


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

Dana Dachman-Soled 

University of Maryland, College Park, MD, USA

Julian Loss  

CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

Adam O’Neill  

Manning College of Information & Computer Sciences, UMass Amherst, MA, USA

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 that perform *generic* computation on the RSA instance $x^e \bmod N$ yet arbitrary computation on the modulus N , namely a careful adaptation of the well-known generic ring model of Aggarwal and Maurer (Eurocrypt 2009) to the preprocessing setting. In particular, in this setting we show the existence of a factoring algorithm with polynomially related parameters, for any setting of RSA parameters.

Our main technical contribution is a set of new information-theoretic techniques that allow us to handle or eliminate cases in which the Aggarwal and Maurer result does not yield a factoring algorithm in the standard model with parameters that are polynomially related to those of the RSA algorithm. These techniques include two novel compression arguments, and a variant of the Fiat-Naor/Hellman tables construction that is tailored to the factoring setting.

2012 ACM Subject Classification Security and privacy → Public key (asymmetric) techniques; Security and privacy → Information-theoretic techniques

Keywords and phrases RSA, factoring, generic ring model, preprocessing

Digital Object Identifier 10.4230/LIPIcs.ITC.2024.8

Related Version *Full Version:* <https://eprint.iacr.org/2022/1261>

Funding *Dana Dachman-Soled:* 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.

Julian Loss: Supported by the European Union, ERC-2023-STG, Project ID: 101116713. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements We thank Nikki Sigurdson for collaboration in the early stages of this work.



© Dana Dachman-Soled, Julian Loss, and Adam O’Neill;
licensed under Creative Commons License CC-BY 4.0

5th Conference on Information-Theoretic Cryptography (ITC 2024).

Editor: Divesh Aggarwal; Article No. 8; pp. 8:1–8:24



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 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. The generic ring model (GRM) of AM considers an algorithm (called a generic ring algorithm or GRA) models generic computation on \mathbb{Z}_N via 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. Now, it is instructive to see why this model, extended to the preprocessing setting in the obvious way, is not suitable. In such an extension, at the end of the preprocessing stage, the adversary outputs a GRA to run in the online stage. But then observe that the best the adversary against RSA with preprocessing is the one that simply outputs, in the preprocessing stage, a 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 model, however, does not capture our intuition for the best-possible preprocessing attack against RSA in the GRM. For example, the following simple adversary is better but not captured: During preprocessing,

it picks many instance-solution pairs $((N_1, e_1, x_1^{e_1} \bmod N_1), x_1), \dots, ((N_i, e_i, x_i^{e_i} \bmod N_i), x_i)$ and inserts them into a hash table, then outputs this hash table as well as the aforementioned optimal GRA. Then, in the online stage, it performs a hash table lookup on the input RSA instance (N^*, e^*, y^*) to obtain a list of possible preimages $x_1^* \dots, x_j^*$ (these all being stored in the same location in the hash table). It returns x_k^* such that $(x_k^*)^{e^*} \bmod N^* = y^*$ if it exists, and runs the optimal GRA otherwise. This attack is not captured because the hash table lookup uses the *bit-representation of the RSA instance*, while a GRA is agnostic of the particular representation of the ring. However, the attack is still “generic” because it works for *any* bit-representation of the ring (not just the canonical encoding of \mathbb{Z}_N). While this is a simple example, it captures techniques originating from Hellman tables [19], a common strategy for preprocessing algorithms in practice. In other words, such strategies crucially use the bit-representation of the problem instance.

Our New “GRM-with-Preprocessing” Model. To capture these types of representation-specific strategies, we will associate integers y of bitlength at most κ with *random labels*. We note that Damgard and Koprowski [12] and later Dodis et al. [15] previously considered random labels to model the *multiplicative group* \mathbb{Z}_N^* (with N known to the adversary); it has not been done for the full ring \mathbb{Z}_N . (See below for a detailed comparison with these works.) That is, in our model, we consider an injective mapping π that encodes every element in $\{0, 1\}^\kappa$ as 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 *state* 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 *non-generic* and gets the RSA instance $(N^*, e^*, \pi((x^*)^{e^*} \bmod N^*))$, where $N^* = pq$ has bit-length κ , but does *not* get access to π . This intermediate algorithm is what allows computation that *depends on the bit-representation of the RSA instance*, so that the next online stage can leverage the result in addition to the advice from the preprocessing stage. The intermediate algorithm outputs an *oracle-aided* GRA that computes relative to π , which we then run on the RSA instance. By “relative to π ,” we mean 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 $S = |\text{state}|$ 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. It is instructive to see that the simple adversary related to Hellman tables above is captured by it. This is because the intermediate algorithm can perform a hash table lookup on the input RSA instance $(N^*, e^*, y^* = \pi((x^*)^{e^*} \bmod N^*))$ to get a list $\pi(x_1^*), \dots, \pi(x_j^*)$ of possible preimages, returning the GRA that on input y^* returns $\pi(x_k^*)$ such that $y^* = \pi(x_k^*)^{e^*} \bmod N^*$ if it exists (note finding this value uses access to π , *i.e.*, the ring operations) and runs the aforementioned optimal GRA otherwise. We view the GRM-with-preprocessing model as a main conceptual contribution of our work and contend that it faithfully captures our intuition for preprocessing attacks in the GRM.

We note that the study of upper and lower bounds on preprocessing in idealized models was pioneered by Corrigan-Gibbs and Kogan (CK) [10], who treated discrete-log-related problems in the case of groups. (See a more detailed comparison below.) Our modeling above follows their approach, but 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, while the proofs of

8:4 Breaking RSA Generically Is Equivalent to Factoring, *with Preprocessing*

our results (described in Section 2) like those of CK use compression, the details differ substantially. For example, we do not use the random self-reducibility of RSA, whereas CK rely on self-reducibility of discrete log. This makes our results potentially applicable to broader settings.

