An extended abstract of this paper was published in the proceedings of CT-RSA 2012.

Efficient RSA Key Generation and Threshold Paillier in the Two-Party Setting

Carmit Hazay* Gert Læssøe Mikkelsen†‡ Tal Rabin§ Tomas Toft†
Angelo Agatino Nicolosi¶

Abstract

The problem of generating an RSA composite in a distributed manner without leaking its factorization is particularly challenging and useful in many cryptographic protocols. Our first contribution is the first non-generic *fully simulatable* protocol for distributively generating an RSA composite with security against malicious behavior. Our second contribution is a *complete* Paillier [Pai99] threshold encryption scheme in the two-party setting with security against malicious attacks. We further describe how to extend our protocols to the multiparty setting with dishonest majority.

Our RSA key generation protocol is comprised of the following sub-protocols: (**i**) a distributed protocol for generation of an RSA composite, and (**ii**) a biprimality test for verifying the validity of the generated composite. Our Paillier threshold encryption scheme uses the RSA composite for the public-key and is comprised of the following sub-protocols: (**i**) a distributed generation of the corresponding secret-key shares and, (**ii**) a distributed decryption protocol for decrypting according to Paillier.

Keywords: Secure Two-Party Computation, RSA Generation, Threshold Encryption Scheme, Paillier

*Faculty of Engineering, Bar-Ilan University, Israel. Email: carmit.hazay@biu.ac.il
†Department of Computer Science, Aarhus University, Denmark. Email: gertm@cs.au.dk, ttoft@cs.au.dk.
‡The Alexandra Institute, Email: gert.l.mikkelsen@alexandra.dk
§IBM T.J.Watson Research Center, Email: talr@us.ibm.com.
¶YanchWare, Email: angelonicolosi@yanchware.com.

1 Introduction

1.1 Distributed Generation of an RSA Composite

Generating an RSA composite N (a product of two primes p and q), and secret keying material (values related to $\phi(N)$) in a distributed manner is an important question in secure computation. Many cryptographic protocols require such a composite for which none of the parties knows N's prime factorization. A concrete example where such a protocol is very useful is *threshold cryptography*, where a number of parties exceeding some threshold is required to cooperate in order to carry out a cryptographic task such as computing a decryption or a signature; see [Rab98, Des94, Sho00, CDN01] for just few examples. Specifically, the public-key of Paillier [Pai99], which is one of the most widely used public-key encryption schemes in secure computation, is an RSA composite. Paillier is an important building block due to the fact that it is an additively homomorphic cryptosystem, and is therefore extremely useful. Consequently, it is very important to design an efficient threshold variant for this construction. Another important application is using RSA composites as part of the common reference string (CRS) for secure computation in the UC setting [Can01], as demonstrated for any function in [JS07] or for concrete functions such as the Fiat-Shamir authentication protocol [FS86, FFS88], set-intersection [JL09] and oblivious pseudorandom functions [JL09].

Typically, generating an RSA composite is comprised from two phases; generating a candidate and testing its validity (namely, that it is indeed a product of two primes). This task has proven particularly challenging mainly due to lack of an efficient secure implementation of the later phase. Therefore, most prior works assume that the composite is generated by a *trusted dealer*. In a breakthrough result, Boneh and Franklin [BF01] showed a mathematical method for choosing a composite and verifying that it is of the proper form. Based on this method they designed a protocol in the multiparty setting with security against semi-honest adversaries, assuming honest majority (where a semi-honest adversary follows the protocol's instructions honestly but tries to learn secret information about the honest parties' inputs from the communication). Two followup papers [FMY98, NS10] strengthened this result and obtained security against malicious adversaries (where a malicious adversary follows an arbitrary polynomial-time strategy). We note that the later result relies on a random oracle. Additional solutions for testing primality in the multiparty setting appear in [ACS02, DM10].

Security without assuming honest majority, and in particular in the two-party setting, posed additional barriers even for the semi-honest model. Cocks [Coc97] initiated the study of the shared generation of the RSA composite in the two-party semi-honest model. Nevertheless, his proposed protocol was later found to be insecure [Cop97, BBBG98]. The problem was solved by Gilboa [Gil99] who presented a protocol in the semi-honest model, adapting the [BF01] technique.

In the malicious setting, Blackburn et al. [BBBG98] started examining the setting of an arbitrary adversary, yet they did not provide a proof of security for their protocol. Concurrently, Poupard and Stern [PS98] proposed a solution that runs in time proportional to the size of the domain from which the primes are sampled, which is exponential in the security parameter. Poupard and Stern made attempts to reduce the running time by introducing various modifications. However those are not proven, and as they leak some information, presenting a proof of security (if at all possible) will not be a trivial thing. A detailed explanation about this construction and a discussion about the proof complexity appear in Appendix A. This overview implies that all prior results do not offer an efficient and provable solution for the two-party malicious case.

1.2 Threshold Cryptosystems of Paillier

A threshold cryptosystem is typically involved with two related yet separable components; (**1**) a distributed generation of the public-key and sharing of the corresponding secret key of the cryptosystem, and (**2**) a decryption/signature computation from a shared representation of the secret key. Threshold schemes are

important in settings where no individual party should know the secret key. Prior to this work, solutions for distributed discrete log-based key generation systems [GJKR07], and threshold encryption/decryption for RSA [GKR00, Sho00], DSS [GJKR01] and Paillier [FPS00, DJ01, BFP+01] in the multiparty setting, have been presented. For some cryptosystems (e.g., ElGamal [ElG85]) the techniques from the multiparty setting can be adapted to the two-party case in a relatively straightforward manner. However, a solution for a two-party threshold Paillier encryption scheme that is resistant against malicious attacks has proven more complex and elusive.

Elaborating on these difficulties, we recall that the RSA and Paillier encryption schemes share the same public/secret keys format of a composite N and its factorization. Moreover, the ciphertexts have a similar algebraic structure. Thus, it may seem that decryption according to Paillier should follow from the (distributed) algorithm of RSA, as further discussed in [CGHN01]. Nevertheless, when decrypting as in the RSA case (i.e., raising the ciphertext to the power of the inverse of N modulo the unknown order), the decrypter must extract the randomness of the ciphertext in order to complete the decryption. This property is problematic in the context of simulation based security, because it forces to reveal the randomness of the ciphertext and does not allow cheating in the decryption protocol. We further recall that by definition, Paillier's scheme requires an extra computation in order to complete the decryption (on top of raising the ciphertext to the power of the secret value) since the outcome from this computation is the plaintext multiplied with this secret value. In the distributive setting this implies that the parties must keep two types of shares. Due to these reasons it has been particularly challenging to design a complete threshold system for Paillier in the two-party setting without the help of a trusted party.

1.3 Our Contribution

In this work, we present the first *fully simulatable* and *complete* RSA key generation and Paillier [Pai99] threshold scheme in the two-party malicious setting. Namely, we define the appropriate functionalities and prove that our protocols securely realize them. Our formalization further takes into account a subtle issue in the public-key generation, which was initially noticed by Boneh and Franklin [BF01]. Informally, they showed that their protocol leaks a certain amount of information about the product, and proved that it does not pose any practical threat for the security. Nevertheless, it does pose a problem when simulating since the adversary can influence the *distribution* of the generated public-key. We therefore also work with a slightly modified version of the natural definition for a threshold encryption functionality; see the formal definition in Section 4. In more details, our scheme is comprised of the following protocols:

1. **A distributed generation of an RSA composite.** We present the first fully simulatable protocol for securely computing a Blum integer composite as a product of two primes without leaking information about its factorization (in the sense of [BF01]). Our protocol follows the outlines of [BF01] and improves the construction suggested by [Gil99] in terms of security level and efficiency; namely we use an additional trial division protocol on the individual prime candidates which enables us to exclude many of them earlier in the computation. We note that [Gil99] did not implement the trial division as part of his protocol, thus our solution is the first for the two-party setting that employs this test. Here we take a novel approach of utilizing two different additively homomorphic encryption schemes, that enable to ensure active security at a very low cost. In Appendix C we further show how to extend this protocol and the biprimality test to the multiparty setting with dishonest majority, presenting the first actively secure k parties for an RSA generation protocol, that tolerates up to $k-1$ corruptions.

2. **A distributed biprimality test.** We adopt the biprimality test proposed by [BF01] into the malicious two-party setting and provide a proof of security for this protocol. This test essentially verifies whether the generated composite is of the correct form (i.e., it is a product of exactly two primes of an appropriate length). We further note that the biprimality test by Damgård and Mikkelsen [DM10] has a

better error estimate, yet it cannot be used directly in the two-party setting with malicious adversaries. In Appendix B we adapt their test into the two-party setting when the parties are semi-honest.

3. **Distributed generations of the secret key shares.** Motivated by the discussion above, we present a protocol for generating shares for a decryption key of the form $d \equiv 1 \bmod N \equiv 0 \bmod \phi(N)$. Specifically, we take the same approach of Damgård and Jurik [DJ01], except that in their threshold construction the shares are generated by a trusted party. We present the first concrete protocol for this task with semi-honest security, and then show how to adapt it into the malicious setting.

4. **A distributed decryption.** Finally, we present a distributed protocol for decrypting according to the decryption protocol of Damgård and Jurik [DJ01], but for two-parties. Namely, each party raises the ciphertext to the power of its share and proves consistency. We remark that even though our decryption protocol is similar to that of [DJ01], there are a few crucial differences: (**i**) First, since we only consider two parties, an additive secret sharing suffices. (**ii**) Secret sharing is done over the integers rather than attempting to perform a reduction modulo the secret value $\phi(N) \cdot N$. (**iii**) Finally, the Damgård-Jurik decryption protocol requires N to be a product of safe-primes to ensure hardness of discrete logarithms. To avoid this requirement, we ensure equality of discrete logarithms in different-order groups.

1.4 Efficiency

Distributed RSA Composite. We provide a detailed efficiency analysis for all our subprotocols in Section 6. All our subprotocols are round efficient due to parallelization of the generation and testing the potential RSA composite. This includes the biprimality test and trial division. Moreover, all our zero-knowledge proofs run in constant rounds and require constant number of exponentiations (except one that achieves constant complexity on the average). We further note that the probability of finding a random prime is independent of the method in which it is generated, but rather only depends on the density of the primes in a given range. We show that due to our optimizations we are able to implement the trial division which greatly reduces the number of candidates that are expected to be tested, e.g. for a 512 bit prime we would need to test about 31,000 candidates without the trial division, while the number of candidates is reduced to 484 with the trial division. We give a detailed description of our optimizations in Section 6.

The only alternative to distributively generate an RSA composite with malicious security is using generic protocols that evaluate a binary or an arithmetic circuit. In this case the circuit must sample the primes candidates first and test their primality, and finally compute their product. The size of a circuit that tests primality is polynomial in the length of the prime candidate. Furthermore, the best known binary circuit that computes the multiplication of two numbers of length n requires $O(n \log n)$ gates. This implies a circuit of polynomial size in n, multiplied with $O(n^2)$ which is the required number of trials for successfully sampling two primes with overwhelming probability. Moreover, generic secure protocols are typically proven in the presence of malicious attacks using the cut-and-choose technique, which requires sending multiple instances of the garbled circuit [LP11]. Specifically, the parties repeat the computation a number of times such that half of these computations are examined in order to ensure correctness. The best cut-and-choose analysis is due to [Lin13] which requires that the parties exchange 40 circuits (with some additional overhead). This technique inflates the communication and computation overheads.

Protocols that employ arithmetic circuits [BDOZ11b, DPSZ12] work in the preprocessing model where the parties first prepare a number of triples of multiplications that is proportional to the circuit's size, and then use them for the circuit evaluation in the online phase. Therefore the number of triples is proportional to the size of the circuit that tests primality times the number of trials. On the other hand, our protocol presents a relatively simpler approach with a direct design of a key generation protocol without building

a binary/arithmetic circuit first, which is fairly complicated for this task. This follows for the multi-party setting as well since our protocol is the first protocol that achieves security in the malicious setting.

Threshold Paillier. This phase is comprised out of two protocols. First, the generation of the multiplicative key shares protocol, which is executed only once, and requires constant overhead and constant round complexity. The second protocol for threshold decryption is dominated by the invocation of a zero-knowledge proof which requires constant number of exponentiations for long enough challenge (see more discussion about this proof in Section 3.2, Item 5). For batch decryption the technique of Cramer and Damgård [CD09] can be used to achieve amortized constant overhead. Our protocols are the first secure protocols in the two-party malicious setting.

Additional practical considerations are demonstrated in Section 6.3.

1.5 Experimental Results

We further present an implementation of Protocol 1 with security against semi-honest adversaries. Our primary goal is to examine the overall time it takes to generate an RSA composite of length 2048 bits in case the parties do not deviate, and identify bottlenecks. The bulk of our implementation work is related to reducing the computational overhead of the trial division phase which is the most costly phase. Namely, based on our study of the resources required by the parties for generating a legal RSA composite, we concluded that the DKeyGen protocol is too expensive for real world applications (at least 3 hours running time). We therefore focused on algorithmic improvements of the performance of this phase, which lead to two optimizations: protocol *BatchedDec* and protocol *LocalTrialDiv*; details below. In addition, to improve the overhead relative to the ElGamal PKE, we implement this scheme on elliptic curves using the MIRACL library [MIR] and use the elliptic curve *P-192* [FIP09] based on Ecrypt II yearly report [ECR11]; see [INI99, SEC00] for more practical information. A brief background on elliptic curves in found in Section 7.1.1.

BatchedDec protocol The underlying idea of the first optimization is due to the performance analysis of the decryption operation carried out within our composite generation protocol (executed within the trial division protocol). Specifically, we observed that the decryption operation is the most expensive cryptographic operation. To to improve this overhead we modified the decryption protocol in order to let the parties collect the results of several trial division rounds and decrypt all the results at once when needed. This implementation improved the overhead by much and required 40 minutes on the average in order to generate a 2048 bits RSA composite.

LocalTrialDiv protocol The second optimization is focused on reducing the usage of expensive cryptographic operations by taking alternative cheaper approaches. This protocol variant permits to securely compute a 2048 bits RSA composite in 15 minutes on the average.

Security for our two optimizations follows easily. Our detailed results can be found in Section 7.

2 Preliminaries

We denote the security parameter by n. A function $\mu(\cdot)$ is negligible in n (or just negligible) if for every polynomial $p(\cdot)$ there exists a value m such that for all $n > m$ it holds that $\mu(n) < \frac{1}{p(n)}$. Let $X = \{X(n,a)\}_{n\in\mathbb{N},a\in\{0,1\}^*}$ and $Y = \{Y(n,a)\}_{n\in\mathbb{N},a\in\{0,1\}^*}$ be distribution ensembles. Then, we say that X

and Y are computationally indistinguishable, denoted $X \stackrel{c}{\equiv} Y$, if for every non-uniform probabilistic polynomial-time distinguisher D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$,

$$|\Pr[D(X(n,a)) = 1] - \Pr[D(Y(n,a)) = 1]| < \mu(n)$$

We adopt the convention whereby a machine is said to run in polynomial-time if its number of steps is polynomial in its *security parameter*. We use the shorthand PPT to denote probabilistic polynomial-time.

2.1 Hardness Assumptions

Our constructions rely on the following hardness assumptions.

Definition 2.1 (DDH) *We say that* the decisional Diffie-Hellman (DDH) problem is hard relative to $\mathbb{G} = \{\mathbb{G}_n\}$ *if for any PPT algorithm $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\left| \Pr\left[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1\right] - \Pr\left[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1\right] \right| \leq \mathsf{negl}(n),$$

where q is the order of $\mathbb{G}$ and the probabilities are taken over the choices of g and $x, y, z \in \mathbb{Z}_q$.

Definition 2.2 (DCR) *We say that* the decisional composite residuosity (DCR) problem is hard *if for any PPT algorithm $\mathcal{A}$ there exists a negligible function* negl *such that*

$$\left| \Pr\left[\mathcal{A}(N, z) = 1 | z = y^N \bmod N^2\right] - \Pr\left[\mathcal{A}(N, z) = 1 | z = (N+1)^r \cdot y^N \bmod N^2\right] \right| \leq \mathsf{negl}(n),$$

where N is a random n-bit RSA composite, r is chosen at random in $\mathbb{Z}_N$, and the probabilities are taken over the choices of N, y and r.

2.2 Public-Key Encryption Schemes

We begin by specifying the definitions of public-key encryption and IND-CPA security. We conclude with a definition of homomorphic encryption, specifying two encryption schemes that meet this definition.

Definition 2.3 (PKE) *We say that* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a* public-key encryption scheme *if* $\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}$ *are polynomial-time algorithms specified as follows:*

- Gen*, given a security parameter n (in unary), outputs keys (pk, sk), where pk is a public-key and sk is a secret key. We denote this by $(pk, sk) \leftarrow \mathsf{Gen}(1^n)$.*
- Enc*, given the public-key pk and a plaintext message m, outputs a ciphertext c encrypting m. We denote this by $c \leftarrow \mathsf{Enc}_{pk}(m)$; and when emphasizing the randomness r used for encryption, we denote this by $c \leftarrow \mathsf{Enc}_{pk}(m; r)$.*
- Dec*, given the public-key pk, secret key sk and a ciphertext c, outputs a plaintext message m s.t. there exists randomness r for which $c = \mathsf{Enc}_{pk}(m; r)$ (or $\perp$ if no such message exists). We denote this by $m \leftarrow \mathsf{Dec}_{pk,sk}(c)$.*

For a public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and a non-uniform adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, we consider the following *IND-CPA game*:

$(pk, sk) \leftarrow \mathsf{Gen}(1^n)$.
$(m_0, m_1, history) \leftarrow \mathcal{A}_1(pk)$, s.t. $|m_0| = |m_1|$.
$c \leftarrow \mathsf{Enc}_{pk}(m_b)$, where $b \in_R \{0,1\}$.
$b' \leftarrow \mathcal{A}_2(c, history)$.
$\mathcal{A}$ wins if $b' = b$.

