.

Now consider the submatrix Q' of Q with rows labeled by values in $\mathcal{Y}$ where $\pi(y) \in \mathcal{Y}$ if $y \in \mathbb{Z}_N$ and $(y, \pi(y)) \notin \mathsf{Table}$.

Then the submatrix Q' has at least $\epsilon' \cdot N \cdot 2^t - |\mathsf{Table}| \cdot 2^t = (\epsilon' \cdot N - |\mathsf{Table}|)2^t$ number of 1's. If at each step of the randomized procedure, the size of Table is upper bounded by $\epsilon' N/2$, then at each step, the submatrix Q' corresponding to $\mathcal{Y} \times \{0,1\}^t$ has at least $(\epsilon' N/2)2^t$ number of 1's. Further, the fraction of 1's in Q' is at least

$$\frac{(\epsilon' \cdot N - |\mathsf{Table}|)2^t}{(N - |\mathsf{Table}|)2^t} = \frac{\epsilon' \cdot N - |\mathsf{Table}|}{N - |\mathsf{Table}|} \geq \frac{\epsilon' \cdot N - \epsilon' N/2}{N} = \epsilon'/2.$$

Further, the fraction of 1's at each step is at least $\epsilon'/2$.

Thus, Failure Event 2.1 occurs with probability at most $(1-\epsilon'/2)^{r_1} \leq e^{(-\epsilon' r_1/2)} = e^{(-2\epsilon' \log(N)/(2\epsilon'))} \leq 1/N$ in any repetition of the inner loop. As $r_2 = \epsilon' N/(2T_{2,r})$, by a union bound, the probability that any of the r_2 repetitions of the outer loop produce Event 2.1, is at most $N\epsilon'/(2T_{2,r}) \cdot 1/N \leq 1/2$. □

Recall that J is the maximum integer that satisfies $J \leq \frac{\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}) \rfloor}{8r_1}$ and $r_1 J T_{2,r}$ is a power of two. Note that we can lower bound J by $J \geq \frac{(1-\epsilon'/2)N}{32 r_1 T_{2,r}}$. Plugging the value of J gives us that the length of the encoding is upperbounded as follows:

$$\begin{aligned}
&\log\left(\frac{2^{S_r} \cdot \binom{L}{N} \cdot N! \cdot N^2 (2r_1 J)^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}}{(\lfloor N/T_{2,r} \rfloor - \lfloor \epsilon' N/(2T_{2,r}) \rfloor)^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}}\right) \\
&\leq \log\left(2^{S_r} \cdot \binom{L}{N} \cdot N! \cdot N^2 (1/4)^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}\right) \\
&= S_r + \log\binom{L}{N} + \log(N!) + 2\log(N) - \log\left(4^{\lfloor \epsilon' N/(2T_{2,r}) \rfloor}\right) \\
&= S_r + \log\binom{L}{N} + \log(N!) + 2\log(N) - 2\lfloor \epsilon' N/(2T_{2,r}) \rfloor.
\end{aligned}$$

Thus, Enc^π does not fail with probability at least $1/2$ over its randomness and returns an encoding E of the appropriate size. Since by assumption, $S_r \cdot T_{2,r} \leq \epsilon' 2^\kappa/4$, and since $N \geq 2^\kappa/2$, we have $S_r \cdot T_{2,r} \leq \epsilon' N/2$. Substituting $S_r \leq \epsilon' N/(4T_{2,r})$ in the above expression yields, $|\mathsf{E}| < \log\binom{L}{N} + \log(N!) + 2\log(N) - \epsilon' N/(2T_{2,r})$.

B SLP Factoring Algorithm

In this section we present $\mathsf{SLPFactoring}^\pi$, which is a slightly modified version of Algorithm 1 due to Aggarwal and Maurer [1] and is used in one of the cases of

the proof of our main result. In this algorithm, we let $H(b(x), c(x))$ denote the non-trivial, non-invertible element output when Euclid's algorithm is executed on $\mathbb{Z}_N[x]$ with input $b(x)$ and $c(x)$. The only difference from Algorithm 1 of Aggarwal and Maurer [1] is that we repeat their entire algorithm M' times with independent random coins in order to improve the success probability.

Algorithm $\mathsf{SLPFactoring}^{\pi}$

Input: A T-step SLP S, a parameter M', and an integer N.
Output: A factor of N.

1. Repeat M' times, each with a freshly chosen random tape:
   (a) Choose a monic polynomial $h(x)$ uniformly at random from all monic polynomials of degree T in $\mathbb{Z}_N[x]$.
   (b) Compute $h'(x)$, the derivative of $h(x)$ in $\mathbb{Z}_N[x]$.
   (c) Choose a random element $r(x) \in \mathbb{Z}_N[x]/(h(x))$.
   (d) Compute $z(x) = f(r(x))$ in $\mathbb{Z}_N[x]/(h(x))$ using the instructions of SLP S (where f is the polynomial function computed by SLP S).
   (e) Run Euclid's algorithm in $\mathbb{Z}_N[x]$ on $h(x)$ and $z(x)$. If this fails, return $\gcd(N, H(h(x), z(x)))$.
   (f) Run Euclid's algorithm in $\mathbb{Z}_N[x]$ on $h(x)$ and $h'(x)$. If this fails, return $\gcd(N, H(h(x), h'(x)))$.

Fig. 10: Modified version of Algorithm 1 from Aggarwal and Maurer

Lemma 10. *Algorithm* $\mathsf{SLPFactoring}^{\pi}$ *takes as input* $N = pq$ *where* $p, q > 3$ *are primes, as well as* $T \in \mathbb{N}$*, a* T*-step SLP* S *and, for any* $M \in \mathbb{N}$ *such that* $M \cdot \nu_N(f)/(8T) \leq 1/2$*, runs in time* $O(M \cdot T^3\kappa^2)$*, and does the following: if* $(P^S(x), Q^S(x)) = (f_S(x), 1)$*,* $f_S(x) \not\equiv 0 \mod N$*, then* $\mathsf{SLPFactoring}^{\pi}$ *outputs a non-trivial factor of* N *with probability at least* $\Omega\left(M \cdot \nu_N(f_S)/(16T)\right)$*.*

Proof. Algorithm 1 from [1] is identical to lines all lines of $\mathsf{SLPFactoring}^{\pi}$ inside the loop (namely, lines 1.(a) through 1.(f)). The analysis of Algorithm 1 [1] gives us that each run of this loop takes $O(T^3 \log^2 N)$ time, and if the given SLP S is such that $(P^S(x), Q^S(x)) = (f_S(x), 1)$ and $f(x) \not\equiv 0 \mod N$, then the algorithm returns a factor of N with probability at least $\dfrac{\nu_N(f_S)}{8T}$. Then the runtime of $\mathsf{SLPFactoring}^{\pi}$ is $O(M \cdot T^3 \log^2 N)$ and its failure probability is at most

$$\begin{aligned}\left(1 - \frac{\nu_N(f_S)}{8T}\right)^M &= \left(1 - \frac{\nu_N(f_S)}{8T}\right)^{8T/\nu_N(f_S)\cdot M\cdot\nu_N(f_S)/(8T)} \\ &= e^{-M\cdot\nu_N(f_S)/(8T)} \\ &\leq 1 - M \cdot \nu_N(f_S)/(16T),\end{aligned}$$

where the inequality follows due to the fact that $e^{-2x} \leq 1 - x$ for $x \leq 1/2$. Thus the success probability is at least $M \cdot \nu_N(f_S)/(16T)$.