A Result in the Random Injective Function Model. We present two main results below, which both emanate from a more basic result in *random injective function model* (RIM). In the RIM, the adversary has access to a random injective function with suitable parameters. 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 RIM, with polynomially related parameters. This gets us a long way in answering 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, as long as we permit the factoring algorithm to operate in the RIM.

► **Theorem 1** (Informal). *Suppose there is an RSA adversary 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 injective function model (RIM) with space $S_f = S_r + O(1)$ and running-time $T_f = \text{poly}(\kappa, T_{1,r}, T_{2,r}, 1/\epsilon_r)$ that succeeds with probability $\epsilon_f = \text{poly}(\epsilon_r)$.*

See Theorem 7 for the formal statement. We will explain the bounds (which are identical) in the context of our random oracle model result below. Since our model allows an inefficient preprocessing phase, the RI function 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 RI function 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 RI “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 to removing the RI would be to show that the global information about the random injective function (which has length S_f) can be simulated by fixing the input/output of some set of some q queries to the random injective function, 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, *e.g.* [9, 14]. However, this line of work proved a lower bound that q must be larger than $S_f T_f / (\epsilon_f)^2$ for simulation by the plain-model adversary to be ϵ_f -indistinguishable to an RIM adversary making T_f queries (note that we require $\approx \epsilon_f$ -indistinguishability to guarantee that the factoring algorithm in the plain model still succeeds with probability $\text{poly}(\epsilon_f) = \text{poly}(\epsilon_r)$). For us, this would lead to trivial parameter settings. We show how to remove the RI function using an alternative argument below.

In particular, we extend the RIM result in two ways.

A result in the random oracle model. We first note that the RIM is much less natural to study factoring-with-preprocessing in than its counterpart the *random oracle model* (ROM) [2], hence we would like a result in the latter. The classical result of Luby and Rackoff shows that a 4-round Feistel network with random oracles in place of round functions is indistinguishable from a random permutation with forwards and backwards access. However, the distinguishing probability of an (unbounded) adversary is $\Omega(\frac{q^2}{2^{\kappa/2}})$, where $\kappa/2$ is the input/output length of the random oracle, and q is the number of queries made by the adversary, and this bound is known to be tight. In the preprocessing setting, the adversary

can query the entire random oracle $q = 2^{\kappa/2}$, and so the distinguishing probability becomes vacuous. We present a technique to lift the Luby-Rackoff result to the case of unbounded preprocessing by using a slight modification of a 4-round Feistel network to implement a random injective function, instead of a random permutation. This 4-round Feistel will use round functions with input/output length $m/2$ to implement an injective function with domain size of $2^\kappa \ll 2^{m/2}$ and will thus circumvent the issue discussed above. We thus obtain the following result (see Theorem 11 for the formal statement), with the same concrete bounds as the RIM result.

► **Theorem 2** (Informal). *Suppose there is an RSA adversary 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 $T_f = \text{poly}(\kappa, T_{1,r}, T_{2,r}, 1/\epsilon_r)$ that succeeds with probability $\epsilon_f = \text{poly}(\epsilon_r)$.*

Note that the space complexity of our factoring algorithm is essentially the same as that of the RSA algorithm, namely $S + O(1)$. 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 ROM factoring algorithm and that of AM in Section 4. We believe using the ROM for the above result is reasonable since prior work on space/time tradeoffs (such as the seminal results of Hellman [19] and Fiat-Naor [17]) either required a random oracle or achieved simplified algorithms/improved parameters in the random oracle model. Nevertheless, it begs the question of whether the situation could change in the plain model.

A result in the plain model. Above we explained why it is difficult to remove the RI function while maintaining the desired parameters. Nevertheless, by developing new techniques for our setting we are finally able to show the following theorem statement, which is in the *plain model*.

► **Theorem 3** (Informal). *Suppose there is an RSA adversary 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 plain model with space $S_f = O(S_r)$ and running-time $T_f = \text{poly}(\kappa, T_{1,r}, T_{2,r}, 1/\epsilon_r)$ that succeeds with probability $\epsilon_f = \text{poly}(\epsilon_r)$.*

The parameters are thus worse than what we obtain in the ROM, which is why we include the ROM result above. The main insight used for the plain-model result is to note that the online stage of the RI factoring algorithm we obtain has a particular form in which only a single uniformly random query is made to the forward direction of π , and a set of non-adaptive queries is made to the backward direction of π^{-1} . Combining a compression argument with a new argument based on a lemma of Drucker [16] (which to the best of our knowledge has not been previously used in a generic model setting) we are able to show that the online portion of the RIM factoring algorithm can be efficiently simulated in the plain model. Note, however, that we still include the ROM result above as it is obtained en route to our plain model result, illustrating many of our main techniques, and it enjoys a tighter reduction.

On Interpretation of our Results. To understand the significance of our results, it is helpful to recall the motivation for the GRM in the first place. AM explain that the GRM is an important computational model to consider for problems like RSA, where the adversary must output a function of its input that results in a ring element. As also mentioned below,

there exist decisional problems (*e.g.*, Jacobi symbol computation) that are hard in the GRM but easy in \mathbb{Z}_N . Nevertheless, this does not seem to affect the question of the hardness of RSA. Indeed, an interesting question for future work would be to extend our results to hold relative to a Jacobi symbol oracle. Now, moving onto our GRM-with-preprocessing model, we stress that the intermediate stage, being non-generic, can run standard factoring algorithms on N such as NFS. Of course, if it simply factors N to break RSA, there is nothing to show. However, our results demonstrate that such non-generic computation cannot be fruitfully combined with *generic* computation on the RSA instance to obtain a significantly faster-than-factoring attack on RSA. In other words, essentially the entire computation by an RSA adversary needs to be non-generic for such a speed-up to be possible.

Finally, 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. That is, 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.

On Using Bit-Fixing Instead of Compression. Another question is whether it is possible to rely on bit-fixing as alternative to our use of the compression technique (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 labelling 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.