Denote by $\mathsf{Adv}_{\Pi,\mathcal{A}}(n)$ the probability that $\mathcal{A}$ wins the IND-CPA game.

Definition 2.4 (IND-CPA) *A public-key encryption scheme* $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* semantically secure*, if for every non-uniform adversary* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *there exists a negligible function* negl *such that* $\mathsf{Adv}_{\Pi,\mathcal{A}}(n) \leq \frac{1}{2} + \mathsf{negl}(n)$.

An important tool that we exploit in our construction is *homomorphic* encryption over an additive group as defined below.

Definition 2.5 (Homomorphic PKE) *A public-key encryption scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is* homomorphic *if for all* n *and all* (pk, sk) *output by* $\mathsf{Gen}(1^n)$*, it is possible to define groups* $\mathbb{M}, \mathbb{C}$ *such that:*

- *The plaintext space is* $\mathbb{M}$*, and all ciphertexts output by* Enc_{pk} *are elements of* $\mathbb{C}$*.*
- *For any* $m_1, m_2 \in \mathbb{M}$ *and* $c_1, c_2 \in \mathbb{C}$ *with* $m_1 = \mathsf{Dec}_{sk}(c_1)$ *and* $m_2 = \mathsf{Dec}_{sk}(c_2)$*, it holds that*

$$\{pk, c_1, c_1 \cdot c_2\} \equiv \{pk, \mathsf{Enc}_{pk}(m_1), \mathsf{Enc}_{pk}(m_1 + m_2)\}$$

  where the group operations are carried out in $\mathbb{C}$ *and* $\mathbb{M}$*, respectively, and the encryptions of* m_1 *and* $m_1 + m_2$ *use independent randomness.*

Any additive homomorphic scheme supports the multiplication of a ciphertext by a scalar by computing multiple additions.

2.2.1 The Paillier Encryption Scheme

The Paillier encryption scheme [Pai99] is an example of a public-key encryption scheme that meets Definition 2.5. We focus our attention on the following, widely used, variant of Paillier comprised of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Namely, the key generation algorithm, Gen, chooses two equal length primes p and q and outputs a public-key pk that equals $N = pq$, and a matching secret-key $sk = \phi(N)$. We use a simplified encryption function proposed by Damgård and Jurik [DJ01], where $g = N+1$ is a generator of the subgroup of $\mathbb{Z}_{N^2}$ of order N. Thereby, encryption of a plaintext m with randomness $r \in_R \mathbb{Z}_N^*$ ($\mathbb{Z}_N$ in practice) is computed by,

$$E_N(m, r) = r^N \cdot (N+1)^m \bmod N^2.$$

Finally, decryption is performed by,

$$\mathsf{Dec}_{sk}(c) = \frac{[c^{\phi(N)} \bmod N^2] - 1}{N} \cdot \phi(N)^{-1} \bmod N.$$

The security of Paillier is implied by the Decisional Composite Residuosity Assumption (DCR).

2.2.2 Additively Homomorphic ElGamal Variant

In our protocols we further use an additively homomorphic variation of ElGamal encryption [ElG85]. Namely, let $\mathbb{G}$ be a group generated by g of prime order Q, in which the decisional Diffie-Hellman (DDH) problem is hard. A public-key is then a pair $pk = (g, h)$ and the corresponding secret key is $s = \log_g(h)$, i.e. $g^s = h$. Encryption of a message $m \in \mathbb{Z}_Q$ is defined as $\mathsf{Enc}_{pk}(m; r) = (g^r, h^r \cdot g^m)$ where r is picked uniformly at random in $\mathbb{Z}_Q$. Decryption of a ciphertext (α, β) is then performed as $\mathsf{Dec}_{sk}(\alpha, \beta) = \beta \cdot \alpha^{-s}$. Note that decryption yields g^m rather than m. As the discrete log problem is hard, we cannot in general hope to determine m. Fortunately, we only need to distinguish between zero and non-zero values, hence the

lack of "full" decryption is not an issue. We abuse notation and write $c \cdot c'$ to denote the componentwise multiplication of two ciphertexts c and c', computed by $(\alpha \cdot \alpha', \beta \cdot \beta')$, where $c = (\alpha, \beta)$ and $c' = (\alpha', \beta')$.

We require that the parties run a threshold version for ElGamal for generating a public-key and additive shares for the secret key, as well as a distributive decryption protocol. The key generation construction can be easily obtained based on the Diffie-Hellman protocol [DH76] with additive shares. A distributive decryption follows easily as well. We denote the distributed key generation protocol by π_{GEN} and the distributed decryption protocol by π_{DEC}.

2.3 Integer Commitment Schemes

In order to ensure correct behavior of the parties (and do so efficiently), our key generation protocol utilizes integer commitments which rely on the fact that the committer does not know the order of the group $\mathbb{G}$, denoted by $|\mathbb{G}|$, from which it picks the committed elements. Therefore, it cannot decommit into two different values, such as m and $m + |\mathbb{G}|$. This property is crucial for ensuring that the parties' shares are indeed smaller than some threshold. An example of such a commitment is the Paillier based scheme of Damgård and Nielsen [DN02, DN03], which is comprised of the following two algorithms:

1. SETUP. The receiver, R, generates a Paillier key N, i.e. an RSA modulus. It then picks r at random in $\mathbb{Z}^*_{N^2}$, computes $g = r^N$, and sends N, g to the committing party, C, along with zero-knowledge proofs that N is an RSA modulus and that g is a Paillier encryption of zero.

2. COMMIT/OPEN. To commit to $m \in \mathbb{Z}_N$, C picks r_m at random in $\mathbb{Z}_N$ and computes $\texttt{Com}\,(m; r_m) = g^m \cdot r_m^N$. To open, C simply reveals r_m and m to R.

HIDING/BINDING. The scheme is perfectly hiding, as a commitment is simply a random encryption of zero. Further, opening to two different values implies an Nth root of g (which breaks the underlying assumption of Paillier, i.e., DCR).

2.4 Σ-Protocols

Definition 2.6 (Σ-protocol) *A protocol π is a* Σ-protocol *for relation R if it is a 3-round public-coin protocol and the following requirements hold:*

- COMPLETENESS: *If P and V follow the protocol on input x and private input w to P where $(x, w) \in R$, then V always accepts.*

- SPECIAL SOUNDNESS: *There exists a polynomial-time algorithm A that given any x and any pair of accepting transcripts $(a, e, z), (a, e', z')$ on input x, where $e \neq e'$, outputs w such that $(x, w) \in R$.*

- SPECIAL HONEST-VERIFIER ZERO KNOWLEDGE: *There exists a* PPT *algorithm M such that*

$$\Big\{\langle P(x,w), V(x,e)\rangle\Big\}_{x \in L_R} \equiv \Big\{M(x,e)\Big\}_{x \in L_R}$$

  where $M(x, e)$ denotes the output of M upon input x and e, and $\langle P(x, w), V(x, e)\rangle$ denotes the output transcript of an execution between P and V, where P has input (x, w), V has input x, and V's random tape (determining its query) equals e.

2.5 Security in the Presence of Malicious Adversaries

In this section we briefly present the standard definition for secure two-party computation and refer to [Gol04, Chapter 7] for more details and motivating discussion.

Two-party computation. A two-party protocol problem is cast by specifying a random process that maps pairs of inputs to pairs of outputs (one for each party). We refer to such a process as a functionality and denote it $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$, where $f = (f_1, f_2)$. That is, for every pair of inputs (x, y), the output-vector is a random variable $(f_1(x, y), f_2(x, y)$ ranging over pairs of strings where P_0 receives $f_1(x, y)$ and P_1 receives $f_2(x, y)$. We sometimes denote such a functionality by $f(x, y) \mapsto (f_1(x, y), f_2(x, y))$.

Adversarial behavior. Loosely speaking, the aim of a secure two-party protocol is to protect honest parties against dishonest behavior by other parties. In this section, we outline the definition for *malicious adversaries* who control some subset of the parties and may instruct them to arbitrarily deviate from the specified protocol. We also consider static malicious corruptions.

Security of protocols (informal). The security of a protocol is analyzed by comparing what an adversary can do in a real protocol execution to what it can do in an ideal scenario that is secure by definition. This is formalized by considering an *ideal* computation involving an incorruptible *trusted third party* to whom the parties send their inputs. The trusted party computes the functionality on the inputs and returns to each party its respective output. Loosely speaking, a protocol is secure if any adversary interacting in the real protocol (where no trusted third party exists) can do no more harm than if it was involved in the above-described ideal computation. One technical detail that arises when considering the setting of no honest majority is that it is impossible to achieve fairness or guaranteed output delivery [Cle86]. That is, it is possible for the adversary to prevent the honest party from receiving outputs. Furthermore, it may even be possible for the adversary to receive output while the honest party does not.

Execution in the ideal model. In an ideal execution, the parties send their inputs to the trusted party who computes the output. An honest party just sends the input that it received whereas a corrupted party can replace its input with any other value of the same length. Since we do not consider fairness, the trusted party first sends the output of the corrupted parties to the adversary, and the adversary then decides whether the honest parties receive their (correct) outputs or an abort symbol $\perp$. Let f be a two-party functionality where $f = (f_1, f_2)$, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine, and let $I \subseteq [2]$ be the set of corrupted parties (either P_0 is corrupted or P_1 is corrupted or neither). Then, the ideal execution of f on inputs (x, y), auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\text{IDEAL}_{f,\mathcal{A}(z),I}(x, y, n)$, is defined as the output pair of the honest party and the adversary $\mathcal{A}$ from the above ideal execution.

Execution in the real model. In the real model there is no trusted third party and the parties interact directly. The adversary $\mathcal{A}$ sends all messages in place of the the corrupted party, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of the specified protocol π.

Let f be as above and let π be a two-party protocol for computing f. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine and let I be the set of corrupted parties. Then, the real execution of π on inputs (x, y), auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\text{REAL}_{\pi,\mathcal{A}(z),I}(x, y, n)$, is defined as the output vector of the honest parties and the adversary $\mathcal{A}$ from the real execution of π.

Security as emulation of a real execution in the ideal model. Having defined the ideal and real models, we can now define security of protocols. Loosely speaking, the definition asserts that a secure party protocol (in the real model) emulates the ideal model (in which a trusted party exists). This is formulated by saying that adversaries in the ideal model are able to simulate executions of the real-model protocol.

Definition 2.7 *Let f and π be as above. Protocol π is said to* securely compute f with abort in the presence of malicious adversaries *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model, such that for every $I \subseteq [2]$,*

$$\left\{\text{IDEAL}_{f,\mathcal{S}(z),I}(x,y,n)\right\}_{x,y,z\in\{0,1\}^*,n\in\mathbb{N}} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(x,y,n)\right\}_{x,y,z\in\{0,1\}^*,n\in\mathbb{N}}$$

where $|x| = |y|$.

Reactive functionalities. Until now we have considered the secure computation of simple functionalities that compute a single pair of outputs from a single pair of inputs. However, not all computations are of this type. Rather, many computations have multiple rounds of inputs and outputs. Furthermore, the input of a party in a given round may depend on its output from previous rounds, and the outputs of that round may depend on the inputs provided by the parties in some or all of the previous rounds. In the context of secure computation, multi-phase computations are typically called reactive functionalities. Such functionalities can be modeled as a series of functions $(f^1, f^2, \ldots)$ where each function receives some state information and two new inputs. That is, the input to function f^j consists of the inputs (x_j, y_j) of the parties in this phase, along with a state input σ_{j-1} output by f^{j-1}. Then, the output of f^j is defined to be a pair of outputs $f_1^j(x_j, y_j, \sigma_{j-1})$ for P_1 and $f_2^j(x_j, y_j, \sigma_{j-1})$ for P_2, and a state string σ_j to be input into f^{j+1}. We stress that the parties receive only their private outputs, and in particular do not receive any of the state information; in the ideal model this is stored by the trusted party.

3 Zero-Knowledge Proofs

In order to cope with malicious adversaries, our protocols employ zero-knowledge (ZK) proofs. In this section we provide a description of the proofs we use; while some of these proofs are known, others are new to this work and are interesting by themselves. We note that except from a single proof, all proofs require a strict constant overhead. Fortunately, since our proofs are employed for multiple instances, the analysis of [CD09] ensures that the average overhead is constant; details follow.

3.1 Discrete Logarithms

1. The following Σ-protocol, denoted by π_{DL}, demonstrates knowledge of a discrete logarithm. The proof follows due to Schnorr [Sch91].

$$\mathcal{R}_{\text{DL}} = \{((\mathbb{G}, g, h), w) \mid h = g^w\}.$$

2. The Σ-protocol π_{DH} demonstrates that a quadruple (g_0, g_1, h_0, h_1) is a Diffie-Hellman tuple, i.e. that $\log_{g_0}(h_0) = \log_{g_1}(h_1)$ for $g_i, h_i \in \mathbb{G}$. This proof is due to Chaum and Pedersen [CP92].

$$\mathcal{R}_{\text{DH}} = \{((\mathbb{G}, g_0, g_1, h_0, h_1)\, w) \mid h_i = g_i^w \text{ for } i \in \{0,1\}\}.$$

3.2 Plaintext Relations

1. Protocol π_{ENC} demonstrates knowledge of an encrypted plaintext.

$$\mathcal{R}_{\text{ENC}} = \{((c, pk), (\alpha, r)) \mid c = \mathsf{Enc}_{pk}(\alpha; r)\}.$$

   The protocols are due to Schnorr [Sch91] (for ElGamal encryption) and Cramer et al. [CDN01] (for Paillier encryption).

2. Protocol π_{ZERO} demonstrates that a ciphertext c is an encryption of zero and is captured by the following language.

$$\mathrm{L}_{\text{ZERO}} = \{((c, pk), r) \mid c = \mathsf{Enc}_{pk}(0; r)\}.$$

For ElGamal this is merely π_{DH}, demonstrating that the key and ciphertext are a Diffie-Hellman tuple. For Paillier encryption this is a proof of Nth power shown by [DJ01].

3. The zero-knowledge proof of knowledge π_{MULT} proves that the plaintext of c_2 is the product of the two plaintexts encrypted by c_0, c_1. More formally,

$$\mathcal{R}_{\text{MULT}} = \{((c_0, c_1, c_2, pk), (\alpha, r_\alpha, r_0)) \mid c_1 = \mathsf{Enc}_{pk}(\alpha; r_\alpha) \wedge c_2 = c_0^\alpha \cdot \mathsf{Enc}_{pk}(0; r_0)\};$$

This proof is due to Damgård and Jurik [DJ01] (for both Paillier and ElGamal). A similar proof of knowledge is possible for commitment schemes, when the contents of *all three* commitments are known to the prover, [DN02, DN03]; this is required in π_{BOUND} below.

4. Protocol π_{BOUND} demonstrates boundedness of an encrypted value, i.e. that the plaintext is smaller than some public threshold B. Formally,

$$\mathrm{L}_{\text{BOUND}} = \{((c, pk, B), (\alpha, r)) \mid c = \mathsf{Enc}_{pk}(\alpha; r) \wedge \alpha < B \in \mathbb{N}\}.$$

The "classic" solution is to provide encryptions to the individual bits and prove in zero-knowledge that they are bits using the compound proof of Cramer et al. [CGS97]. The actual encryption is then constructed from these.

An alternative is to take a detour around integer commitments; this allows a solution requiring only $O(1)$ exponentiations [Bou00, Lip03, DJ02]. Sketching the solution, the core idea is to commit to α using a homomorphic integer commitment scheme; the typical suggestion is schemes such as [FO97, DF02]. The prover then demonstrates that α and $B - 1 - \alpha$ are non-negative (using the fact that any non-negative integer can be phrased as the sum of four squares), implying that $0 \leq \alpha < B$. Finally, the prover demonstrates that the committed value equals the encrypted value. For simplicity, we use the commitment scheme of Section 2.3. Note that for *small* B, the classic approach may be preferable in practice.

5. The proof π_{EQ} is of correct exponentiation in the group $\mathbb{G}$ with encrypted exponent (where the encryption scheme does *not* utilize the description of $\mathbb{G}$). Formally,

$$\mathcal{R}_{\text{EQ}} = \left\{\left((c, pk, \mathbb{G}, h, h'), (\alpha, r)\right) \mid \alpha \in \mathbb{N} \wedge c = \mathsf{Enc}_{pk}(\alpha; r) \wedge h, h' \in \mathbb{G} \wedge h' = h^\alpha\right\}.$$

The protocol is a simple cut and choose approach; its idea is originates from [CKY09]. Namely, the prover selects at random s, r_s and sends $c_s = \mathsf{Enc}_{pk}(s, r_s)$ and h^s to the verifier, who returns a random challenge bit b. The prover then replies with $\alpha \cdot b + s$ and $r^b \cdot r_s$, i.e. it sends either s or $s + m$. Privacy of α is ensured as long as the bit length of s is longer than the bit length of α by at least κ bits, where κ is a statistical parameter (fixing $\kappa = 100$ is typically sufficient to mask α). Note that it is only possible to answer both challenges if the statement is well-formed and thus the proof is also a proof of knowledge. Finally, we note that the protocol must be repeated in order to obtain negligible soundness. For a challenge of length 1 inducing soundness $1/2$, the number of repetitions must be $\omega(\log n)$ for n the security parameter. Reducing the number of repetitions is possible by increasing the length of the challenge. However, in this case we must ensure that $\alpha \cdot b + s$ is still private, namely, s is longer than the bit length of α plus the bit length of b, by at least κ. We point out that we can obtain constant amortized cost for multiple instances when the proof is instantiated with the technique of [CD09]. Indeed, we employ this proof multiple times when checking primality and potentially, for multiple decryptions. In this work, we instantiate Enc with the ElGamal PKE.