On our use of information-theoretic techniques. We leverage information-theoretic techniques in three points in our proofs. First, in Section 4, we use a compression argument along with a theorem of De, Trevisan, Tulsiani [13] to show that the output of a successful RSA algorithm in the GRM with preprocessing model will satisfy a certain condition with high probability (see technical overview for more details). The condition being satisfied with high probability will then imply that the Aggarwal-Maurer factoring algorithm can be efficiently instantiated *in the ideal cipher or random oracle model*.

In Section 4.2, we use a different compression argument along with a theorem of Drucker [16] to show that our factoring algorithm (which invokes the Aggarwal-Maurer algorithm) in the ideal-cipher model can be efficiently simulated in the standard model by augmenting the preprocessing advice with a small table consisting of oracle query/response

pairs. Note that this argument is not straightforward, since the online part of the factoring algorithm must respond to queries consistently with the pre-processing information, and it is not clear that this can be done efficiently (e.g. sampling the responses of the ideal cipher “on-the-fly” can lead to inconsistencies with the pre-processing information which may depend on global properties of the oracle).

Finally, in the full version [11], we use a variant of the Fiat-Naor/Hellman tables [19, 17] to obtain a factoring algorithm with the required space, time complexity for a range of parameters that is not covered by our factoring algorithm (which invokes the Aggarwal-Maurer algorithm) from above.

1.2 Related Work

There is an extensive body of literature on the hardness of the RSA problem and its 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.* [20] shows that when given access to an oracle computing e th roots modulo N of integers $x + c$ (where c is fixed and x varies), computing e th 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 [22] 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 be used to analyze the security of verifiable delay functions [29].

The Generic Group Model (with Preprocessing). Starting with Nechaev [25], a long line of work has studied the complexity of group algorithm in the generic group model (GGM) [30, 24]. Algorithms in this model of Maurer [24] 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 [12] who ported Shoup’s generic group model [30] to the setting of groups with unknown order and showed the generic hardness of computing e th roots in this model. The second work is that of Dodis *et al.* [15] who considered the instantiability of the hash function in FDH-RSA. On the one hand, unlike the GRA model that we use for online adversary, they only model the *multiplicative group* \mathbb{Z}_N^* as generic. In other words, they do not allow the adversary to take advantage of the full ring structure of \mathbb{Z}_N . On the other, their model allows the online adversary to perform arbitrary side computations. Recall that we do not allow such computations in our model, as the online adversary is a GRA. We face many additional technical issues due to this point as well as preprocessing. 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. In addition to modeling differences mentioned above, they also consider adversaries who, in the online phase, may perform arbitrary side computations. Finally, the distinction between Maurer’s and Shoup’s GGM (*i.e.*, whether or not a labeling function is used) was studied in detail by Zhandry [32]; to our knowledge, the analogous issue has not been studied for GRAs.

The Algebraic Group Model. More recently, a series of works has explored the algebraic group model [18] as a means to abstract the properties of the groups \mathbb{QR}_N and the multiplicative group \mathbb{Z}_N^* more faithfully. The work of Katz et al. [21] 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 [31] who gave a general framework for computational reductions in the (strong) algebraic group model over \mathbb{Z}_N^* . Additionally, Rotem [28] reduces RSA to factoring in the algebraic group model over \mathbb{Z}_N^* .

2 Technical Overview

Our main result shows that any generic attack on RSA with preprocessing gives rise to a factoring algorithms with preprocessing in the random oracle model and plain models 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 $\pi : \{0, 1\}^\kappa \rightarrow \{0, 1\}^m$ and is completely unbounded *both* in terms of computation and number of queries to π . A_0^π finally outputs a state *state* of size S_r (called A ’s *space*). A_1 takes as input *state* 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 *state* 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 $\pi, \textit{state}, N, e$ 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 *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 $S_f = S_r + O(1)$, $T_f = \text{poly}(\kappa, T_{1,r}, T_{2,r}, 1/\epsilon_r)$ and $\epsilon_f = \text{poly}(\epsilon_r)$, where $\kappa = \log(N)$ is security parameter. 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 $T_r = T_{r,1} + T_{r,2}$.

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 briefly explain how to extend our result to the case where A_1 may output a generic ring algorithm (GRA).

First case analysis: Fiat-Naor argument. In the case that $T_r \cdot S_r \geq \epsilon_r \cdot 2^\kappa/4$, 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 [17], which extends Hellman’s seminal result on space/time tradeoffs for inversion of a *random* function [19], to obtain space/time tradeoffs for inversion of *any* function f . Specifically, Fiat and Naor consider an arbitrary function $f : D \rightarrow D$ 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 .¹ 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 $\kappa = \log(N)$. By carefully setting parameters and using properties of the second moment of the divisor function, to bound $q(f)$ as $q(f) \in O(\frac{\kappa^3}{2^\kappa})$, we obtain a factoring algorithm $S_f = S_r$, $T_f = \text{poly}(\kappa) \cdot T_{2,r}^2$ and inversion probability $O(\epsilon_r)$. Note that all parameters are polynomial in the parameters of the RSA algorithm. See [11] for more details.

Factoring from RSA. We now consider the main parameter regime of interest, where $T_r \cdot S_r < \epsilon_r \cdot 2^\kappa/4$. 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 begin by defining an “inversion procedure” on SLP’s that, given S^π *with oracle access* to π and such that $S^\pi(\pi(y)) = \pi(x)$, outputs an SLP \tilde{S} *with no oracle access* such that $\tilde{S}(y) = x$. (Crucially, the inversion procedure itself requires oracle access to π .) 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 $S_{\pi(y)}^\pi$ *after seeing* input $\pi(y)$ (we use the notation $S_{\pi(y)}^\pi$ to emphasize that the chosen SLP may depend on $\pi(y)$). This means that the SLP $S_{\pi(y)}^\pi$ output by A_1 can be tailored to succeed on $\pi(y)$ and on only *few* other inputs. Note that it is possible for A_1 to maintain an overall high success probability with this strategy. 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.