3.3 Public-Key and Ciphertext Relations

1. We include the folklore protocol π_{RSA} for proving that N and $\phi(N)$ are co-prime for some integer N, i.e. the protocols demonstrate membership of the language,

$$\mathrm{L}_{\text{RSA}} = \{(N, \phi(N)) \mid N \in \mathbb{N} \wedge \mathrm{GCD}\,(N, \phi(N)) = 1\}\,.$$

   The solution is to let the verifier pick $x = y^N$ and prove the knowledge of an Nth root (essentially execute the Paillier version of π_{ZERO}). The prover (who generated N) returns an Nth root y'. If $\mathrm{GCD}\,(N, \phi(N)) = 1$, then y is unique, hence $y = y'$. Otherwise there are multiple candidates, and the probability that $y = y'$ is $\leq 1/2$. This is repeated (in parallel) until a sufficiently low probability is reached. We note that this protocol is used only once, therefore its overhead does not dominate the analysis. Constructing a simulator is straightforward by simply extracting y first and then sending it to the verifier.

2. We also require a zero-knowledge proof, π_{MOD}, for proving consistency between two ciphertexts in the sense that one plaintext is the other one reduced modulo a fixed public prime. This proof is required for proving correctness within the trial division stage included in the key generation protocol (cf. Section 4). Formally,

$$\mathrm{L}_{\text{MOD}} = \left\{\left((c, c', p, pk), (\alpha, r, r')\right) \mid c = \mathsf{Enc}_{pk}(\alpha; r) \wedge c' = \mathsf{Enc}_{pk}(\alpha \bmod p; r')\right\}.$$

   Informally, the parties additionally compute $c'' = \left(c \cdot (c')^{-1}\right)^{p^{-1}}$, which is an encryption of $\alpha \operatorname{div} p$ (assuming that c' is correct). The prover then executes π_{BOUND} twice, on (c', p) and on $(c'', \lceil M/p \rceil)$, where M is an upper bound on the size of α. This demonstrates that α has been decomposed correctly, i.e. that the division has been performed correctly.

3. For public Paillier key N, we require Σ-protocol $\pi_{\text{EXP-RERAND}}$ that allows a prover to demonstrate that ciphertext c' is in the image of $\phi : \mathbb{Z}_N \times \mathbb{Z}^*_{N^2} \mapsto \mathbb{Z}^*_{N^2}$, defined by $\phi(m, r) = c^m \cdot r^N \bmod N^2$ for a fixed ciphertext $c \in \mathbb{Z}^*_{N^2}$. Namely,

$$\mathrm{L}_{\text{EXP-RERAND}} = \left\{\left(\left(N, c, c'\right), (m, r)\right) \mid c' = c^m \cdot r^N\right\}$$

   In order to demonstrate this, the prover picks v, r_v at random, and sends $A = \phi(v, r_v)$ to the verifier, who responds with a challenge $e < N$. The verifier then replies with $(z_1, z_2) = (me + v, r^e r_v)$, and the verifier checks that $\phi(z_1, z_2) = (c')^e \cdot A$.

   This is a Σ-protocol. To see this, note first that it is straightforward to verify completeness. Special soundness follows from the fact that for two accepting conversions $(A, e, (z_1, z_2))$ and $(A, e', (z_1', z_2'))$ with $e \neq e'$, we may find integers α, β such that $\alpha(e - e') + \beta N = 1$ using Euclid's algorithm (unless $(e - e', N)$ are not co-prime, which implies that we have found a factor of N). It is easily verified that $(\alpha(z_1 - z_1'); (z_2/z_2')^\alpha \cdot (c')^\beta)$ is a preimage of c' under ϕ. Finally, the simulator for the special honest-verifier zero-knowledge is straightforward: given a commitment, pick (z_1, z_2) at random and compute A.

3.3.1 A Zero-Knowledge Proof for π_{VERLIN}

In this section we give the details of ZK proof π_{VERLIN} used in Step 3a of Protocol 1 and in proving correctness in Protocol 3. Let N_P be a public Paillier key, with $g = N_P + 1$ generating the plaintext group. π_{VERLIN} is a Σ-protocol allowing a prover P to demonstrate to a verifier V that a Paillier ciphertext, c_x

has been computed based on two other ciphertexts c and c' as well as a known value, i.e. that P knows a preimage of c_x with respect to

$$\phi_{(c,c')}\left(x, x', x'', r_x\right) = c^x \cdot c'^{x'} \cdot \mathsf{Enc}_{pk}\left(x'', r_x\right).$$

This is done by first picking a, a', a'' uniformly at random from $\mathbb{Z}_{N_P}$ and r_a uniformly at random from $\mathbb{Z}^*_{N_P}$, and sending

$$c_a = \phi_{(c,c')}\left(a, a', a'', r_a\right)$$

to the verifier, V. V then picks a uniformly random t-bit challenge,[1] e, and sends this to P, who replies with the tuple

$$\left(z, z', z'', r_z\right) = \left(xe + a, x'e + a', x''e + a'', r_x^e r_a\right).$$

V accepts if and only if $\phi_{(c,c')}\left(z, z', z'', r_z\right) = c_x^e \cdot c_a$.

Proposition 3.1 *Assuming hardness of the DCR problem, π_{VERLIN} is a Σ-protocol with constants costs.*

Proof: We prove that all three properties required for Σ-protocols (cf. Definition 2.6) are met.

Completeness. An honest prover always convinces the verifier, since

$$\begin{aligned}
\phi_{(c,c')}\left(z, z', z'', r_z\right) &= \phi_{(c,c')}\left(xe + a, x'e + a', x''e + a'', r_x^e r_a\right) \\
&= c^{xe+a} \cdot c'^{x'e+a'} \cdot g^{x''e+a''} \cdot \left(r_x^e r_a\right)^{N_P} \\
&= \left(c^x \cdot c'^{x'} \cdot g^{x''} \cdot r_x^{N_P}\right)^e \cdot \left(c^a \cdot c'^{a'} \cdot g^{a''} \cdot r_a^{N_P}\right) \\
&= c_x^e \cdot c_a
\end{aligned}$$

Special soundness. A preimage of c_x may be computed given two accepting conversations with same initial message, c_a, and differing challenges $e \neq \bar{e}$. Denote the final messages of the two executions (z, z', z'', r_z) and $(\bar{z}, \bar{z}', \bar{z}'', r_{\bar{z}})$, and compute integers α and β such that

$$\alpha\left(e - \bar{e}\right) + \beta \cdot N_P = 1$$

using the extended Euclidian algorithm. This is always possible as $e - \bar{e}$ and N_P are co-prime. It is straightforward but tedious to verify that

$$\left(\alpha\left(z - \bar{z}\right), \alpha\left(z' - \bar{z}'\right), \alpha\left(z'' - \bar{z}''\right), \left(r_z \cdot r_{\bar{z}}^{-1}\right)^\alpha \cdot \left(c_x \bmod N_P\right)^\beta\right)$$

is a preimage of c_x, briefly

$$\begin{aligned}
&\phi_{(c,c')}\left(\alpha\left(z - \bar{z}\right), \alpha\left(z' - \bar{z}'\right), \alpha\left(z'' - \bar{z}''\right), \left(r_z \cdot r_{\bar{z}}^{-1}\right)^\alpha \cdot \left(c_x \bmod N_P\right)^\beta\right) \\
&\quad = \left(c_x^e \cdot c_a\right)^\alpha / \left(c_x^{\bar{e}} \cdot c_a\right)^\alpha \cdot \left(\left(c_x \bmod N_P\right)^\beta\right)^{N_P} \\
&\quad = c_x^{\alpha(e-\bar{e})} \cdot c_x^{\beta \cdot N_P} \\
&\quad = c_x.
\end{aligned}$$

[1] Choose t such that 2^t is smaller than any prime factor of N_P. This guarantees that the difference between challenges is co-prime to N_P.

Special Honest-Verifier Zero-knowledge. Accepting transcripts are easily simulated. Given challenge e, pick z, z', z'' uniformly at random from $\mathbb{Z}_{N_P}$ and r_z uniformly at random from $\mathbb{Z}^*_{N_P}$. Then, compute

$$c_a = \phi_{(c,c')}\left(z, z', z'', r_z\right) \cdot c_x^{-e}.$$

Clearly, this is an accepting conversation. Moreover, for preimage (x, x', x'', r_x) (i.e., the witness),

$$\left(z - ex, z' - ex', z'' - ex'', r_z \cdot r_x^{-e}\right)$$

is uniformly random, and a preimage of c_a, i.e. it corresponds to the choice of (a, a', a'', r_a); hence c_a is distributed as in the real protocol. Finally, given this random choice and a witness, (z, z', z'', r_z) is exactly the final message that an honest prover would send. ■

4 A Distributed Generation of an RSA Composite

This section presents a protocol, denoted DKeyGen, for distributively generating an RSA composite without disclosing any information about its factorization and with security against malicious activities. In this protocol the parties generate candidates for the potential composite which they run through a biprimality test for checking its validity. Our protocol is useful for designing distributive variants of the RSA encryption and signature schemes, as well as other schemes that rely on factoring related hardness assumptions. In this paper we use this protocol for distributively generating the public-key for Paillier [Pai99] encryption scheme. The starting point for DKeyGen is the protocols of [BF01, Gil99]. These protocols are designed to distributively generate an RSA composite $N = p \cdot q$ with an unknown factorization. Specifically, the protocol by Boneh and Franklin [BF01] assumes honest majority, whereas the protocol by Gilboa [Gil99] adopts ideas from [BF01] into the two-party setting; both are secure in the semi-honest setting.

Recall that when coping with malicious adversaries it must be assured that the parties follow the protocol specifications. In our context this means that the parties' shares must be of the appropriate length and that no party gains any information about the factorization of N, even by deviating.[2] This challenging task is typically addressed by adding commitments and zero-knowledge proofs to each step of the protocol. Unfortunately, this is usually not very practical since the statements that needed to be proven are complicated, and therefore leading to highly inefficient protocols. Instead, we will be exploiting specific protocols for our tasks (some new to this work), that are both efficient and fully secured. By proper analysis of where to use the zero knowledge proofs, which proofs to use and moreover, by a novel technique of utilizing two different encryption schemes with different homomorphic properties, we achieve a highly efficient key generation protocol. It should be noted that except for the setup step which is only executed once we can avoid expensive zero knowledge proofs based on the cut and choose technique. Additional optimizations can be found in Section 6.

For the sake of completeness we include a short description of [BF01] as adapted by [Gil99] for the two-party setting. These protocols consist of the following three steps: (**1**) Each party P_i generates two random numbers p_i and q_i representing shares of p and q, such that $p = \sum_i p_i$, $q = \sum_i q_i$ and $p \equiv q \equiv 3 \pmod 4$. We note that [BF01] includes a distributed trial division of p and q for primes less than a bound B, which greatly improves the efficiency of the protocol. This trial division is not adopted by [Gil99], making our solution the first two party protocol achieving the significant speedup from this trial division. (**2**) After

[2] In some settings, early abort may not be considered as a breach of security (even if the abort occurs as a result of gaining information about the factorization of the public-key). This is due to the fact that the honest party halts as well, outputting $\perp$. Thus, essentially, no damage was caused. However, this is not true for applications where the shares are chosen based on the honest party's secret state. Realizing this functionality requires to incorporate into it a secret state of the users. Unfortunately, our protocol cannot compute this functionality in a secure manner, as it must be that the composite generation and the biprimality test run together. Meaning, the parties only learn the composite if it is accepted by the biprimality test.

having created the two candidates for being primes the parties execute a secure multiplication protocol to compute $N = (p_0 + p_1)(q_0 + q_1)$. In [BF01] this step is based on standard generic solutions. Here we take a novel approach of utilizing both ElGamal and Paillier encryption schemes, giving us active security at a very low cost. Generating the RSA composite this way does not guarantee that the composite is made of uniformly random primes since the adversary can, in some limited way, influence the distribution of the primes. This issue was observed in [BF01] and discussed further below. (**3**) Finally, the candidate N for being an RSA composite is tested by a distributed biprimality test, which requires $p \equiv q \equiv 3 \pmod 4$. If the biprimality test rejects N as being a proper RSA modulus, the protocol is restarted.

Typically, the definition for the key generation algorithm requires that the RSA composite would be a product of two randomly chosen equal length primes p and q. However, in order to use the distributed biprimality test of Boneh and Franklin N must be a Blum integer (namely, $N = pq$, where $p \equiv q \equiv 3 \mod 4$). Note that this is a common requirement for distributed biprimality tests and it does not decrease the security of the constructions that use these type of keys, since about $1/4$ of all the RSA modulus are Blum integers. Moreover, as pointed out by [BF01], the parties can always influence the distribution of the most significant bit of each prime. This is because p and q are generated by adding shares over the integers which implies that they are not uniformly random, so that each party has some (limited) knowledge of the distribution based on its shares.

We will therefore define a new public-key generation algorithm, Gen', which captures this deviation and generates N by the same distribution as the protocol. This is obtained by Gen' receiving additional two inputs r_p and r_q, representing potential adversary's input shares. Gen' adds this shares to some truly random chosen shares and ensures that the sum is congruent to $3 \mod 4$. Formally, let $\mathsf{Gen}'(1^n, r_p, r_q)$ denote a public-key generation that takes two additional inputs besides the security parameter 1^n and works as follows:

1. If $r_p, r_q \geq 2^{n-2}$ output $\perp$ and halt.
2. Otherwise, choose a uniform random $s_p \in \{0,1\}^{n-2}$.
3. Calculate $p = 4(r_p + s_p) + 3$ and examine the outcome:
   - if p is composite, then goto Step 2 and choose a new value for s_p.
   - if p is prime, then repeat the process to generate q.
4. Return $N = pq$, and generate the private key as in Gen.

As proven by Boneh and Franklin, using Gen' instead of Gen does not give the adversary the ability to factor N even if it can slightly influence its distribution. We formally state the security statement proven by [BF01],

Lemma 4.1 (Revised [BF01, Lemma 2.1]) *Suppose there exists a* PPT *algorithm $\mathcal{B}$ that chooses values (r_p, r_q) as above, then given $N \leftarrow \mathsf{Gen}'(1^n, r_p, r_q)$ and finally, factors N with probability at least $1/n^d$. Then there exists an expected polynomial time algorithm $\mathcal{B}'$ that factors a random RSA modulus with n-bit factors with probability at least $1/(2^7 n^d)$.*

This lemma was originally stated for semi-honest adversaries but holds for the malicious case as well. We note that protocols with p and q being chosen as uniform random values do exist, however, they are significantly less efficient. Therefore, we choose to accept this nonuniform distribution induced by protocol $\mathsf{DKeyGen}$ and let the ideal functionality capture this deviation. Functionality $\mathcal{F}_{\mathrm{GEN}}$ formalizes this discussion in Figure 1.

We are now ready to describe our protocol which in addition to the above three steps includes a *key-setup* step used to generate keys for commitments and encryption schemes used in the protocol. Namely, a

Functionality $\mathcal{F}_{\text{GEN}}$

Key Generation: Upon receiving from party P_i a $(\mathsf{Generate}, 1^n)$ message, functionality $\mathcal{F}_{\text{THRES}}$ sends a message $(\mathsf{RandInput})$ to the adversary and waits for the adversary to reply with $(\mathsf{GenInput}, r_a, r_b)$. $\mathcal{F}_{\text{THRES}}$ then invokes $(pk, sk) \leftarrow \mathsf{Gen}'(1^n, r_a, r_b)$, records sk and sends pk to the adversary. If the adversary replies allow, the functionality sends pk to the parties, ignoring further messages of this form. Otherwise, it sends $\perp$ to the honest party.

Figure 1: The RSA Modulus Generation Functionality

shared key is generated for the distributed additively homomorphic ElGamal encryption scheme and each party generates a key for standard non-distributive Paillier encryption and integer commitments. We observe that the reason for using both ElGamal and Paillier is due to efficiency considerations. Namely, most zero-knowledge proofs used here can be implemented in an efficient manner when applied on ElGamal (with a known group order), rather than on Paillier. Nevertheless, the plaintext cannot be recovered efficiently and therefore we use Paillier in a non-distributive fashion. We note that consistency between the encryptions using Paillier and ElGamal is only proven implicitly, by repeating the computations and verifying that the two executions give the same result. In addition, the fact that a distributive ElGamal variant can be easily obtained allows us to design a the trial division test that is run on individual primes and improves the numbers of trials. In order to cope with malicious adversaries our protocols employ zero-knowledge (ZK) proofs, some are known, others are new to this work and are interesting by themselves. We note that all the proofs that participate in Protocol 1 require a strict constant overhead. In Section 3 we specify these proofs in detail.