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

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 a (comparatively simple) third case which will imply existence of a factoring algorithm using the arguments of AM.

Second case analysis: Compression. For this case, we show how to construct an encoding routine that compresses the function table of a random injection π . Our main leverage to achieve this is the following idea. Suppose that 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 a space-efficient way for an encoding routine E^π (with oracle access to π) to transmit y to a decoder D (without oracle access to π) who knows only $\tilde{S}_{\pi(y)}$: Simply output the index of y among the J roots of $\tilde{S}_{\pi(y)}(Y)^e - Y$. (This takes $\log(J)$ bits.) 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 in the encoding which is being constructed by E^π . Making this intuition rigorous, however, is quite challenging.

First, we must show how the encoder can efficiently transmit the description of $\tilde{S}_{\pi(y)}$ to the decoder. We may assume that A_1 and *state* will be known to the decoder (we can include *state* in the encoding). However, to obtain the correct SLP $\tilde{S}_{\pi(y)}$, 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 $S_{\pi(y)}^\pi$ 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 [13]. 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, our encoder E^π includes 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 the full version [11] for more details.)

Factoring and Extending to the GRA case. Once we have ensured that the the SLP $S_{\pi(y)}(Y)^e - Y$ has *at least* J roots w.h.p., we can directly apply a theorem of AM to obtain a factoring algorithm. Our final step will then be to extend the above discussion to a slightly broader setting in which A_1 outputs a GRA G^π rather than an SLP S^π . Here, we once again build on arguments of AM, although we need to put in some additional effort to make them work in our setting with preprocessing. In particular, 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} . This is because our factoring algorithm requires access to such a random injection in order to consistently simulate the oracle π to the RSA adversary over the preprocessing phase and the online phase in a space efficient manner. Thus, it remains to show how this oracle can be simulated in order to obtain a factoring algorithm in the plain model. For simplicity, we omit our intermediate result in the Random Oracle Model from this technical overview.

Obtaining our plain model result. In the following, we denote the random injective function by H and we denote by π the GRM oracle interface expected by the RSA adversary. We note that using backwards and forwards access to H , one can easily simulate queries made to π . We show that with some additional work one can dispense with the RI in our result and obtain a result in the plain model. To do so, we first observe that the online portion of our factoring algorithm in the RI model makes only a *single* query to H in the forward direction (on a *uniformly random* input modulo N), and makes a series of non-adaptive queries to H^{-1} . We will first show that we can simulate all the responses to the queries to H^{-1} while adding only a small overhead to the non-uniform advice. We will then show that the single query to the forward direction of H can be simulated as well.

Simulating queries to H^{-1} . Recall that A_1 receives the non-uniform advice *state* and the input $(N, e, \pi(x^e \bmod N))$ and outputs a GRA. The factoring algorithm will run the GRA inversion algorithm by evaluating π^{-1} on hardcoded labels in the GRA that are *not equal* to the input label $\pi(x^e \bmod N)$. Intuitively, since π is expanding, and since A_1 may not query the oracle, the only way A_1 can hardcode a valid label into the GRA is if this label is somehow stored in *state*. To formalize this intuition, for a fixed π , we consider the set S_π of valid images of π that are hardcoded into a GRA outputted by A_1 with sufficiently high probability over choice of input $(N, e, \pi(x^e \bmod N))$ and the random coins of A_1 .

We use a compression argument to show that for most choices of π , the set S_π is sufficiently small such that it can be added to A_1 's advice *state*. By definition, for a fixed π , it is unlikely for A_1 to hardcode images of π into its outputted GRA if these images are not part of S_π . Thus, queries to π^{-1} can be simulated *without making a corresponding query to H* by using *state* as a lookup table.

Simulating the query to H . There is still a single query to the forward direction of H that must be taken care of. This is the query made by the factoring algorithm when generating the input to A_1 . Specifically, it is a query with input $y = x^e \bmod N$ and output $\pi(y) = \tilde{y}$. To simulate this query without accessing H , we construct a simulated plain model factoring algorithm as follows: In the preprocessing phase, the plain model algorithm internally samples a random injective function H , and the output of A_0 in the preprocessing stage is computed relative to this chosen H . Note that we can view A_0 's input in the preprocessing stage as the entire oracle, and in particular, this will include the input/output pair (y, \tilde{y}') , where $y = x^e \bmod N$ corresponds to the input value that will be given to A_1 in the online phase. In the online phase, our plain model factoring algorithm will actually resample the output value of H on input y and replace it with a uniform random string \tilde{y} . This resampled value

\tilde{y} will then be given to A_1 in the online phase as the supposed value of $\pi(y)$. A lemma of Drucker [16] implies that (on average) the output distribution of a compressing algorithm A_0 , which outputs *state*, does not change much when a single input in a randomly chosen location (location y) is switched from a fixed value to a randomly resampled value. This implies that the RI factoring algorithm will behave roughly the same when π is simulated in this manner. See the full version [11] for further details.

3 Preliminaries

3.1 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 $y \leftarrow A(x)$. We sometimes also write $y := A(x; \omega)$ 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 $\{1, \dots, N\}$. 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 $\text{Func}[k, m]$ the set of *functions* $F: \{0, 1\}^k \rightarrow \{0, 1\}^m$. Denote by $\text{Perm}[m]$ the set of *permutations* on $\{0, 1\}^m$. We denote by $\text{FuncInj}[k, m]$ the set of *injective functions* $I: \{0, 1\}^k \rightarrow \{0, 1\}^m$.

3.2 Incompressibility Lemmas

We use the following lemma by De et al. [13].

► **Lemma 4** (De, Trevisan, Tulsiani [13]). *Let $E: \mathcal{X} \times \{0, 1\}^\rho \rightarrow \{0, 1\}^m$ and $D: \{0, 1\}^m \rightarrow \mathcal{X} \times \{0, 1\}^\rho$ be randomized encoding and decoding procedures such that, for every $x \in \mathcal{X}$, $\Pr_{r \leftarrow \{0, 1\}^\rho} [D(E(x, r), r) = x] \geq \gamma$. Then, $m \geq \log |\mathcal{X}| - \log 1/\gamma$.*

► **Remark 5.** As noted by [10], this lemma also holds when the encoding and decoding algorithms have access to a common random oracle.

The following lemma is from Drucker [16].

► **Lemma 6** (Drucker [16]). *Let $N, S, m \geq 1$ be integers. Given a possibly-randomized mapping $A_0(\tilde{y}_0, \dots, \tilde{y}_{N-1}): \{0, 1\}^{N \times m} \rightarrow \{0, 1\}^S$, and a collection $\mathcal{D}_0, \dots, \mathcal{D}_{N-1}$ of mutually independent distributions over $\{0, 1\}^m$, for $y \in \mathbb{Z}_N$, let*

$$\gamma_y := \mathbb{E}_{\tilde{y} \sim \mathcal{D}_y} [\|A_0(\mathcal{D}_0, \dots, \mathcal{D}_{y-1}, \tilde{y}, \mathcal{D}_{y+1}, \dots, \mathcal{D}_{N-1}) - A_0(\mathcal{D}_0, \dots, \mathcal{D}_{N-1})\|_{\text{stat}}],$$

where the notation $\|\cdot - \cdot\|_{\text{stat}}$ denotes the statistical distance between two distributions.

We have that

$$\frac{1}{N} \sum_{y \in \mathbb{Z}_N} \gamma_y \leq \sqrt{\frac{\ln 2}{2} \cdot \frac{S+1}{N}}.$$

4 Main Results

We begin by stating two theorems that will be used to obtain both our plain model and RO model results.

► **Theorem 7.** *Let $S_r := S_r(\kappa)$, $T_{1,r} := T_{1,r}(\kappa)$, $T_{2,r} := T_{2,r}(\kappa)$, $\epsilon = \epsilon(\kappa)$ such that for sufficiently large κ , $S_r \cdot T_{2,r} \leq \epsilon/162^\kappa$. Let $A = (A_0^\pi, A_1)$ be an $(S_r, T_{1,r}, T_{2,r})$ -GP-RSA algorithm relative to RSAGen , and let $\text{Adv}_{\text{RSAGen}}^{\text{ag-rsa}}(A) = \epsilon$.*

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

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

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

This theorem is proved in the full version [11].

► **Remark 8.** We give a comparison here of the bounds we achieve in Theorem 7 versus those achieved by AM's factoring algorithm. First, we consider our runtime of $T_f := \text{poly}(\kappa) \cdot (T_{1,r} + T_{2,r}^5 + \frac{T_{2,r}^{7/2}}{\epsilon^{3/2}})$, 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 SLPFAC^π algorithm with $M' := \text{poly}(\kappa) \cdot (T_{2,r})^2$. 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 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 $\text{polylog}(N) = \text{poly}(\kappa)$ 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.

► **Remark 9.** 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 $T_f = O((2^{\kappa/10})^5) = O(2^{\kappa/2})$, with zero pre-processing and success probability 1, as well as a trivial factoring algorithm that achieves success probability $\Omega((2^{-\kappa/6})^3) = \Omega(2^{-\kappa/2})$ with zero pre-processing and $\text{poly}(\kappa)$ 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 $\epsilon' > 2^{-\kappa/6}$.

The following theorem instantiates the algorithm of Fiat-Naor in the setting of factoring-with-preprocessing.

► **Theorem 10.** *Let $\tilde{S} = \tilde{S}(\kappa)$, $\tilde{T} = \tilde{T}(\kappa)$, $\tilde{\epsilon} = \tilde{\epsilon}(\kappa)$ such that for sufficiently large κ , $\tilde{S} \cdot \tilde{T} \geq \tilde{\epsilon}2^\kappa$. Then there exists a plain-model (S_f, T_f) -factoring-with-preprocessing algorithm A such that for $\kappa \in \mathbb{N}$, we have*

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

we further have that $S_f = \tilde{S}$, and $T_f = \text{poly}(\kappa) \cdot \tilde{T}^2$.

This theorem is proved in the full version [11].

In Section 4.1 and 4.2 we formally state our results for the RO and plain model and explain how Theorems 7 and 10 are used to obtain those results.

4.1 The RO model result

► **Theorem 11.** *Let $A = (A_0^\pi, A_1)$ be an $(S_r, T_{1,r}, T_{2,r})$ -GP-RSA algorithm relative to RSAGen , and let $\epsilon := \text{Adv}_{\text{RSAGen}}^{\text{ag-rsa}}(A)$.*

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

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

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

To prove Theorem 11 we first show that the RI-model factoring algorithm from Theorem 7 (which gets backwards and forwards access to the random injective function), can be converted into a factoring algorithm in the Random Oracle model with the same parameters.

Specifically, in Proposition 21 we take A to be our final factoring algorithm FAC^π (see [11]) 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 FAC^π 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 FAC^π , answering its queries via Luby-Rackoff. This RO-model factoring algorithm is for the case that for sufficiently large κ , $S_r \cdot T_{2,r} \leq \epsilon/162^\kappa$.

Setting $\tilde{S} = S_r$, $\tilde{T} = T_{2,r}$, $\tilde{\epsilon} = \epsilon/16$ and applying Theorem 10, we obtain a plain model factoring algorithm with parameters $S_f = S_r$, $T_f = \text{poly}(\kappa) \cdot T_{2,r}^2$, and advantage $\epsilon_f \in \Omega(\epsilon)$. This plain-model factoring algorithm is for the case that for sufficiently large κ , $S_r \cdot T_{2,r} \geq \epsilon/162^\kappa$.