Protocol 1 [DKeyGen] *A distributed generation of an RSA composite with malicious security:*

- **Inputs for parties** P_0, P_1**:** *A security parameter* 1^n *and a threshold* B *for the trial division.*

1. **Key-Setup.**

   (a) *The parties run protocol* π_{GEN} *(cf. Section 2.2.2) for generating a public-key* $pk_{\text{EG}} = (g, h)$ *and secret key shares* sk^0_{EG} *and* sk^1_{EG} *for ElGamal.*

   (b) *Each party* P_i *generates a Paillier key pair* $(pk^i_{\text{Pa}}, sk^i_{\text{Pa}})$ *with a modulus bit length of* $\lambda > 2n$*, and sends* $pk^i_{\text{Pa}} = N^i_{\text{Pa}}$ *to the other party. Each party proves correctness of* N^i_{Pa} *by* π_{RSA} *(cf. Section 3). The Paillier keys are used for encryptions and commitments (cf. Sections 2.2.1, 2.3)*

2. **Generate Candidates.**

   (a) **Generate Shares of Candidate.** *Each party* P_i *picks a random* $(n-2)$*-bit value* $\bar{p}_i$*, encrypts it and sends* $\bar{c}_i = \mathsf{Enc}_{pk_{\text{EG}}}(\bar{p}_i)$ *to the other party. The parties prove knowledge of the plaintexts, via* π_{ENC}*, and prove that* $\bar{p}_i < 2^{n-2}$ *via* π_{BOUND}.
   In order to ensure that $p_0 \equiv 3 \bmod 4$*, the parties compute* $c_0 \leftarrow (\bar{c}_0)^4 \cdot \mathsf{Enc}_{pk_{\text{EG}}}(3)$*. Similarly, the parties ensure that* $p_1 \equiv 0 \bmod 4$ *by* $c_1 \leftarrow (\bar{c}_1)^4$.

   (b) **Trial division.** *For all primes* $\alpha \leq B$*, the parties run a trial division on* $p = p_0 + p_1$*. Each party* P_i *sends an encryption* $c^{(\alpha)}_i = \mathsf{Enc}_{pk_{\text{EG}}}(p_i \bmod \alpha)$ *to the other party, and proves the correctness of the computation using* π_{MOD}.
   The parties compute $c^{(\alpha)} \leftarrow c^{(\alpha)}_0 \cdot c^{(\alpha)}_1$ *and* $\tilde{c}^{(\alpha)} \leftarrow c^{(\alpha)} \cdot \mathsf{Enc}_{pk_{\text{EG}}}(-\alpha)$*. Clearly* α *divides* p *if and only if* $p^{(\alpha)}_0 + p^{(\alpha)}_1 \in \{0, \alpha\}$*, i.e. when either* $c^{(\alpha)}$ *or* $\tilde{c}^{(\alpha)}$ *is an encryption of zero. This is checked by raising these to secret, non-zero exponents and decrypting. If no prime* $\alpha < B$ *divides the candidate it is accepted by trial division.*

   (c) **Repeat.** *Repeat Steps 2a- 2b until two candidates* p *and* q *survive trial division.*

3. **Compute Product ($N = pq$).**

   (a) **Compute the product.** *P_0 sends P_1 encryptions of $\tilde{p}_0 = p_0$ and $\tilde{q}_0 = q_0$ under pk^0_{Pa} and proves knowledge of plaintexts using π_{ENC}. (Note that a malicious P_0 may send encryptions of incorrect values). Next, P_1 computes and sends:*

$$c_{\tilde{N}-\tilde{p}_0\tilde{q}_0} \leftarrow \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(p_0)^{q_1} \cdot \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(q_0)^{p_1} \cdot \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(p_1 q_1) = \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}((p_0+p_1)(q_0+q_1) - p_0 q_0)$$

   Furthermore, P_1 proves that $c_{\tilde{N}-\tilde{p}_0\tilde{q}_0}$ has been computed as a known linear combination based on $\mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(\tilde{p}_0)$ and $\mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(\tilde{q}_0)$ using π_{VERLIN}. P_0 decrypts, thus obtaining the plaintext $m_{\tilde{N}-\tilde{p}_0\tilde{q}_0}$; from this $\tilde{N} = m_{\tilde{N}-\tilde{p}_0\tilde{q}_0} + \tilde{p}_0\tilde{q}_0$ is computed and sent to P_1 along with an encryption $c_\pi = \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(\tilde{p}_0\tilde{q}_0)$. Finally, using π_{MULT} and π_{ZERO}, P_0 proves that c_π contains the product of the two original ciphertexts and that $\tilde{N}$ is the plaintext of

$$c_{\tilde{N}-\tilde{p}_0\tilde{q}_0} c_\pi = \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}((\tilde{p}_0+p_1)(\tilde{q}_0+q_1)).$$

   (b) **Verify Multiplication.** *The parties use the homomorphic property of ElGamal encryption to compute an encryption of $N = (p_0+p_1)(q_0+q_1)$ from the ciphertexts generated at Step 2a. The computation is analogous to that of Step 3a, again using π_{MULT} for proving correct multiplication of $(p_i \cdot q_i)$*

   The parties use secure decryption of distributed ElGamal π_{DEC} (cf. Section 2.2.2) to obtain g^N, where both verify that $g^{\tilde{N}} = g^N$, i.e. that $N = \tilde{N}$ and abort if equality does not hold.

4. **Biprimality Test.**

   Execute biprimality test (cf. Section 4.2) and accept N if the test has accepted, otherwise the protocol is restarted from Step 2a.

Theorem 4.1 *Assuming hardness of the DDH and DCR problems, Protocol 1 realizes $\mathcal{F}_{\mathrm{GEN}}$ in the presence of malicious adversaries.*

A proof overview. Note that if both parties follow the protocol then a valid RSA modulus N is generated with high probability. Specifically, in the last iteration of the protocol two elements are chosen randomly and independently of previous generated candidates and are multiplied to produce N. By the correctness of the biprimality test specified below, N is a product of two primes with overwhelming probability.

We assume the simulator has knowledge of the distribution of the loops in the protocol, from the protocol returning to step 2a when candidates are rejected. The simulator can simulate the distribution by running the protocol "in its head", emulating the role of the honest party. Namely, denoting by P_i the corrupted party, then upon extracting $\mathcal{A}$'s shares p_i, q_i, $\mathcal{S}$ picks two shares p_{1-i}, q_{1-i} as the honest party would do and checks whether $N_{\mathcal{S}} = (p_i + p_{1-i})(q_i + q_{1-i})$ constitutes a valid RSA composite. If this is not the final iteration of the protocol, implied by the fact that $N_{\mathcal{S}}$ is not a valid RSA composite, $\mathcal{S}$ uses p_{1-i}, q_{1-i} to perfectly emulate the role of the honest P_{1-i}. If this is the final iteration, $\mathcal{S}$ asks the trusted party for $\mathcal{F}_{\mathrm{GEN}}$ to generate an RSA composite with p_i, q_i being the adversary's input (as specified in Figure 1) and completes the execution by emulating the role of the honest party on arbitrary shares. The simulation is different for the two corruption cases as the parties' roles are not symmetric. For the case that P_0 is corrupted, the simulator sends back in Step 3a the encryption of the composite returned from the trusted party and makes the ElGamal decryption decrypted into this composite as well. For the case that P_1 is corrupted the simulator "decrypts" the Paillier ciphertext result into that composite and then makes the ElGamal decryption return the same outcome.

In Step 3a, where the parties compute the product, we note that it is insufficient to let P_1 complete the computation over the encrypted shares of P_0 without verification of correctness. The problem is that P_1 may attempt to compute N in a different, potentially failing way. Hence if it *finds* N, this may leak information. Although this issue does not seem critical for practical considerations we have to deal with it in order to obtain simulation based security. This makes this corruption case particular challenge since we

had to show that an alternative computation in a *successful* execution implies determining the factors before the RSA-modulus is revealed.

The complete proof follows.

Proof: The proof is shown in a hybrid model, where a trusted party replaces the protocols π_{GEN}, π_{ENC}, π_{MOD}, π_{VERLIN}, π_{MULT}, π_{ZERO}, π_{DEC} and DPrim. We assume the simulator $\mathcal{S}$ has knowledge of the distribution of the loops in the protocol. The simulator can then simulate the distribution by running the protocol "in its head", emulating the role of the honest party. Namely, denoting by P_i the corrupted party, then upon extracting $\mathcal{A}$'s shares p_i, q_i, $\mathcal{S}$ picks two shares p_{1-i}, q_{1-i} as the honest party would do and checks whether $N_{\mathcal{S}} = (p_i + p_{1-i})(q_i + q_{1-i})$ constitutes a valid RSA composite. If this is not the final iteration of the protocol, implied by the fact that $N_{\mathcal{S}}$ is not a valid RSA composite, $\mathcal{S}$ uses p_{1-i}, q_{1-i} to perfectly emulate the role of the honest P_{1-i}. If this is the final iteration, $\mathcal{S}$ asks the trusted party for $\mathcal{F}_{\text{GEN}}$ to generate an RSA composite with p_i, q_i being the adversary's input (as specified in Figure 1) and completes the execution as follows. We distinguish between three different corruption cases.

No party is corrupted. In this case the adversary only sees the communication exchanged between the two parties. Specifically, we claim that such an attacker that only eavesdrops the communication cannot learn meaningful information about the factorization of N. This is because even an adversary that fully corrupts one of the parties cannot obtain such information. Let alone an adversary that does not control the secret shares of one of the parties.

P_0 is corrupted. Let $\mathcal{A}$ denote an adversary controlling party P_0. Then, construct a simulator $\mathcal{S}$ simulating the view of the adversary as follows.

1. KEY-SETUP.
   (a) $\mathcal{S}$ emulates the trusted party $\mathcal{F}_{\text{GEN}}$ by generating an ElGamal key pair $(pk_{\text{EG}}, sk_{\text{EG}})$, sharing sk_{EG} to $sk_{\text{EG}}^{\mathcal{S}}$ and $sk_{\text{EG}}^{\mathcal{A}}$ and handing pk_{EG} and $sk_{\text{EG}}^{\mathcal{A}}$ to $\mathcal{A}$.
   (b) $\mathcal{S}$ generates a Paillier key pair $(pk_{\text{Pa}}^{\mathcal{S}}, sk_{\text{Pa}}^{\mathcal{S}})$ as described, sends pk_{Pa}^{S} to $\mathcal{A}$, and receives a key $pk_{\text{Pa}}^{\mathcal{A}}$ from $\mathcal{A}$. π_{RSA} is executed to verify that the Paillier keys are well formed.
2. GENERATE CANDIDATES.
   (a) GENERATE SHARES OF CANDIDATE. $\mathcal{S}$ uses 0 for $\tilde{p}_{\mathcal{S}}$, encrypts it and sends $\tilde{c}_{\mathcal{S}} = \mathsf{Enc}_{pk_{\text{EG}}}(0)$ to $\mathcal{A}$, and receives $\tilde{c}_{\mathcal{A}}$ from $\mathcal{A}$. $\mathcal{S}$ emulates the trusted party for $\mathcal{F}_{\text{ENC}}$, receiving $p_{\mathcal{A}}$ and the randomness used as witness and verifying $\tilde{c}_{\mathcal{A}}$. If the verification fails $\mathcal{S}$ halts. $\mathcal{S}$ stores the share $p_{\mathcal{A}}$. π_{BOUND} is executed where $\mathcal{S}$ once plays the role of the verifier and once the role of the prover (note that 0 meets the bound requirement made in Step 2a of the protocol).
   In order to ensure that $p_0 \equiv 3 \bmod 4$, the parties compute $c_0 \leftarrow (\tilde{c}_0)^4 \cdot \mathsf{Enc}_{pk_{\text{EG}}}(3)$. Similarly, the parties ensure that $p_1 \equiv 0 \bmod 4$ by $c_1 \leftarrow (\tilde{c}_1)^4$.
   (b) TRIAL DIVISION. For primes $\alpha \leq B$, $\mathcal{S}$ sends an encryption $c_{\mathcal{S}}^{(\alpha)} = \mathsf{Enc}_{pk_{\text{EG}}}(0)$ to $\mathcal{A}$, and receives $c_{\mathcal{A}}^{(\alpha)}$ from $\mathcal{A}$. $\mathcal{S}$ emulates the trusted party for $\mathcal{F}_{\text{MOD}}$, receiving the randomness used for encrypting $c_{\mathcal{A}}^{(\alpha)}$ as witness and verifying its correctness.
   $\mathcal{S}$ and $\mathcal{A}$ each compute $c^{(\alpha)} \leftarrow c_{\mathcal{A}}^{(\alpha)} \cdot c_{\mathcal{S}}^{(\alpha)}$ and $\tilde{c}^{(\alpha)} \leftarrow c^{(\alpha)} \cdot \mathsf{Enc}_{pk_{\text{EG}}}(-\alpha)$, and raising these to secret, non-zero exponents.
   $\mathcal{S}$ simulates decrypting, by emulating the output of $\mathcal{F}_{\text{DEC}}$ to be either zero or a random nonzero element in the plaintext space. A nonzero element is output when trial division by α should

succeed, and zero is the output when trial division should fail. This is simulated according to the distribution of the execution.

(c) REPEAT. Repeat step 2a and 2b according to the distribution of execution.

3. COMPUTE PRODUCT ($N = pq$).

(a) COMPUTE THE PRODUCT. $\mathcal{S}$ sends p_0 and q_0 to $\mathcal{F}_{\text{THRES}}$ and receives an RSA composite N from $\mathcal{F}_{\text{THRES}}$. If $\mathcal{A}$ does not deviate and sends proper encryptions of p_0, q_0, $\mathcal{S}$ simulates $c_{\tilde{N}-\tilde{p}_0\tilde{q}_0} \leftarrow \mathsf{Enc}_{pk^0_{\text{Pa}}}(N - p_0q_0)$ and emulates $\mathcal{F}_{\text{VERLIN}}$ as accepting. $\mathcal{S}$ then receives $\tilde{N}$ and c_π from $\mathcal{A}$ and emulates $\mathcal{F}_{\text{MULT}}$ and $\mathcal{F}_{\text{ZERO}}$ by receiving the witnesses and verifying the statements. If the conditions for accepting are not met, $\mathcal{S}$ halts, aborting the execution.

If $\mathcal{A}$ does not send proper encryptions of p_0, q_0, $\mathcal{S}$ uses p_1, q_1, picked at the outset of this iteration, for completing the execution. (Note that $N_{\mathcal{S}} = (p_0 + p_1)(q_0 + q_1)$ form a valid RSA composite. Looking a head, this would imply that the decryption of $c_{\tilde{N}-\tilde{p}_0\tilde{q}_0}$ is identically distributed in both the simulated and hybrid executions since in both cases the adversary sees some information of shares picked as the honest P_1 would).

(b) VERIFY MULTIPLICATION. $\mathcal{S}$ simulates the computation of the encrypted N by running the protocol as specified and emulating $\mathcal{F}_{\text{MULT}}$ twice; once by receiving the witness and verifying the statement, and once by emulating an accepting answer for verifying the honest P_1's computations.

If $\mathcal{A}$ did not deviate in step 3a, then $\mathcal{S}$ emulates ideal execution for $\mathcal{F}_{\text{DEC}}$ as outputting $g^N = g^{\tilde{N}}$ where N is the composite returned by $\mathcal{F}_{\text{GEN}}$. If $\mathcal{A}$ deviated in step 3a, then $\mathcal{S}$ emulates $\mathcal{F}_{\text{DEC}}$ as outputting $g^{N_{\mathcal{S}}} \neq g^{\tilde{N}}$, and aborting the protocol.

4. BIPRIMALITY TEST.

Simulate DPrim as either accepting or rejecting according to the distribution of the protocol run, which can be done by Theorem 4.2.

Clearly $\mathcal{S}$ runs in expected polynomial time. It is left to prove indistinguishability of the simulation in the ideal world and the hybrid execution of the protocol. This will be done by a series of games.

Game H_0**.** This game corresponds to the original simulation.

Game H_1**.** In this game there is no functionality $\mathcal{F}_{\text{THRES}}$. Instead, simulator $\mathcal{S}_1$ generates the RSA modulus N by itself the same way $\mathcal{F}_{\text{THRES}}$ does it. This means $\mathcal{S}_1$ knows the factorization of N. Furthermore, $\mathcal{S}_1$ plays the exact same role as $\mathcal{S}$. H_1 is clearly perfectly indistinguishable from H_0.

Game H_2**.** Simulator $\mathcal{S}_2$ does not know the secret ElGamal key. That is, instead of emulating $\mathcal{F}_{\text{GEN}}$ in step 1a, $\mathcal{S}_2$ receives pk_{EG} from an oracle, generates a random share as $sk^{\mathcal{A}}_{\text{EG}}$ and hands pk_{EG} and $sk^{\mathcal{A}}_{\text{EG}}$ to $\mathcal{A}$. Since $\mathcal{S}_1$ does not use its knowledge of sk_{EG}, and since $sk^{\mathcal{A}}_{\text{EG}}$ can be simulated statistically close to the real value. H_2 is statistically indistinguishable from H_1.

Game H_3**.** In this game simulator $\mathcal{S}_3$ uses N instead of $N_{\mathcal{S}}$ in case $\mathcal{A}$ deviates in Step 3a of the simulation above. Then, the only difference between the views of games H_2 and H_3 is with respect to this step. However, since the rest of the messages computed by $\mathcal{S}_2$ and $\mathcal{S}_3$ are independent of these composites. The adversary's view is identically distributed in both games.

Game H_4**.** In this game simulator $\mathcal{S}_4$ does not simulate the ElGamal ciphertexts by sending encryptions of 0, rather it sends encryptions of the real shares that yield the right RSA composite N. More specifically, since $\mathcal{S}_4$ extracts the shares p_0, q_0 of $\mathcal{A}$ and since $\mathcal{S}_4$ generates N by itself it can compute shares p_1, q_1 such that $N = (p_0 + p_1)(q_0 + q_1)$, and simulate the execution resulting in an accepted N.

H_4 is computationally indistinguishable from H_3 which can be proven by a reduction to the sematic security of ElGamal (according to IND-CPA game specified in Definition 2.4). More formally, the reduction goes as follows: We assume the existence of an distinguisher D_{3-4} capable of distinguishing H_3 from H_4 with more than negligible probability. We now construct an adversary $\mathcal{A}_{\mathrm{EG}}$ for breaking the security of the ElGamal encryption scheme. $\mathcal{A}_{\mathrm{EG}}$ follows $\mathcal{S}_4$'s instructions, except instead of encrypting values with the pk_{EG} it asks an oracle to encrypt either $m_0 = 0$ or m_1 being the real share that yields a proper prime. $\mathcal{A}_{\mathrm{EG}}$ completes the execution of D_{3-4} as in game H_4. If D_{3-4} guesses H_3 as being executed, then $\mathcal{A}_{\mathrm{EG}}$ outputs 0, if D_{3-4} guesses H_3 as being executed, then $\mathcal{A}_{\mathrm{EG}}$ outputs 1.