Note that it is also possible that neither of the above cases is satisfied and that, rather, for *infinitely many* κ , $S_r(\kappa) \cdot T_{2,r}(\kappa) \geq \epsilon(\kappa) \cdot 2^\kappa/16$, and *simultaneously*, for *infinitely many* κ , $S_r(\kappa) \cdot T_{2,r}(\kappa) < \epsilon(\kappa) \cdot 2^\kappa/16$. If this occurs, we obtain our factoring algorithm by having the unbounded pre-processing stage of the factoring algorithm 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 $S_r(\kappa) \cdot T_{r,2}(\kappa) \geq \epsilon(\kappa) \cdot 2^\kappa/16$ or $S_r(\kappa) \cdot T_{r,2}(\kappa) < \epsilon(\kappa) \cdot 2^\kappa/16$. If the former is true, it will append a “0” bit to the preprocessing advice *state* to tell the online portion of the factoring algorithm to run the factoring algorithm for the first case. 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 for the second case. 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 , $\mathbf{Adv}_{\text{RSAGen}}^{\text{fac}}(B)$ remain the same and therefore satisfy the required constraints of Theorem 11.

4.2 The plain model result

► **Theorem 12.** Let $A = (A_0^\pi, A_1)$ be an $(S_r, T_{1,r}, T_{2,r})$ -GP-RSA algorithm relative to RSAGen , and let $\epsilon := \text{Adv}_{\text{RSAGen}}^{\text{ag-rsa}}(A)$.

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

$$\text{Adv}_{\text{RSAGen}}^{\text{fac}}(B) \in \Omega(\epsilon^6),$$

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

To prove Theorem 12 we start from the RI model factoring algorithm obtained in Theorem 7 and prove the following theorem:

► **Theorem 13.** Let $S_r := S_r(\kappa)$, $T_{1,r} := T_{1,r}(\kappa)$, $T_{2,r} := T_{2,r}(\kappa)$, $\epsilon = \epsilon(\kappa)$. Let $A = (A_0^\pi, A_1)$ be an $(S_r, T_{1,r}, T_{2,r})$ -GP-RSA algorithm relative to RSAGen , and let $\epsilon := \text{Adv}_{\text{RSAGen}}^{\text{ag-rsa}}(A)$.

There exists a constant c such that, if for sufficiently large κ , $S_r \cdot T_{2,r} \leq c \cdot \epsilon^6 2^\kappa$, then there exists a (S_f, T_f) -factoring algorithm B in the plain model relative to RSAGen such that

$$\text{Adv}_{\text{RSAGen}}^{\text{fac}}(B) \in \Omega(\epsilon^6),$$

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

The proof of Theorem 13 appears in the full version [11]. To obtain an algorithm for the case that for sufficiently large κ , $S_r \cdot T_{2,r} \geq c \cdot \epsilon^6 2^\kappa$, we set $\tilde{S} = S_r$, $\tilde{T} = T_{2,r}$, $\tilde{\epsilon} = c \cdot \epsilon^6$ and apply Theorem 10. This yields a plain model factoring algorithm with parameters $S_f = S_r$, $T_f = \text{poly}(\kappa) \cdot T_{2,r}^2$, and advantage $\epsilon_f \in \Omega(\epsilon^6)$. Using the same argument as in Section 4.1, we obtain a single factoring algorithm that covers all cases in the plain model with the parameters of Theorem 12.

Two Events. Fix A, N, e, π and *state* as in Lemma 5 in [11]. We consider the probability of two events over the randomness of ComGRA and choice of $y \leftarrow \mathbb{Z}_N$. Set $J := \frac{(1-\epsilon'/2)\epsilon' \cdot N}{8 \log NT_{2,r}} = \frac{(1-\epsilon'/2) \cdot N}{4R_1 T_{2,r}} = N \cdot \delta$, where $\delta := J/N$.

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

► **Corollary 14** (of Lemma 5 in [11]). *Suppose that the conditions of Lemma 5 in [11] hold. Then at least one of the events $E[N, e, \text{state}, \pi]_1$ or $E[N, e, \text{state}, \pi]_2$ occurs with probability at least $\epsilon/4$.*

Looking ahead, if $E[N, e, \text{state}, \pi]_1$ occurs, then A will be useless for factoring. Our task, therefore, is to prove that $E[N, e, \text{state}, \pi]_1$ occurs with probability less than $\epsilon' = \epsilon/4$ (which we do in the full version [11] via a compression argument). We therefore conclude that $E[N, e, \text{state}, \pi]_2$ occurs with probability at least $\epsilon' = \epsilon/4$.

References

- 1 Divesh Aggarwal and Ueli Maurer. Breaking RSA generically is equivalent to factoring. In Antoine Joux, editor, *EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 36–53. Springer, Heidelberg, April 2009. doi:10.1007/978-3-642-01001-9_2.