Game H_5**.** In this game simulator $\mathcal{S}_5$ does not extract the shares of the candidates for being primes from $\mathcal{A}$. Instead, $\mathcal{S}_5$ generates the shares as the honest P_1 does in the hybrid execution. This implies that the only difference in $\mathcal{A}$'s view within executions H_4 and H_5 is with respect to step 3b of the simulation above, where $\mathcal{S}_4$ makes the protocol abort if $\mathcal{A}$ deviates in step 3a by not sending proper encryptions of p_0, q_0. Note that there are two cases here: (**1**) $\mathcal{A}$ deviates by sending encryptions of values different than p_0 and q_0, which leads to a different composite than the one computed using the ElGamal encryptions in Step 2a of the simulation. (**2**) $\mathcal{A}$ deviates by sending encryptions of values different than p_0 and q_0, which leads to the same composite than the one computed using the ElGamal encryptions in Step 2a.

Focusing on (1), we note that the adversary always sees the same view in both games, since in both cases it sees some information of the real shares picked by P_1 that correspond to N, and get caught. Since there is no match between the two composites. As for (2), we note that as proven in appendix 4.1, honest P_1 always detects such a cheating, implying that H_4 is statistically close to H_5.

Finally, since H_5 corresponds to executing the real protocol in the hybrid model, we conclude that if P_0 is corrupted then the real and the ideal executions are computationally indistinguishable.

P_1 **is corrupted.** Let $\mathcal{A}$ denote an adversary controlling party P_1. Then, construct a simulator $\mathcal{S}$ as follows. All steps except step 3a are analog to the previous simulation.

3. COMPUTE PRODUCT. ($N = pq$)

   (a) COMPUTE THE PRODUCT. $\mathcal{S}$ simulates the encryptions of $\tilde{p}_0$ and $\tilde{q}_0$ as encryptions of 0 under $pk^{\mathcal{S}}_{\mathrm{Pa}}$ and emulates $\mathcal{F}_{\mathrm{ENC}}$, as accepting.
   $\mathcal{S}$ receives $c_{\tilde{N}-\tilde{p}_0\tilde{q}_0}$ from $\mathcal{A}$ and during emulation of $\mathcal{F}_{\mathrm{VERLIN}}$, $\mathcal{S}$ extracts the values p_1 q_1 and p_1q_1, and if these values are not consistent with the values extracted in step 2a, then $\mathcal{S}$ continues simulating the protocol using $N_{\mathcal{S}}$ and p_0, q_0.
   If $\mathcal{A}$ does not deviate, $\mathcal{S}$ sends encryption of $\tilde{N} = N$ to $\mathcal{A}$ along with an encryption $c_\pi = \mathsf{Enc}_{pk^0_{\mathrm{Pa}}}(0)$ and emulates $\mathcal{F}_{\mathrm{MULT}}$ and $\mathcal{F}_{\mathrm{ZERO}}$ as accepting.

Clearly $\mathcal{S}$ runs in expected polynomial time, it is left to prove indistinguishability of the simulation in the ideal world and the real execution of the protocol. This will be done by a series of games.

Game H_0 **-** H_4**.** These games are identical to games H_0 - H_4 in the case of the corruption of P_0.

Game H_5**.** In this game simulator $\mathcal{S}_5$ does not simulate the Paillier encryptions by encryptions of 0. Instead, it sends the encryption of p_0q_0. We claim that H_4 and H_5 are computationally indistinguishable. The proof is completely analog to the indistinguishability of H_3 and H_4.

Game H_6**.** In this game simulator $\mathcal{S}_6$ does not extract the shares of the prime candidates from $\mathcal{A}$ in steps 2a and 3a. Instead, $\mathcal{S}_6$ generates the shares as the honest P_0 does in the hybrid execution. Recall that in the previous game $\mathcal{S}_5$ makes the protocol abort in step 3b if $\mathcal{A}$ deviates in step 3a, however, as described in appendix 4.1 this will also be the case in this game. Therefore, H_5 and H_6 are statistically close.

Since H_6 corresponds to executing the real protocol in the hybrid model, we conclude that if P_0 is corrupted then the real and the ideal executions are computationally indistinguishable. ■

4.1 Deviation while Computing $\tilde{N}$ is Always Detected

The ZK proofs used in the calculation of $\tilde{N}$ in Step 3a in Protocol 1 ensure that the parties compute $\tilde{N}$ based on values they know, but not that the correct p_i and q_i are used for this computation. More specifically, this step ensures that $\tilde{N} \equiv N \bmod N_0$, where N_0 is the Paillier key picked by P_0. However, this does not rule out from a malicious party, $\mathcal{A}$, supplying wrong values but still getting the right result. For instance, $\mathcal{A}$ can guess the difference δ between p and q and switch the factors around by using $p_i + \delta$ and $q_i - \delta$ instead of p_i and q_i, respectively. Now, since we have that

$$p_0 + p_1 + \delta = q, \quad q_0 + q_1 - \delta = p$$

the product of $p_i + \delta$ and $q_i - \delta$ still equals N. Intuitively, this specific attack is infeasible, however, we must rule out *all* such attacks. A much worse attack would be one where the attacker could induce failure, say depending on some specific bit of N (which is of course unknown to $\mathcal{A}$ at the time). In a real execution of the protocol, since N is the output, $\mathcal{A}$ simply learns the bit ahead of time, which may not be critical in practice. However, in the simulated execution, $\mathcal{A}$ learns a bit of the simulated modulus, N, which may differ from the analogous bit of the modulus supplied by the ideal functionality, $\mathcal{F}_{\mathrm{GEN}}$. Hence $\mathcal{A}$ may be able to distinguish between the executions.

In this section we show that no such behavior is possible. Specifically, we show that if an execution with a malicious $\mathcal{A}$ results in $\tilde{N} = N$, then this must have been computed using the correct shares of p and q. We do this by using $\mathcal{A}$ to break the IND-CPA security of either ElGamal or Paillier encryption (cf. Definition 2.4). We distinct the cases of corrupted P_0 and P_1.

P_0 is corrupted. Let $\mathcal{A}$ denote an adversary controlling party P_0 and denote by bad the event in which a malicious P_0 sends in Step 3a of Protocol 1 encryptions of $\tilde{p}$ and $\tilde{q}$ with $\tilde{p}_0 \neq p_0 \vee \tilde{q}_0 \neq q_0$ and an honest P_1 does not abort the execution (where p_0, q_0 are the decryptions of ciphertexts sent in Step 2a using the ElGamal scheme). Then, conditioned on bad, we have that

$$\begin{aligned}\tilde{q}_0 p_1 + \tilde{p}_0 q_1 + q_1 p_1 + \tilde{q}_0 \tilde{p}_0 &\equiv \tilde{N} \bmod N_0 \\ &\equiv N \bmod N_0 \\ &\equiv q_0 p_1 + p_0 q_1 + q_1 p_1 + q_0 p_0 \bmod N_0,\end{aligned}$$

since the execution of π_{MULT} ensures that P_0 correctly adds the product of the two initial values in the final step, and $\tilde{N} \neq N$ would be caught in the following step, causing P_1 to abort. This implies that we know $\alpha = \tilde{q}_0 - q_0,\ \ \beta = \tilde{p}_0 - p_0,\ \ \gamma = \tilde{q}_0 \tilde{p}_0 - q_0 p_0 \in \mathbb{Z}_{N_0}$ such that

$$\alpha \cdot p_1 + \beta \cdot q_1 + \gamma \equiv 0 \bmod N_0$$

and either $\alpha \neq 0$ or $\beta \neq 0$. These values will be used to determine information about p_1 and q_1 which may, in turn, be used to break the IND-CPA security of ElGamal.

More formally, assume bad occurs with probability ϵ. Then, construct an adversary $\mathcal{A}_{\mathrm{EG}}$ that breaks the IND-CPA security of ElGamal with probability $1/2 + \epsilon/2$ as follows. Given an ElGamal public-key

$pk'_{\mathrm{EG}} = (g, h')$, and a ciphertext c_b which is an encryption of $b \in \{0,1\}$, $\mathcal{A}_{\mathrm{EG}}$ simulates the initial key generation by replacing pk_{EG} with pk'_{EG} (faking the proof that knows part of the decryption key). $\mathcal{A}_{\mathrm{EG}}$ then extract p_0 and q_0 from $\mathcal{A}$'s ElGamal encryptions and generates two independent sets of candidates, $p_1^{(0)}, p_1^{(1)}, q_1^{(0)}, q_1^{(1)}$ which will pass trial division. $\mathcal{A}_{\mathrm{EG}}$ then computes encryptions of one of the pairs:

$$c_{p_1}^{(b)} = (c_b)^{p_1^{(1)} - p_1^{(0)}} \mathsf{Enc}\left(p_1^{(0)}\right) \qquad c_{q_1}^{(b)} = (c_b)^{q_1^{(1)} - q_1^{(0)}} \mathsf{Enc}\left(q_1^{(0)}\right).$$

Note that in case $b = 0$, then $c_{p_1}^{(b)}$ and $c_{q_1}^{(b)}$ correspond to encryptions of $p_1^{(0)}$ and $q_1^{(0)}$, respectively. Otherwise, we get encryptions of $p_1^{(1)}$ and $q_1^{(1)}$. $\mathcal{A}_{\mathrm{EG}}$ then simulates the trial division (which is passed), and finally reach the step, where it computes $\tilde{N}$, and extracts $\tilde{p}_0$ and $\tilde{q}_0$ from $\mathcal{A}$. Note that this allows $\mathcal{A}_{\mathrm{EG}}$ to compute α, β, and γ without even completing this step. $\mathcal{A}$ outputs 0 if and only if

$$\alpha \cdot p_1^{(0)} + \beta \cdot q_1^{(0)} + \gamma \equiv 0 \bmod N_0.$$

Due to π_{RSA}, it holds that N_0 is the product of two large primes with overwhelming probability. Neglecting the event where N_0 is a product of more than two primes, it holds that p_1 and q_1 are co-primes to N_0. In addition, if either $\mathrm{GCD}(\alpha, N_0) \neq 1$ or $\mathrm{GCD}(\beta, N_0) \neq 1$, we may compute a factor f of N_0, such that

$$\alpha \not\equiv 0 \bmod f \vee \beta \not\equiv 0 \bmod f$$

and both α and β co-prime with f unless congruent to 0. Otherwise let f equal the trivial factor, N_0. As the execution would pass if it continued, it must holds that

$$\alpha \cdot p_1 + \beta \cdot q_1 + \gamma \equiv 0 \bmod f,$$

i.e. we find a linear equation in p_1 and q_1. However, these will be either $p_1^{(0)}, q_1^{(0)}$ or $p_1^{(1)}, q_1^{(1)}$ depending on the unknown bit chosen by the ElGamal oracle. Hence, if $\mathcal{A}$ can cheat with probability ϵ, then $\mathcal{A}_{\mathrm{EG}}$ breaks the IND-CPA security of ElGamal encryption with probability $1/2 + \epsilon/2$ by checking which pair of candidates gives a linear equation of the form specified above. Formally,

$$\begin{aligned}
&\mathsf{Adv}_{\Pi_{\mathrm{EG}}, \mathcal{A}_{\mathrm{EG}}}(n) \\
&= \frac{1}{2}\left(\Pr[\mathcal{A}_{\mathrm{EG}}(c_{p_1}^{(b)}, c_{q_1}^{(b)}) = 0 | b = 0] + \Pr[\mathcal{A}_{\mathrm{EG}}(c_{p_1}^{(b)}, c_{q_1}^{(b)}) = 1 | b = 1]\right) \\
&= \frac{1}{2}\left|\Pr[\mathcal{A}_{\mathrm{EG}}(c_{p_1}^{(b)}, c_{q_1}^{(b)}) = 0 | b = 0] - \Pr[\mathcal{A}_{\mathrm{EG}}(c_{p_1}^{(b)}, c_{q_1}^{(b)}) = 0 | b = 1]\right| + \frac{1}{2} \\
&= \Pr[\mathsf{bad} | b = 0] - \mathsf{negl} \geq \frac{\epsilon}{2} + \frac{1}{2}
\end{aligned}$$

where negl is a negligible function in the security parameter. The reason $\mathcal{A}_{\mathrm{EG}}$ outputs 0 with negligible probability in the case where $b = 1$ is due to the fact that $p_1^{(0)}, q_1^{(0)}$ are independent of $p_1^{(1)}, q_1^{(1)}$ and information theoretic hidden from $\mathcal{A}$ (as $\mathcal{A}_{\mathrm{EG}}$ either uses $p_1^{(0)}, q_1^{(0)}$ or $p_1^{(1)}, q_1^{(1)}$). Therefore, the probability that the outcome forms a linear equation for $p_1^{(0)}, q_1^{(0)}$ is negligible.

P_1 is corrupted. Let $\mathcal{A}$ denote an adversary controlling party P_1 and consider the event bad in which the honest P_0 does not abort even though the malicious P_1 computes $c_{\tilde{N} - p_0 q_0}$ differently than specified. Note that even if $\mathcal{A}$ returns a ciphertext $c_{\tilde{N} - p_0 q_0}$, encrypting $N - p_0 q_0$ and computed differently than specified in the protocol, the ZK proof π_{VERLIN} enables to extract values $x, x', x'' \in \mathbb{Z}_{N_0}$ and $r_x \in \mathbb{Z}_{N_0}^*$ such that

$$c_{\tilde{N} - p_0 q_0} = c^x \cdot \left(c'\right)^{x'} \cdot \mathsf{Enc}(x'', r_x),$$

where c is an encryption of p_0 and c' is an encryption of of q_0. These values will be used to break either the IND-CPA security of either ElGamal or Paillier encryptions.

First, construct a distinguisher $\mathcal{A}_{\text{Pa}}$ based on $\mathcal{A}$ which breaks the semantic security of Paillier encryption. Specifically, simulate steps 1 and 2 as above but also extract p_1 and q_1 during the simulation. In step 3, $\mathcal{A}_{\text{Pa}}$ is given a Paillier key to be used as N_0 and an encryption c_b of a bit b. Similarly to above, we generate two sets of candidates, $p_0^{(0)}, p_0^{(1)}, q_0^{(0)}, q_0^{(1)}$ which will pass trial division, and compute encryptions of one of the pairs:

$$c_{p_0^{(b)}} = (c_b)^{p_0^{(1)} - p_0^{(0)}} \mathsf{Enc}\left(p_0^{(0)}\right) \qquad c_{q_0^{(b)}} = (c_b)^{q_0^{(1)} - q_0^{(0)}} \mathsf{Enc}\left(q_0^{(0)}\right).$$

We send $c_{p_0^{(b)}}$ and $c_{q_0^{(b)}}$ as the initial messages in the $\tilde{N}$-computation and fake the proofs of known plaintext. Once $\mathcal{A}$ has returned $c_{\tilde{N}-p_0 q_0}$ (which is an encryption of $N - p_0^{(b)} q_0^{(b)}$ by assumption), we extract x, x', x'', and r_x. Again, we do not need to finish this step of the protocol, but may directly use these values to break the IND-CPA security.

As $\tilde{N} = N$, we have the following equation in the two unknowns $p_0^{(b)}$ and $q_0^{(b)}$

$$\begin{aligned} & x p_0^{(b)} + x' q_0^{(b)} + x'' \equiv q_1 p_0^{(b)} + p_1 q_0^{(b)} + p_1 q_1 \bmod N_0 \\ \Updownarrow \\ & (q_1 - x)\, p_0^{(b)} + \left(p_1 - x'\right) q_0^{(b)} + \left(p_1 q_1 - x''\right) \equiv 0 \bmod N_0 \end{aligned}$$

There are two cases:

Case 1: $q_1 - x = 0 \vee p_1 - x' = 0$**.** In this case, we can compute either $p_0^{(b)}$ or $q_0^{(b)}$ by:

$$p_0^{(b)} = \left(x'' - p_1 q_1\right)(q_1 - x)^{-1} \qquad q_0^{(b)} = \left(x'' - p_1 q_1\right)(p_1 - x)^{-1}.$$

This calculation requires that the non-zero value is invertible. However, if it is not, then we may compute a factor of N_0 using GCD. This allows us to compute the decryption key ourselves, which trivially breaks the security of Paillier.

Case 2: $q_1 - x \neq 0 \wedge p_1 - x' \neq 0$**.** Again, we may assume that both $q_1 - x$ and $p_1 - x'$ are invertible. Similarly to the case of a corrupt P_0, we now have a linear equation in $p_0^{(b)}$ and $q_0^{(b)}$, and we can easily check which pair, $\left(p_0^{(0)}, q_0^{(0)}\right)$ or $\left(p_0^{(1)}, q_0^{(1)}\right)$, fits the equation; this breaks the security of Paillier encryption as demonstrated above for ElGamal.

Similarly to the above, we can design a reduction to the security of ElGamal by computing encryptions of $p_0^{(b)}$ and $q_0^{(b)}$. Then in the computation of $\tilde{N}$, we obtain x, x', x'', and r_x, which again splits into two cases and used to break IND-CPA security.

4.2 The Biprimality Test

The distributed biprimality test for checking the validity of a candidate for being an RSA composite, is based on a test by Boneh-Franklin [BF01] where the parties first agree on a random element $\gamma \in \mathbb{Z}_N^*$ with Jacobi symbol 1, and then raise γ to a power calculated from their shares. The test accepts a number with more than two prime factors with probability at most $1/2$. Therefore, the parties must repeat this test sufficiently many times in order to decrease the error. We adopt this test for the malicious setting. As a side remark, we note that although the biprimality test by Damgård and Mikkelsen [DM10] has a better error estimate, it cannot be used efficiently in the two-party setting with malicious adversaries. In Appendix B we show how to adapt their test into the two-party setting when the parties are semi-honest.

Protocol 2 [DPrim] *A distributed biprimality test:*

- **Inputs:** *A security parameter* 1^n*, a statistical parameter* 1^ℓ *and a public-key candidate* N*.*
- **The Protocol:**
  1. *The parties jointly generate a random element* $\gamma \in \mathbb{Z}_N^*$ *with Jacobi symbol* $\mathcal{J}(\gamma) = 1$*. By standard techniques this is made secure against active deviation.*
  2. *The parties compute the encryption* $e_0 = \mathsf{Enc}_{pk_{\mathrm{EG}}}\left(\frac{N-(p_0+q_0)+1}{4}\right)$ *using the homomorphic property of ElGamal (*P_1 *knows the encryptions of* p_0 *and* q_0 *from the earlier protocol). Furthermore,* P_0 *sends* $\gamma_0 = \gamma^{\left(\frac{N+1-(p_0+q_0)}{4}\right)} \bmod N$ *and proves consistency between* e_0 *and* γ_0 *using* π_{EQ}*.*
  3. P_1 *sends* $\gamma_1 = \gamma^{\left(\frac{-(p_1+q_1)}{4}\right)} \bmod N$ *to* P_0 *and proves consistency using* π_{EQ} *to an encryption* e_1 *of* $\frac{-(p_1+q_1)}{4}$*, computed as above.*
  4. *Finally, the parties reject* N *if and only if* $\gamma_0 \cdot \gamma_1 \bmod N \neq \pm 1$*. We further note that the test by [BF01] includes an additional step were instead of using* γ*, the parties randomly pick an element from the group* $(\mathbb{Z}_N[x]/(x^2+1))^*/\mathbb{Z}_N^*$*; we omit the details due to the similarity of the above test.*
  5. *This test is repeated* ℓ *times to achieve sufficiently small error.*

Theorem 4.2 *Assuming hardness of the DDH problem, Protocol 2 is a distributed* Monte Carlo *algorithm such that on a statistical parameter* 1^ℓ *and a random* γ*, it holds that*

- *A correctly formed RSA modulus* $N = pq$*, where* $p \equiv q \equiv 3 \pmod 4$ *is always accepted.*
- *The average case probability of accept if either* p *or* q *is a composite, is at most* $2^{-\ell}$*.*
- *The protocol is secure (simulatable with abort without knowledge of the factorization of* N*) in the presence of malicious adversaries.*

Informally, correctness follows from [BF01] and security is proven by simulation where the simulator is able to simulate the corrupted party's view by having knowledge of the adversaries shares of p and q, and thereby being able to calculate γ_0 or γ_1, respectively. With this knowledge the simulator can simulate acceptance of a modulus without knowledge of the factorization. A complete proof follows.

Proof: The theorem states both correctness and security. By correctness we mean: If the parties do not deviate, then Blum integers are never rejected whereas, numbers with more than two prime factors are rejected with probability at least $2^{-\ell}$. Since the biprimality test is identical to the one of Boneh and Franklin [BF01], we refer the reader to [BF01] for a proof of correctness.

Security is proven in the $\mathcal{F}_{\mathrm{EQ}}$-hybrid model. We assume that simulator $\mathcal{S}$ has knowledge of the shares of adversary $\mathcal{A}$ from protocol DKeyGen for distributively generating an RSA composite. Observe that there are two possible outcomes of DPrim, either N is rejected, in which case N is not a Blum integer in the real protocol. This can easily be simulated by $\mathcal{S}$ choosing shares on behalf of the honest player such that N is not a Blum integer and executing the real protocol as the honest party would. It is easy to verify that if $\mathcal{A}$ deviates then it cannot make π_{EQ} accept. In the other case, N is a Blum integer and should be accepted. Therefore, $\mathcal{S}$ has to be able simulate $\mathcal{A}$'s view without knowing the factorization of N, which is possible in the following way. Assume that P_0 is corrupted by $\mathcal{A}$; the simulation is analog to the other corruption case.

1. $\mathcal{S}$ emulates the choice of γ by choosing a uniform random value $a \in \mathbb{Z}_N^*$ with Jacobi symbol $\mathcal{J}(a) = 1$ and a uniform random bit $b \in \{0,1\}$ and fixing $\gamma = a^2(-1)^b \bmod N$.

2. Each player is supposed to calculate $e_0 = \mathsf{Enc}_{pk_{\mathrm{EG}}}\left(\frac{N-(p_0+q_0)+1}{4}\right)$.

   $\mathcal{S}$ receives $\gamma_{\mathcal{A}}$ from $\mathcal{A}$, and since $\mathcal{S}$ has knowledge of the shares p_0 and q_0 of $\mathcal{A}$, $\mathcal{S}$ knows a priori the expected value of $\gamma_{\mathcal{A}}$. $\mathcal{S}$ emulates $\mathcal{F}_{\mathrm{EQ}}$, by receiving $\left(\frac{N-(p_0+q_0)+1}{4}\right)$ as withness, and verifying whether $\gamma_{\mathcal{A}}$ is calculated correct. (Abort if $\gamma_{\mathcal{A}}$ is not the expected value)

3. Each player is supposed to calculate $e_1 = \mathsf{Enc}_{pk_{\mathrm{EG}}}\left(\frac{-(p_1+q_1)}{4}\right)$, we note that this is done with the simulated values for p_1 and q_1.

   $\mathcal{S}$ sends $\gamma_{\mathcal{S}} = (\gamma_{\mathcal{A}})^{-1}(-1)^b \bmod N$ to $\mathcal{A}$ and emulates $\mathcal{F}_{\mathrm{EQ}}$ accepting.

4. Finally, the parties rejects N if and only if $\gamma_{\mathcal{A}} \cdot \gamma_{\mathcal{S}} \bmod N \neq \pm 1$ (in this iteration they always accept).

   The simulation of the additional test in the group $(\mathbb{Z}_N[x]/(x^2+1))^*/\mathbb{Z}_N^*$ is analog to the simulation described above.

Note the following:

I. Because N is a Blum integer, the size of the subgroup of quadratic residues QR in $\mathbb{Z}_N^*$ is half the size of the subgroup of elements with Jacobi symbol 1 in $\mathbb{Z}_N^*$. Furthermore, -1 is a quadratic nonresidue modulo N with Jacobi symbol $\mathcal{J}(-1) = 1$. This means that γ is a uniform random element of $\mathbb{Z}_N^*$ with Jacobi symbol $\mathcal{J}(\gamma) = 1$. This is the exact distribution in the real execution, since we assume $\mathcal{A}$ cannot influence the distribution of γ in DPrim.

II. In the real world execution of DPrim the following holds:

$$\gamma_0 \cdot \gamma_1 \equiv 1 \mod N \text{ if } \gamma \in \mathrm{QR}.$$
$$\gamma_0 \cdot \gamma_1 \equiv -1 \mod N \text{ if } \gamma \notin \mathrm{QR}.$$

   From b, $\mathcal{S}$ knows whether γ is in QR or not, and therefore $\gamma_{\mathcal{S}}$ is simulated perfectly by $\gamma_{\mathcal{S}} = (\gamma_{\mathcal{A}})^{-1}(-1)^b \bmod N$.

Finally, since all values except the encryptions e_0 and e_1 are simulated perfectly $\mathcal{A}$ can only distinguish the real and hybrid executions if it can break the sematic security of ElGamal, and therefore break the DDH assumption. ■

5 A Complete Threshold Paillier Cryptosystem

In the following section, we describe our threshold construction in the two-party setting for the Paillier encryption scheme [Pai99]. Our Threshold Paillier Scheme, $\mathbf{TPS}$, is comprised of the following subprotocols: (**i**) The protocol $\mathsf{DKeyGen}$ (cf. Section 4) for distributed generation of an RSA composite. (**ii**) A protocol for distributed generation of the corresponding secret-key shares, denoted by Dsk (cf. Section 5.1.1). (**iii**) A protocol for distributed decryption, denoted by DDec, for decrypting according to Paillier's specifications while maintaining the randomness of the ciphertext a secret (cf. Section 5.1.2). These protocols rely on the following standard hardness assumptions: (**1**) DDH (cf. Definition 2.1), due to employing the ElGamal scheme [ElG85] and (**2**) DCR (cf. Definition 2.2), due to employing the Paillier scheme [Pai99] and integer commitments [DN02, DN03].

Our protocols form the *first complete threshold scheme for Paillier* in the two-party setting with security in the presence of malicious adversaries under full simulation based definitions, following the ideal/real model paradigm. We denote by $\Pi = (\mathsf{Gen}', \mathsf{Enc}, \mathsf{Dec})$ the Paillier encryption scheme that is depicted in

Section 2.2.1, with the modified key generation algorithm Gen' specified in Section 4, encryption algorithm Enc and decryption algorithm Dec. The formal description of the threshold functionality, $\mathcal{F}_{\text{THRES}}$ is found in Figure 2.

Theorem 5.1 *Assuming hardness of the DDH and DCR problems, scheme* $\mathbf{TPS} = (\mathsf{DKeyGen}, \mathsf{Dsk}, \mathsf{DDec})$ *computes functionality* $\mathcal{F}_{\text{THRES}}$ *in the presence of malicious adversaries.*

Proof: The proof for this theorem follows from the proofs for Theorems 4.1, 4.2, 5.2 and 5.3. That is, Theorems 4.1 and 4.2 form a complete key generation protocol, where the parties compute an RSA composite without leaking its factorization. Moreover, Theorems 5.2 and 5.3 guarantee that the parties decrypt according to Paillier in a secure manner. ■

Functionality $\mathcal{F}_{\text{THRES}}$

KEY GENERATION: Identical to KEY GENERATION in $\mathcal{F}_{\text{GEN}}$.

DECRYPTION: Upon receiving a $(\mathsf{Decrypt}, c, \mathsf{Both})$ message from party P_i, $\mathcal{F}_{\text{THRES}}$ continues as follows:

1. If there exists a recorded secret key, then $\mathcal{F}_{\text{THRES}}$ forwards $(\mathsf{Decrypt}, c, \mathsf{Both})$ to party P_{1-i}.
   (a) If P_{1-i} replies with allow and $\mathsf{Both} = \phi$, $\mathcal{F}_{\text{THRES}}$ sends $\mathsf{Dec}_{sk}(c)$ only to P_i.
   Otherwise, if $\mathsf{Both} \neq \phi$, $\mathcal{F}_{\text{THRES}}$ sends $\mathsf{Dec}_{sk}(c)$ to both parties.
   (b) If P_{1-i} replies with disallow, then $\mathcal{F}_{\text{THRES}}$ forwards disallow to P_i.
2. If there does not exist a recorded secret key, functionality $\mathcal{F}_{\text{THRES}}$ sends disallow to P_i.

Figure 2: The (Paillier) Threshold Functionality

5.1 A Distributed Decryption for Paillier

In this section we present a secure decryption protocol in the distributed setting. We first note that a typical way to decrypt is to use the algorithm of the RSA scheme, where the decrypter raises the ciphertext to the power of the inverse of N modulo $\phi(N)$, as shown in [CGHN01]. This similarity follows because both schemes have the same public key of an RSA composite N and a secret key that is the factorization of N. Furthermore, the ciphertexts in both schemes have similar algebraic structure.

However, in some scenarios this type of algorithm may be problematic, since the decrypter must extract first the randomness used for computing the ciphertext in order to complete the decryption. As desirable as this property may be, it is problematic in the context of simulation based secure computation because the parties have to present the randomness of the ciphertext instead of proving correctness using ZK proofs. This means that a potential simulator cannot cheat in the decryption by encrypting arbitrary values and then fake their decryption. We further note that Paillier's scheme requires extra computation in order to complete the decryption. This means that on top of raising the ciphertext to the power of the secret value, the outcome must be multiplied with the inverse of the secret key in order to extract the plaintext. In the distributive setting this implies that the parties must keep two types of shares. When coping with malicious behavior it is not immediately clear how to efficiently verify the parties' computations. Notably, the protocol of Damgård and Jurik [DJ01] circumvents this technicality by having a trusted party picking a secret $d \equiv 1 \bmod N \equiv 0 \bmod \phi(N)$.

Our protocol offers a distributive decryption for Paillier with simulation based security against malicious adversaries without randomness extraction. It is comprised of the following two subprotocols: First, the

parties produce shares of a value d similarly to the Damgård-Jurik scheme [DJ01]. This protocol is executed only once. Next, the parties run the distributed decryption algorithm using their shares. As mentioned earlier we use the simplified encryption function of Damgård and Jurik, i.e., use $g = N + 1$ as a generator of the subgroup of $\mathbb{Z}_{N^2}^*$ of order N. Encryption of a plaintext m with randomness r is then,

$$\mathsf{Enc}_N(m, r) = r^N \cdot (N+1)^m \bmod N^2.$$

5.1.1 Generating a Shared Paillier Decryption Key

We now present our protocol for generating a shared Paillier decryption key. As stated, similarly to Damgård and Jurik [DJ01], we share a decryption exponent as follows

$$d \equiv \begin{cases} 0 \bmod \phi(N) \\ 1 \bmod N \end{cases}$$

However, since there are only two parties, a full threshold sharing is not necessary and we therefore use an additive sharing. Initially, we focus on the task at hand and present a protocol with semi-honest security (where the parties are assumed to follow the protocol's description). This is then compiled into a protocol with full active security towards both parties by adding zero-knowledge proofs, ensuring that the parties cannot deviate.

Protocol 3 [Dsk] *A distributed generation of a shared Paillier decryption key with passive security:*

- **Inputs:** *A public RSA modulus $N = pq$ with unknown factorization, additive shares of $\phi(N)$: $sk_0 = N - p_0 - q_0 + 1$ and $sk_1 = -p_1 - q_1$ held by P_0 and P_1 respectively. A public ElGamal key (g, h) with the secret key shared between the parties. A public Paillier key $N_0 \gg N^2$ with the secret key held by P_0.*
- **The protocol:**
  1. *P_0 encrypts sk_0 using N_0 and sends this to P_1.*
  2. *P_1 picks $r_1 \in \mathbb{Z}_N^\star$ and $r_\sigma \in \mathbb{Z}_{2^{\log N + \kappa}}$ uniformly at random (for a statistical parameter κ that enables to mask the secret key). P_1 computes an encryption of*

     $$(sk_0 + sk_1) \cdot r_1 + N \cdot r_\sigma$$

     using the homomorphic property of Paillier encryption. This is rerandomized and sent to P_0.
  3. *P_0 decrypts, thus obtaining plaintext r_0; P_0 computes ${r_0}^{-1} \bmod N$ and encrypts this as well as plaintext $sk_0({r_0}^{-1} \bmod N)$ under public-key N_0. Both ciphertexts are sent to P_1.*
  4. *Based on the encryptions of ${r_0}^{-1} \bmod N$ and $sk_0({r_0}^{-1} \bmod N)$, P_1 computes an encryption of*

     $$\begin{aligned} d &= \left(sk_0({r_0}^{-1} \bmod N) \cdot r_1 + ({r_0}^{-1} \bmod N)(sk_1 \cdot r_1)\right) \\ &= r_1(sk_0 + sk_1)({r_0}^{-1} \bmod N) \\ &= r_1 \cdot \phi(N) \cdot ({r_0}^{-1} \bmod N). \end{aligned} \tag{1}$$

     P_1 then picks $\tilde{d}_1$ uniformly at random in $\mathbb{Z}_{2^{3\log N + \kappa}}$, and computes and rerandomizes an encryption of $d + \tilde{d}_1$. This is sent to P_0 and finally, P_1 sets its share of d to the integer $-\tilde{d}_1$.
  5. *P_0 decrypts and obtains d_0: its share of d.*

Correctness. Since no overflow modulo N_0 occurs in Eq. (1), all calculations can be viewed to occur over the integers. Thus, since $\phi(N)$ is a factor of d, clearly $d \equiv 0 \bmod \phi(N)$. Moreover, as $r_0 \equiv \phi(N) \cdot r_1 \bmod N$ we also have $d \equiv 1 \bmod N$. Finally, as $d = d_0 + d_1$ over the integers, clearly we have an additive sharing of a value with the desired property.

Theorem 5.2 *Assuming hardness of the DDH and DCR problems, Protocol 3 computes additive shares of a value d specified above in the presence of semi-honest adversaries.*

Proof: The case follows by two corruption cases.

P_0 is corrupted. We demonstrate how to generate a view statistically close to that of P_0. During the protocol, P_0 receives two messages: the encryptions of r_0 and d_0. As the encryptions have been rerandomized, they are indistinguishable from fresh encryptions of the plaintext values. Moreover, both of these are statistically close to uniformly random:

- $r_0 \bmod N$ is uniformly random in $\mathbb{Z}_N^*$ due to the multiplication by r_1.
- $\lfloor r_0/N \rfloor$ is statistically close to a uniformly random $\log N + \kappa$ bit integer.
- d_0 is statistically close to a uniformly random $3 \log N + \kappa$ bit integer.

Hence, to simulate, it suffices to pick and encrypt values distributed as the masks generated by P_1.

P_1 is corrupted. We cannot achieve unconditional security towards P_1, however, a computationally indistinguishable view may be generated. During the protocol, the messages received by P_1 consist of three Paillier encryptions under the key N_0. Assuming that Paillier encryption is semantically secure, this is indistinguishable from three encryptions of 0. ■

We now demonstrate how to achieve security against malicious adversaries by adding zero-knowledge proofs to support correct behavior of each party.