- 2 Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993. doi:10.1145/168588.168596.
- 3 Daniel J. Bernstein and Tanja Lange. Non-uniform cracks in the concrete: The power of free precomputation. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 321–340. Springer, Heidelberg, December 2013. doi:10.1007/978-3-642-42045-0_17.
- 4 Dan Boneh. Twenty years of attacks on the rsa cryptosystem. *Notices of the American Mathematical Society (AMS)*, 46(2):203–213, 1999.
- 5 Dan Boneh and Ramarathnam Venkatesan. Breaking RSA may not be equivalent to factoring. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 59–71. Springer, Heidelberg, May / June 1998. doi:10.1007/BFb0054117.
- 6 D. R. L. Brown. Breaking rsa may be as difficult as factoring. *Eprint Cryptology Archive*, 2006.
- 7 Don Coppersmith. Modifications to the number field sieve. *J. Cryptol.*, 6(3):169–180, 1993.
- 8 Sandro Coretti, Yevgeniy Dodis, and Siyao Guo. Non-uniform bounds in the random-permutation, ideal-cipher, and generic-group models. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 693–721. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96884-1_23.
- 9 Sandro Coretti, Yevgeniy Dodis, Siyao Guo, and John P. Steinberger. Random oracles and non-uniformity. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 227–258. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78381-9_9.
- 10 Henry Corrigan-Gibbs and Dmitry Kogan. The discrete-logarithm problem with preprocessing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 415–447. Springer, Heidelberg, April / May 2018. doi:10.1007/978-3-319-78375-8_14.
- 11 Dana Dachman-Soled, Julian Loss, and Adam O'Neill. Breaking rsa generically is equivalent to factoring, with preprocessing. *Cryptology ePrint Archive*, Paper 2022/1261, 2022. URL: <https://eprint.iacr.org/2022/1261>.
- 12 Ivan Damgård and Maciej Koprowski. Generic lower bounds for root extraction and signature schemes in general groups. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 256–271. Springer, Heidelberg, April / May 2002. doi:10.1007/3-540-46035-7_17.
- 13 Anindya De, Luca Trevisan, and Madhur Tulsiani. Time space tradeoffs for attacks against one-way functions and prgs. In Tal Rabin, editor, *Advances in Cryptology – CRYPTO 2010*, pages 649–665, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- 14 Yevgeniy Dodis, Siyao Guo, and Jonathan Katz. Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 473–495. Springer, Heidelberg, April / May 2017. doi:10.1007/978-3-319-56614-6_16.
- 15 Yevgeniy Dodis, Iftach Haitner, and Aris Tentes. On the instantiability of hash-and-sign RSA signatures. In Ronald Cramer, editor, *TCC 2012*, volume 7194 of *LNCS*, pages 112–132. Springer, Heidelberg, March 2012. doi:10.1007/978-3-642-28914-9_7.
- 16 Andrew Drucker. New limits to classical and quantum instance compression. In *53rd FOCS*, pages 609–618. IEEE Computer Society Press, October 2012. doi:10.1109/FOCS.2012.71.
- 17 Amos Fiat and Moni Naor. Rigorous time/space tradeoffs for inverting functions. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, May 5-8, 1991, New Orleans, Louisiana, USA*, pages 534–541, January 1991. doi:10.1145/103418.103473.

- 18 Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 33–62. Springer, Heidelberg, August 2018. doi:10.1007/978-3-319-96881-0_2.
- 19 Martin E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theory*, 26(4):401–406, 1980.
- 20 Antoine Joux, David Naccache, and Emmanuel Thomé. When e -th roots become easier than factoring. In Kaoru Kurosawa, editor, *ASIACRYPT 2007*, volume 4833 of *LNCS*, pages 13–28. Springer, Heidelberg, December 2007. doi:10.1007/978-3-540-76900-2_2.
- 21 Jonathan Katz, Julian Loss, and Jiayu Xu. On the security of time-lock puzzles and timed commitments. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 390–413. Springer, Heidelberg, November 2020. doi:10.1007/978-3-030-64381-2_14.
- 22 Gregor Leander and Andy Rupp. On the equivalence of RSA and factoring regarding generic ring algorithms. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT 2006*, volume 4284 of *LNCS*, pages 241–251. Springer, Heidelberg, December 2006. doi:10.1007/11935230_16.
- 23 Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2), 1988.
- 24 Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, December 2005.
- 25 V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.
- 26 Ronald L. Rivest, Adi Shamir, and Leonard 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 Ronald L. Rivest, Adi Shamir, and David A. Wagner. Time-lock puzzles and timed-release crypto. Technical report, MIT, 1996.
- 28 Lior Rotem. Revisiting the uber assumption in the algebraic group model: Fine-grained bounds in hidden-order groups and improved reductions in bilinear groups. In Dana Dachman-Soled, editor, *3rd Conference on Information-Theoretic Cryptography, ITC 2022, July 5-7, 2022, Cambridge, MA, USA*, volume 230 of *LIPICs*, pages 13:1–13:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022.
- 29 Lior Rotem and Gil Segev. Generically speeding-up repeated squaring is equivalent to factoring: Sharp thresholds for all generic-ring delay functions. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 481–509. Springer, Heidelberg, August 2020. doi:10.1007/978-3-030-56877-1_17.
- 30 Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997. doi:10.1007/3-540-69053-0_18.
- 31 Aron van Baarsen and Marc Stevens. On time-lock cryptographic assumptions in abelian hidden-order groups. In *ASIACRYPT*, pages 367–397, 2021.
- 32 Mark Zhandry. To label, or not to label (in generic groups). In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part III*, volume 13509 of *Lecture Notes in Computer Science*, pages 66–96. Springer, 2022.

A Relevant Problems

In this section, 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, $ed = 1 \bmod \phi(N)$.

8:18 Breaking RSA Generically Is Equivalent to Factoring, with Preprocessing

► **Definition 15** (Factoring with Preprocessing). Let $F = (F_0, F_1)$ be an algorithm and RSAGen be an RSA generator. Consider the factoring-with-preprocessing game $\text{fac}_{\text{RSAGen}}^F$:

- **Offline Phase.** Run F_0 on input 1^κ to obtain an advice string state.
- **Online Phase.** Run RSAGen on input 1^κ to obtain (N, e, d, p, q) . Then run F_1 on input (N, state) .
- **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

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

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 16** (RSA with Preprocessing). Let $A = (A_0, A_1)$ be an adversary. Consider the RSA-with-preprocessing game $\text{rsa}_{\text{RSAGen}}^A$:

- **Offline Phase.** Run A_0 on input 1^κ to obtain an advice string state.
- **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 $(N, e, \text{state}, x^e \bmod N)$.
- **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

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

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 17** (Function Inversion with Preprocessing). Let D be a finite set and let $f: D \rightarrow D$ be a function. Let $l = (l_0, l_1)$ be an adversary and Gen a point generator. Consider the function-inversion-with-preprocessing game $\delta_{f, \text{Gen}}^l$:

- **Offline Phase.** Run l_0 on input 1^κ to obtain an advice string state.
- **Online Phase.** Run Gen on input 1^κ to obtain a point $y \in D$. Run l_1 on input (y, state)
- **Output Determination.** When l_1 returns x' , the experiment returns 1 if $f(x') = y$. It returns 0 otherwise.