ENSURING CORRECTNESS OF P_0. First, P_0 must show that the plaintexts of the three encryptions sent in Steps 1 and 3 are as specified by the protocol. Demonstrating that the former encryption contains sk_0 is equivalent to the same task as in the calculation of N (see Protocol 1). Indeed P_0 could simply reuse the encryption from that protocol. Similarly, demonstrating that the later encryption contains the product of the two first ones is simply obtained by an invocation of π_{MULT}.

The more challenging part is to have P_0 demonstrate that the second plaintext is the modulo N inverse of the encrypted value received in Step 3. This requires P_0 to send additional encryptions and execute zero-knowledge proofs based on the following:

1. P_0 executes π_{BOUND} on the encryption of $r_0^{-1} \bmod N$, demonstrating that it is less than N.
2. P_0 sends an encryption of $r_0^{\perp} = r_0 \bmod N$ under N_0 and uses π_{BOUND} to demonstrate that it is less than N.
3. P_0 sends an encryption of $r_0^{\top} = \lfloor r_0/N \rfloor$ under N_0 and demonstrates that $r_0^{\perp} + N \cdot r_0^{\top}$ equals r_0, i.e., equals the plaintext of the encryption received from P_1, using the zero-knowledge proof π_{ZERO}. (I.e., the above boils down to proving that the division of two ciphertexts in an N_0th root.)
4. P_0 sends an encryption of $r_0^{\perp} \cdot (r_0^{-1} \bmod N)$ and demonstrates correct multiplication using π_{MULT}.

5. Both parties compute an encryption of $(r_0^{\perp} \cdot ({r_0}^{-1} \bmod N) - 1) \cdot N^{-1}$; P_0 executes π_{BOUND} to show that the value is less than N.

Note that Steps 1, 2 and 3 demonstrate that $r_0^{\perp} = r_0 \bmod N$. Thus, it remains to demonstrate that the inversion modulo N has been performed properly. This must be the case, as the execution of π_{BOUND} in Step 5 only succeeds if N divides $(r_0^{\perp} \cdot ({r_0}^{-1} \bmod N) - 1)$. Namely, if the product is congruent to $1 \bmod N$. This completes the description of the modified role of P_0.

ENSURING CORRECTNESS OF P_1. We continue with the description of P_1. Security against malicious P_1 follows similarly to the security against a malicious P_1 during the computation of N. More specifically, P_1 is required to prove the correctness of computation specified by the following linear equations $(sk_0 + sk_1) \cdot r_1 + N \cdot r_\sigma$ in step 2 and $\big(sk_0({r_0}^{-1} \bmod N) \cdot r_1 + ({r_0}^{-1} \bmod N)(sk_1 \cdot r_1)\big)$ in step 4, where the variables introduced by P_1 are r_1, r_σ and sk_1. This is done as follows:

- Proving honest behavior in step 2:
    1. P_1 encrypts r_1 and r_σ using the distributed homomorphic ElGamal scheme, and sends the ciphertexts to P_0.
    2. P_1 is using π_{VERLIN} to prove knowledge of the values $(sk_1 \cdot r_1)$, r_1 and $(N \cdot r_\alpha)$ and correct computation of the encryption of $(sk_0 + sk_1) \cdot r_1 + N \cdot r_\sigma = sk_0 \cdot r_1 + sk_1 \cdot r_1 + N \cdot r_\sigma$.
    3. After P_0 has decrypted r_0, P_0 and P_1 re-execute the computation using the homomorphic ElGamal scheme, and using π_{MULT} to prove correct multiplication. The result is decrypted to P_0, which verifies whether the decryption of both schemes results in the same value r_0. If this is the case P_1 has been following the protocol honestly.
- Proving honest behavior in step 4:
    1. P_1 starts by picking $\tilde{d}_1$ and encrypting it using the distributed homomorphic ElGamal scheme, and sends the ciphertexts to P_0 along with a proof of knowledge using π_{ENC}.
    2. P_1 is using π_{VERLIN} to prove knowledge of the values $(sk_1 \cdot r_1)$, r_1 and $\tilde{d}_1$ and correct computation of the encryption of $\big(sk_0({r_0}^{-1} \bmod N) \cdot r_1 + ({r_0}^{-1} \bmod N)(sk_1 \cdot r_1)\big) + \tilde{d}_1$.
    3. Again P_0 and P_1 re-execute the computation using the homomorphic ElGamal scheme, and using π_{MULT} to prove correct multiplication. The result is decrypted to P_0, which verifies whether the decryption of both schemes results in the same value d_0. If this is the case P_1 has been following the protocol honestly.

Since P_1 does not explicitly prove consistency between the encryptions using the Paillier scheme and the ElGamal scheme we need to verify that a corrupted P_1 cannot return encryptions giving the same result which are computed using different values. If that was possible it would make simulating the security impossible. In step 2, it follows easily that if P_1 is capable of returning a Paillier Encryption where different values of $(sk_1 \cdot r_1)$, r_1 and $(N \cdot r_\alpha)$ are used compared to the ElGamal encryption and that the two computations give the same result, then P_1 is able to compute sk_0 from either the Paillier encryption or the ElGamal encryption, which is a contradiction. In step 4, if P_1 is capable of returning a Paillier encryption and an ElGamal encryptions giving the same result, but using different values for $(sk_1 \cdot r_1)$, r_1 and $\tilde{d}_1$, there are two cases, each described below. In the following we label the values used in the Paillier encryption as $(sk_1 \cdot r_1)'$, r_1' and $\tilde{d}_1'$:

1. $\tilde{d}_1 = \tilde{d}_1'$: In this case the corrupt P_1 is capable of computing sk_0 since: $sk_0 = \frac{(sk_1 \cdot r_1) - (sk_1 \cdot r_1)'}{(r_1' - r_1)}$. This is a contradiction since P_1 has only seen encryptions of sk_0. (Note, since $\tilde{d}_1 = \tilde{d}_1'$ then $r_1' \neq r_1$).

2. $\tilde{d_1} \neq \tilde{d_1'}$: In this case P_1 is able to compute a value δ which is divisible by $({r_0}^{-1} \bmod N)$. This is a contradiction since P_1 has only seen semantically secure encryption (Paillier and ElGamal) of $({r_0}^{-1} \bmod N)$. The value δ is computed as:

$$\delta = \tilde{d_1} - \tilde{d_1'} = \big(sk_0 \cdot (r_1' - r_1) + (sk_1 \cdot r_1)' - (sk_1 \cdot r_1)\big) \cdot ({r_0}^{-1} \bmod N).$$

5.1.2 Performing a Joint Paillier Decryption

To perform a joint decryption of some ciphertext c, both parties need to raise c to their share of the key, d_0 or d_1. They then demonstrate that this has been computed correctly using the commitments of the shares. The plaintext is immediately computable from c^{d_0} and c^{d_1}.

Protocol 4 [DDec] *A distributed Paillier decryption with a shared key:*

- **Inputs:** *A public Paillier key $N = pq$ with unknown factorization and a ciphertext $c = E_N(m, r)$. Party P_i holds its share d_i of the secret decryption exponent, $d = d_0 + d_1$ where $d \equiv 1 \bmod N \wedge d \equiv 0 \bmod \phi(N)$. Finally, the parties hold commitments to (or rather: ElGamal encryptions of) their key-shares.*
- **The protocol:**
  1. *P_0 sends $c_0 = c^{sk_0} \bmod N^2$ to P_1. Moreover, P_0 demonstrates that this has been done correctly by executing π_{EQ}, i.e., that the committed number equals the discrete log of c_0 with base c and the plaintext encrypted with ElGamal.*
  2. *P_1 sends $c_1 = c^{sk_1} \bmod N^2$ to P_0. Moreover, P_1 demonstrates that this has been done correctly by executing π_{EQ}, i.e., that the committed number equals the discrete log of c_1 with base c and the plaintext encrypted with ElGamal.*
  3. *Finally, both parties compute the plaintext, $m = ((c_0 \cdot c_1) \bmod N^2 - 1)/N$.*

Theorem 5.3 *Assuming that π_{EQ} is a zero-knowledge proof of knowledge that correctly demonstrates equality of discrete logarithms, Protocol 4 determines the plaintext, m, of ciphertext c in the presence of malicious adversaries.*

Proof: Simulation in a hybrid model with access to $\mathcal{F}_{\text{EQ}}$ is straightforward. Namely, from P_0 viewpoint the only message received is c_1, since the message of π_{DH} are handled by $\mathcal{F}_{\text{EQ}}$. This value may be computed deterministically given the plaintext by:

$$c_1 \leftarrow (1+N)^m \bmod N^2 \cdot c^{-d_0} \bmod N^2.$$

Clearly this is the value sent by P_1 during the protocol execution, as

$$c^{d_1} = c^{d-d_0} = c^d \cdot c^{-d_0} \equiv (1+N)^m \cdot c^{-d_0} \bmod N^2.$$

I.e., c_1 can be computed using values known to the simulator. The case of corrupt P_1 is equivalent. ■

6 The Efficiency of Our Protocols

In this section we discuss the efficiency of our protocols. We split our discussion into two parts: a theoretical analysis with a focus on the asymptotic complexity and optimizations that yield a more practical analysis.

6.1 The Number of Failed Attempts

First, the complexity of our protocol depends heavily on the number of failed attempts when generating the modulus. Recall that without running a trial division the protocol has to restart with two freshly generated prime candidates after every rejected composite, or otherwise the leaked information would make factoring an accepted composite easy. Specifically, without the trial division the expected number of tests is given by the probability of choosing two random primes simultaneously. This can be calculated by the Prime Number Theorem, making the expected number of executions: 512 bit primes: $\left(\ln(2^{512})/2\right)^2 \approx 31000$, 1024 bit primes: $\left(\ln(2^{1024})/2\right)^2 \approx 126000$.

Nevertheless, this can be dramatically improved when employing the trial division test. Following the analysis of [BF01] it can be shown that the probability a generated candidate is a prime, given that it passes the trial division, is computed due to [DeB] and is as follows,

$$\Pr[p \text{ is prime} \mid p \text{ passes trial division with threshold B}] = 2.57 \cdot \frac{\ln \mathrm{B}}{\mathrm{n}} \left(1 + \mathrm{o}\left(\frac{1}{\mathrm{n}}\right)\right)$$

which for $\ln B = 9$ (i.e., $B = 8103$) and $n = 1024$ is $1/44$. This means that our protocol needs to test an expected number of 1936 candidates if $n = 1024$. This shows how important the trial division is for the efficiency of our protocol, which is the first to incorporate this test securely in the two-party setting. Note that this analysis is independent of the construction used for generating the composite and strictly relies on the primes density in a given interval.

6.2 Theoretical Efficiency

We remark that all of our zero-knowledge proofs run in constant round and require constant number of exponentiations; the only exception is π_{EQ}, employed in our threshold decryption protocol, for which there is an amortized constant analysis due to Cramer and Damgård [CD09].

Key Generation. Ignoring the initial key-setup, the complexity of a single RSA composite generation attempt (except for the biprimality test that is separately analyzed below), is dominated by the number of trial divisions; the rest of the secure computation requires only constant work and communication. Each of the trial divisions require only constantly many invocations of sub-protocols, including π_{MOD} (and hence of π_{BOUND}) and all these sub-protocols require only a constant number of exponentiations. Thus, we conclude that the total costs that are incurred by the entire protocol are linear in the number of trial divisions. Further, all sub-protocols at every step of the full protocol may be run in parallel, hence round complexity is constant.

Biprimality Test. The main part of the biprimality test consists of the verification of the secure exponentiation of the random γ. Further, in the test of [BF01], the parties reach a negligible error probability by repeating the test ℓ times – as the test has one-sided error with probability at most $1/2$, it must be run e.g. 40 times in order to achieve an error of 2^{-40}. The most expensive part of the test is the execution of π_{EQ} as it is a cut and choose protocol, i.e. we need $O(\ell)$ exponentiations overall for each run, where ℓ is some statistical security parameter. However, as noted above this may be brought down to amortized constant overhead using the techniques of Cramer and Damgård [CD09]. Further, as the all ℓ tests may be run in parallel, round complexity is constant as well.

Secret-Key Shares. The generation of the multiplicative key shares requires constant overhead and constant round complexity. The communication/computation complexity is dominated by the multiple invocations of π_{BOUND} and π_{VERLIN} which obtain negligible soundness with constant overhead. We note that this protocol is executed only once.

Threshold Decryption. This protocol is dominated by the invocation of π_{EQ} which requires constant number of exponentiations for long enough challenge (see more discussion about this proof in Section 3.2, Item 5). For batch decryption the technique of Cramer and Damgård [CD09] can be used here as well.

6.3 Practical Considerations

To ease the security proof, we have taken a pessimistic approach above. Namely, zero-knowledge proofs have been applied at all stages in order to catch cheating players at once. However, a more optimistic approach allows for a more efficient protocol: All but one of our RSA-composite-generation attempts fail, and most of the zero-knowledge proofs are only needed for the successful modulus generation – hence they may be postponed. In addition to this, further optimizations for distributed RSA key generation are possible. We refer to Boneh and Franklin [BF01] for a list of general optimizations some of which are also applicable in our setting.

For the failing RSA-composite-generation attempts, we utilize the fact that the encryptions provided can be viewed as binding commitments. On failure, the parties reveal all random choices, thereby allowing the other party to verify their correct behavior by "executing" the protocol "in their head" and checking the correctness of the other party's messages, e.g. that plaintexts are appropriately bounded. Thus, overall efficiency of the many *failing* attempts will not be much more costly than twice that of failing attempts for the passively secure protocol. Once an attempt succeeds, ZK-proofs are used to ensure that this was correct. Slightly more formally, the key idea is that the simulator must know that the adversary is cheating (and that an honest party would detect this later, i.e. that the invocation should fail). We cannot simply postpone *all* proofs; care must be taken to allow simulation and to not reveal information that would allow a malicious party to, e.g., fake some zero-knowledge proof at a later point.

Generating the prime candidate. We may omit the invocations of π_{BOUND} on p_i and q_i, as this statement is implicitly shown by the invocations of π_{MOD} in the trial divisions. Further, verification may be postponed until we believe we have successfully generated an RSA-modulus; we cannot ensure correctness underway, but the encryption will be the same, thus, we still accept or reject as if we had run π_{BOUND} immediately.

Trial division. We may postpone the invocations of π_{MOD} at the cost of a few extra executions of simple proofs of knowledge, such as π_{ENC}. This ensures that the party knows its input, *and* that the simulator knows whether a later invocation of π_{MOD} may be successful (as it knows both what the input should be and what it actually is). If a trial division fails incorrectly, the honest party learns this when the corrupt party reveals its random choice, including its share of the random prime and the reduction modulo the trial-division-prime. If a trial division succeeds incorrectly, this can be discovered easily by performing the same trial-division on N – indeed at this point we may use a *larger* bound for the trial division as this can be performed very efficiently on the public N. The only remaining possibility is the case where the test should succeed, and did so despite one party providing an incorrect input. This case is handled by executing π_{MOD} for each trial division once the biprimality test succeeds, at which point the honest party will detect the incorrect behavior.

Computing and verifying the product. For the Paillier computation, we may postpone all checks except the proof that $\tilde{N}$ was the plaintext of the encryption supplied by P_1. Privacy of P_0 follows from the security of Paillier encryption, while privacy for P_1 follows from the fact the encryption sent back by P_1 only contains the desired result. Leakage from constructing a potentially incorrect value is eliminated by the eventual execution of the full zero-knowledge proofs. Alternatively, we may avoid verifying the product altogether. This may leak *a single bit of information*, namely whether some function on the shares of the honest party equals the still hidden modulus, N. Depending on the setting, this leakage may or may not be acceptable.

Biprimality test. The invocation of π_{EQ} can be postponed. If the test fails, the parties simply reveal their shares of the candidates; both parties may then verify that the other performed the exponentiations correctly.

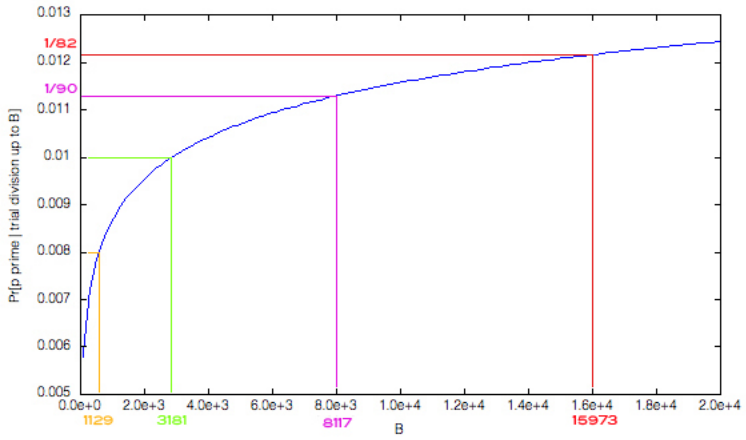

Figure 3: Probability plot with $n = 2048$.

On the other hand, if the test succeeds, the parties have either determined an RSA composite or one of them has cheated. They now execute π_{EQ} to determine which of the two is the case. Since the simulator knows the shares of the corrupt party, it is straightforward for it to check if the value supplied is the correct one.

7 Experimental Results

In this section we present an implementation of Protocol 1 with security against semi-honest adversaries. Our primary goal is to examine the overall time it takes to generate an RSA composite of length 2048 bits in case the parties do not deviate, and identify bottlenecks. The bulk of our implementation work is related to reducing the computational overhead of the trial division which is the most costly phase. To improve the overhead relative to the ElGamal PKE, we implement this scheme on elliptic curves using the MIRACL library [MIR] and use the elliptic curve *P-192* [FIP09] based on Ecrypt II yearly report [ECR11]; see [INI99, SEC00] for more practical information. A brief background on elliptic curves in found in Section 7.1.1. We further consider two optimizations detailed below.