Define l 's advantage in the above experiment as

$$\text{Adv}_{f, \text{Gen}}^\delta(l) = \Pr[\delta_{f, \text{Gen}}^l = 1].$$

We call l an (S, T) -function-inversion algorithm relative to Gen if l_0 outputs advice strings of size at most S and l_1 runs in time at most T .

► **Definition 18** (Collision Probability). Let D be a finite set and let $f: D \rightarrow D$ 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 $f: D \rightarrow D$, denoted by $q(f)$ is defined as follows:

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

► **Theorem 19** (Fiat-Naor [17]). *For any D, f, Gen as in Definition 17 and any S, T such that $T \cdot S^2 = |D|^3 \cdot q(f)$, there is an (S, T) -function-inversion algorithm I such that $\mathbf{Adv}_{f, \text{Gen}}^\delta(\mathsf{I}) \geq 1 - 1/|D|^2$.*

B Computational Models

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

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

B.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 [23].

Luby-Rackoff. We first recall the Luby-Rackoff construction [23], 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 $\text{LubRac}[\rho]$ 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 $\text{LubRack}^{-1}[\rho]$ is defined accordingly.

► **Theorem 20** (Luby-Rackoff [23]). *For any (even unbounded) adversary A making at most q queries it holds that*

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

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 $\pi[\psi]: \{0, 1\}^n \rightarrow \{0, 1\}^m$ as $\pi[\psi](x) := \psi(\text{pad}(x))$ where $\text{pad}(x)$ 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.

² This statement is weaker than the one proven in [17] but is sufficient for our purpose.

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

► **Proposition 21 (RIM-to-ROM).** *For any (even unbounded) adversary A making at most q queries it holds that*

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

B.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 k th 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 k th 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 i th and j th 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 $y_j^b = -y_j$ (mod N). In case $\circ = \cdot, b = -1$, and $y_j = 0$ (mod N), $y_k := \perp$. In case $y_i = \perp$ or $y_j = \perp$, $y_k := \perp$. It continues execution along this path.

- If the k th vertex v_k is labelled (i, j) where $i, j < k$ correspond to the i th and j th 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 $\pi \in \text{FuncInj}[\kappa, m]$ 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 $\pi \in \text{FuncInj}[\kappa, m]$, 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

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

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 $\text{eq}(y_2, 0)$. op_π returns \perp in case either of the operands is \perp or the call to eq returns 1, i.e., if $\pi^{-1}(y_2) = 0 \pmod{N}$.

► **Remark 22.** Throughout the paper, when there is no possibility of confusion we abbreviate oracles op_π and eq_π by π . That is, for an oracle-aided program P we abbreviate $P^{\text{op}_\pi, \text{eq}_\pi}$ by P^π .

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 i th and j th 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 i th and j th 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 k th 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 .

8:22 Breaking RSA Generically Is Equivalent to Factoring, with Preprocessing

- If v is labelled with σ and this is the k th 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 23.** 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, \dots, 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 24.** 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 25.** 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 26.** 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 27.** 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

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

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

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

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

B.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 28** (Generic RSA Problem with Preprocessing). For a tuple of algorithms $A = (A_0^\pi, A_1)$ and an RSA instance generator RSAGen , define experiment $\text{ag-rsa}_{\text{RSAGen}}^A$ as follows:

- **Offline Phase.** Sample $\pi \leftarrow \text{FuncInj}[\kappa, m]$. Run A_0^π on input 1^κ . Let state denote the return value of A_0^π .
- **Online Phase.** Compute $(N, e, d) \leftarrow \text{RSAGen}(1^\kappa)$ and sample $x \leftarrow \mathbb{Z}_N$. Run A_1 on input $(N, e, \pi(x^e), \text{state})$ 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

$$\text{Adv}_{\text{RSAGen}}^{\text{ag-rsa}}(A) = \Pr[\text{ag-rsa}_{\text{RSAGen}}^A = 1].$$

We call $A = (A_0^\pi, A_1)$ an (S, T_1, T_2) -generic-RSA-with-preprocessing algorithm (GP-RSA) relative to RSAGen if A_0^π outputs advice strings state of size at most S , A_1 runs in time at most T_1 , 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 $\pi \in \text{FuncInj}[\kappa, m]$ and $(N, e, d) \in \text{RSAGen}(1^\kappa)$ are a fixed.

► **Definition 29** (Fixed Generic RSA Problem with Preprocessing). Fix integers $(N, e, d) \in \text{RSAGen}(1^\kappa)$, let $\pi \in \text{FuncInj}[\kappa, m]$, and let state be of size at most S . Define experiment $\text{fcrsa}_{(N, e, d, \text{state}, \pi)}^A$ as follows:

- **Online Phase.** Sample $x \leftarrow \mathbb{Z}_N$. Run A on input $(N, e, \pi(x^e), \text{state})$ 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 .

8:24 Breaking RSA Generically Is Equivalent to Factoring, *with Preprocessing*

- **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 $(N, e, d, state, \pi)$ as

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

We call A an (S, T_1, T_2) -fixed-generic-RSA-with-preprocessing (FGP-RSA) algorithm relative to $(N, e, d, state, \pi)$ if A runs in time at most T_1 , 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 $state$ to be output by a preprocessor A_0^π . However, by a standard averaging argument, we obtain the following lemma:

- **Lemma 30.** Let $A = (A_0^\pi, A_1)$ be an (S, T_1, T_2) -GP-RSA algorithm and suppose that $\mathbf{Adv}_{\text{RSAGen}}^{\text{ag-rsa}}(A) \geq \epsilon$. 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 $state$ and RSAGen outputs (N, e, d) s.t. $\mathbf{Adv}_{(N,e,d,state,\pi)}^{\text{fcrsa}}(A_1) \geq \epsilon/2$.