7.1 Trial Division Implementation

As pointed out in Section 6.1, the trial division approach dramatically improves the complexity of the overall key generation process. Viewing the plot from figure 3 (that expresses the classic result of DeBruijn [DeB], analyzing the probability that a candidate integer is a prime given that it passes the trial division), one can easily observe the probability growth rate for finding a prime p while using larger thresholds. Specifically, for larger thresholds the number of comparisons increases, as the interval now covers more primes. Thus, the overall protocol is slowed down. In Section 7.2 we study some candidates for this threshold.

We further note that while implementing our protocols we noticed that more than 80% of the computational time is spent on the decryption process. This operation is required in Step 2b of Protocol 1 to test whether either $c^{(\alpha)}$ or $\tilde{c}^{(\alpha)}$ encrypt zero, for α a prime smaller than the trial division threshold B (namely, decryption is required at the end of each internal trial division check). Our implementations show that running the trial division in its naive form takes several hours. We therefore design two optimizations that reduce the running time of our protocol. It is straightforward to verify that the security of our protocol is not harmed by these modifications and we thus omit the security proofs. We begin with a short introduction on elliptic curve cryptography.

7.1.1 Background on Elliptic Curve Cryptography

In order to improve our performance we implement the ElGamal PKE over the group of points on an elliptic curve defined over a finite field. For an elementary introduction the reader is referred to [Kob87, KMWZ98] and for [Sil94, Sil09] for additional treatment. In the following we introduce some basics notations regarding elliptic curves; for simplicity we only introduce the case where the elliptic curve is defined over the real numbers $\mathbb{R}$.

Definition 7.1 *An elliptic curve over $\mathbb{R}$ is the set of points $x, y \in \mathbb{R}$ which satisfy the equation*

$$y^2 = x^3 + ax + b.$$

The set of points on an elliptic curve form an Abelian group. Moreover, the elliptic curve analogy of multiplying two elements in $\mathbb{G}$ is adding two points on $\mathbb{E}$, where $\mathbb{E}$ is an elliptic curve defined over $\mathbb{G}$. Thus, the analog of raising an element in $\mathbb{G}$ to the kth power is a multiplication of a point $P \in \mathbb{E}$ by an integer k. We now recall the ElGamal PKE over elliptic curves.

Distributed Key-Generation Protocol:

1. The parties agree on an elliptic curve E and a generator G.
2. Party P_0 chooses a random secret key x_{P_0} and sends P_1 the value $Q_{P_0} = x_{P_0}G$.
3. Party P_1 chooses a random secret key x_{P_1} and sends P_0 the value $Q_{P_1} = x_{P_1}G$.
4. The parties compute the shared key $Q = Q_{P_0} + Q_{P_1}$.

Encryption: Let $m \in \mathbb{G}$ be a message and let $\mathbb{E}$ be the set of points over the curve E. Then m must be mapped to a point in $\mathbb{E}$ before encrypted because addition over an elliptic curve is only possible with points on that curve. Thus integers have to be mapped to corresponding points through a mapping function $h : \mathbb{G} \to \mathbb{E}$. Moreover, this mapping should be homomorphic in the sense,

$$h(m_1 + m_2 + \cdots + m_n) = h(m_1) + h(m_2) + \cdots + h(m_n).$$

Thus, each integer $m \in \mathbb{G}$ is mapped to a curve point $M \in \mathbb{E}$ by computing $M = h(m) = mG$. This mapping function meets the required homomorphic property as,

$$M_1 + M_2 + \cdots + M_n = h(m_1 + m_2 + \cdots + m_n) = (m_1 + m_2 + \cdots + m_n)G = m_1G + m_2G + \cdots + m_nG.$$

Let q be the order of the group of points on the curve E, and let r be picked uniformly at random in $\mathbb{Z}_q$, then encrypting the point M is computed as follows:

$$(C_1, C_2) = (rG, M + rQ)$$

Distributed Decryption Protocol:

1. The parties wish to decrypt ciphertext $C = (C_1, C_2)$.
2. Party P_0 computes the secret share $D_0 = x_{P_0}C_1$ and sends it to party P_1.
3. Party P_1 computes the secret share $D_1 = x_{P_1}C_1$ and sends it to party P_0.
4. The parties compute the plaintext point $M = C_2 - D_1 - D_2$.

7.1.2 Optimization I: The BatchedDec Protocol

One way to improve the trial division performance is by reducing the number of decryption operations. The idea is quite simple, the parties do not decrypt $c^{(\alpha)}$ and $\tilde{c}^{(\alpha)}$ after each trial division iteration, but instead they maintain a state S for each T iterations specified by a pair of ciphertexts $(C_f, \tilde{C}_f)$. These ciphertexts encrypt the multiplication of the respective T values encrypted within $c^{(\alpha)}$'s and $\tilde{c}^{(\alpha)}$'s throughout the T iterations. More specifically, at the beginning of the first iteration the parties set $C_f = c^{(\alpha)}$ and $\tilde{C}_f = \tilde{c}^{(\alpha)}$. Recalling that p_0 and p_1 are the respective shares of a prime p, picked by P_0 and P_1, then for each iteration i the parties update C_f as follows (the update of $\tilde{C}_f$ is computed similarly):

1. Party P_0 selects a random value r'_i and computes $R'_i = r'_i G$ and $K'_i = r'_i Q$, where Q is the public-key.
2. P_0 computes $V_{P_0 i} = (p_0 \bmod \alpha_i) C_{f_{i-1}}$. Let $V_{P_0 i} = (C_1, C_2)$, then P_0 computes $H_{P_0 i} = (C_1 + R'_i, C_2 + K'_i)$ and sends $H_{P_0 i}$ to P_1.
3. P_1 computes $H_{P_1 i}$ similarly, selecting a random value r'' and using $p_1 \bmod \alpha_i$. P_1 sends $H_{P_1 i}$ to P_0.
4. The parties compute $H = H_{P_0} + H_{P_1}$ and set $C_f = H$.

Finally, at the Tth iteration the parties decrypt and check whether one of the two ciphertexts is an encryption of zero. Based on the homomorphic properties of our PKE it holds that either C_f or $\tilde{C}_f$ are the encryption of 0 if at least one of the α's tested during one of the T iterations divided the parties' candidate. This approach enables to perform the decryption operation only once every T iterations. Clearly, if T is too small the software does not get a big performance improvement. On the other hand, when T is too big the software will have to perform too many rounds before discovering that much of the work is useless. After some empirical studies we observed that $T = 100$ is a good threshold candidate.

Correctness (informally). We begin from the first iteration of the new trial division protocol. For simplicity, we only address the second iteration. The general case easily generalizes.

- First iteration: Let α_1 be the prime checked during the first iteration, then the parties initialize the status $S = (C_{f_1}, \tilde{C}_{f_1})$ as follows:

  $S = (C_{f_1}, \tilde{C}_{f_1}) = (c^{(\alpha_1)}, \tilde{c}^{(\alpha_1)})$

  Where:

  $$c^{(\alpha_1)} = \mathsf{Enc}_{pk_{\mathrm{EG}}}(p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1) = (r_1 G, r_1 Q + (p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1) G).$$

  and

  $$\tilde{c}^{(\alpha_1)} = \mathsf{Enc}_{pk_{\mathrm{EG}}}(p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1 - \alpha_1) = (r_{\tilde{c}} G, r_{\tilde{c}} Q + (p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1 - \alpha_1) G).$$

- Second iteration: Using the above notations, and keeping in mind that $V_{P_0 2} = (v_1, v_2)$, P_0 computes

  $$H_{P_0 2} = (v_1 + R'_2, v_2 + K'_2)$$

  and sends it to P_1. Thus remembering that $C_{f_1} = (c_1, c_2)$ we have:

  $$\begin{aligned} H_{P_0 2} &= \big((p_0 \bmod \alpha_2) c_1 + R'_2, (p_0 \bmod \alpha_2) c_2 + K'_2\big) \\ &= \big((p_0 \bmod \alpha_2) r_1 G + r'_2 G, (p_0 \bmod \alpha_2)(r_1 Q + (p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1) G + r'_2 Q)\big) \\ &= \big(((p_0 \bmod \alpha_2) r_1 + r'_2) G, ((p_0 \bmod \alpha_2) r_1 + r'_2) Q + (p_0 \bmod \alpha_2)(p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1) G\big). \end{aligned}$$

In addition, party P_1 computes,

$$H_{P_1 2} = \left(((p_1 \bmod \alpha_2)r_1 + r_2'')G, ((p_1 \bmod \alpha_2)r_1 + r_2'')Q + (p_1 \bmod \alpha_2)(p_0 \bmod \alpha_1 + p_1 \bmod \alpha_1)G\right).$$

The parties then exchange $H_{P_0 2}$ and $H_{P_1 2}$ and compute $C_{f_2} = H_{P_0 2} + H_{P_1 2}$. It is now trivial to see that if C_{f_2} is an encryption of 0 then α_1, α_2 or both divide the candidate. The demonstration is analogous for $\tilde{C}_{f_2}$. This optimization enables to maintain the same level of security while improving performance. Indeed looking at table 2 the expected running time of the parties is drops into 47 minutes.

7.1.3 Optimization II: The LocalTrialDiv Protocol

In this section we take a different approach and instruct the parties to "abort" the distributed trial division after a number of checks, continue with the computation of the composite candidate, and only then complete the trial division test on their own (i.e., without communication). More specifically,

1. For the first X rounds the parties run the trial division as explained in the previous section.
2. The parties then quit the trial division loop and compute Step 3a of Protocol 1, computing $\tilde{N}$.
3. The parities then locally complete the trial division for $\tilde{N}$ until the threshold B is reached. Namely, by checking whether $\alpha | \tilde{N}$ or not.

Notably, it has been observed that most of the candidates are rejected at very early stages of the trial division, namely, before α turns to be higher than 31. The main advantage of this solution is that it enables to avoid the usage of cryptographic routines for α's greater than X. This optimization, denoted by *LocalTrialDiv*, improves the trial division performance even further; our experimental results are depicted in table 3. In our experiments we fix X to be 11.

7.2 Performance Results

In order to illustrate the efficiency of our protocols we created a benchmark program that simulates the behavior of the parties in three different cases: (**1**) the original protocol, (**2**) the *LocalTrialDiv* protocol and (**3**) the *BatchedDec* protocol. In this section we present the performance measures we have observed during our experiments. These measurements have been computed by running the programs over an Intel Core i5 dual core 2.3 GHz, with 256KB for L2 cache per core, 3 MB for L3 cache, and 8GB of Ram.

7.2.1 Performance Analysis of the Original Protocol

In order to estimate the resources needed to output a legal RSA composite following the *original* protocol, we let the parties run 10 trial division tests and calculate the average time the parties spent on a single test. This value is then multiplied by the overall expected number of iterations of the trial division (specified by the DeBruijn formula; see figure 3). Observing the results presented in Table 1, one can see that the expected running time is about 3 hours already for a relatively small threshold value. In the following sections we present the performance measurements of our two optimized protocols.

Round #	B = 15973	B = 8117	B = 3181	B = 1129
1	49.30 seconds	16.11 seconds	11.96 seconds	4.18 seconds
2	48.99 seconds	15.16 seconds	6.60 seconds	3.37 seconds
3	47.33 seconds	17.07 seconds	6.32 seconds	3.76 seconds
4	41.23 seconds	13.74 seconds	7.81 seconds	6.64 seconds
5	48.30 seconds	16.08 seconds	7.74 seconds	2.57 seconds
6	54.68 seconds	15.56 seconds	7.15 seconds	2.83 seconds
7	41.38 seconds	15.44 seconds	9.53 seconds	3.98 seconds
8	46.46 seconds	15.25 seconds	6.40 seconds	3.27 seconds
9	41.90 seconds	15.06 seconds	12.35 seconds	2.87 seconds
10	45.56 seconds	14.40 seconds	6.55 seconds	4.03 seconds
Average:	**46.51 seconds**	**15.38 seconds**	**8.24 seconds**	**3.75 seconds**
Iterations Expected:	1804	1980	2200	2750
Total Time Expected:	**23.3 hours**	**8.5 hours**	**5.0 hours**	**2.9 hours**

Table 1: Experimental results for the *original* protocol.

7.2.2 Performance Analysis of the BatchedDec Protocol

In table 2 we present our performance analysis of protocol *BatchedDec*. Each row reports the duration (in minutes) of a complete execution for different selected values of B. We take into account the set of all operations performed by the parties during *BatchedDec* protocol, i.e., from the initialization phase of the libraries and the key generation steps, to the actual output of the RSA composite. We report the average time it takes to generate a valid composite, as well as the average number of iterations required throughout the entire computation, until a valid RSA composite is output.

Observing this data, the reader can see that the theoretical results reported in the previous sections are partly confirmed. Indeed, the cost of the trial division decreases along with the threshold values for B, yet the number of rounds increases. Using $B = 1129$ the parties need more CPU time on the average than for threshold $B = 3181$. This is because the improvement due to using a decreased value of B cannot "compete" with the resources needed to perform the increased number of rounds. Indeed, the former threshold doubles the expected number of rounds. Moreover, increasing the threshold value even further does not improve the performance.

7.2.3 Performance Analysis of the LocalTrialDiv Protocol

Finally, given the results from table 3, one can observe that a similar scenario described with respect to Protocol *LocalTrialDiv* as well. Indeed, setting $B = 8117$, the average CPU time required in order to compute an RSA composite is 37 minutes. Increasing the value of B into 15973 makes the CPU time drop into 15 minutes on the average, yet increasing B even more increases the CPU time beyond that. It can be seen that the values chosen for B are a bit higher than the values chosen for the previous approach; this is because this method is much faster. Thus, the parties are able to handle more trial divisions per round before executing a biprimality test.

Run #	B = 15973	B = 8117	B = 3181	B = 1129
1	164.27 minutes	12.27 minutes	18.28 minutes	13.08 minutes
2	55.18 minutes	66.57 minutes	68.63 minutes	73.80 minutes
3	1.07 minutes	319.30 minutes	69.42 minutes	15.43 minutes
4	29.23 minutes	41.00 minutes	13.63 minutes	85.38 minutes
5	46.48 minutes	79.88 minutes	4.68 minutes	26.63 minutes
6	17.47 minutes	84.38 minutes	28.55 minutes	29.68 minutes
7	79.88 minutes	81.92 minutes	147.75 minutes	213.27 minutes
8	262.72 minutes	46.68 minutes	17.30 minutes	18.55 minutes
9	344.68 minutes	14.25 minutes	74.42 minutes	3.65 minutes
10	531.68 minutes	32.05 minutes	34.73 minutes	91.55 minutes
Iterations Observed: (Average)	**1183**	**1850**	**1899**	**4470**
Total Time Observed: (Average)	**153.27 minutes**	**77.82 minutes**	**47.73 minutes**	**57.10 minutes**

Table 2: Experimental results for the *BatchedDec* protocol.

Run #	B = 65521	B = 15973	B = 8117
1	27.75 minutes	4.93 minutes	34.37 minutes
2	5.10 minutes	7.70 minutes	20.07 minutes
3	4.58 minutes	23.92 minutes	1.60 minutes
4	11.00 minutes	28.18 minutes	6.2 minutes
5	50.10 minutes	27.05 minutes	7.99 minutes
6	3.13 minutes	4.22 minutes	49.50 minutes
7	2.28 minutes	37.53 minutes	15.91 minutes
8	17.17 minutes	1.80 minutes	16.48 minutes
9	42.07 minutes	13.55 minutes	196.28 minutes
10	22.33 minutes	1.18 minutes	22.12 minutes
Iterations Observed: (Average)	1144	1190	3418
Total Time Observed: (Average)	**18.55 minutes**	**15.00 minutes**	**37.05 minutes**

Table 3: Experimental results for the *LocalTrialDiv* protocol.

References

[ACS02] J. Algesheimer, J. Camenisch, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In M. Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 2002.

[BB89] J. Bar-Ilan and D. Beaver. Non-cryptographic fault-tolerant computing in a constant number of rounds of interaction. In Piotr Rudnicki, editor, *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 201–209, New York, 1989. ACM Press.

[BBBG98] S. Blackburn, S. Blake-Wilson, M. Burmester, and S. Galbraith. Shared generation of shared RSA keys. http://cacr.math.uwaterloo.ca/techreports/1998/corr98-19.ps, 1998.

[BDOZ11a] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.

[BDOZ11b] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *EUROCRYPT*, pages 169–188, 2011.

[BF01] D. Boneh and M. K. Franklin. Efficient generation of shared RSA keys. *J. ACM*, 48(4):702–722, 2001.

[BFP+01] O. Baudron, P. A. Fouque, D. Pointcheval, G. Poupard, and J. Stern. Practical multi-candidate election system. In *In PODC*, pages 274–283. ACM Press, 2001.

[Bou00] F. Boudot. Efficient proofs that a committed number lies in an interval. In B. Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 431–444. Springer, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CD09] R. Cramer and I. Damgård. On the amortized complexity of zero-knowledge protocols. In *CRYPTO*, pages 177–191, 2009.

[CDN01] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, pages 280–299, 2001.

[CGHN01] D. Catalano, R. Gennaro, N. Howgrave-Graham, and P. Q. Nguyen. Paillier's cryptosystem revisited. In *ACM Conference on Computer and Communications Security*, pages 206–214, 2001.

[CGS97] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *EUROCRYPT*, pages 103–118, 1997.

[CKY09] J. Camenisch, A. Kiayias, and M. Yung. On the portability of generalized schnorr proofs. In *EUROCRYPT 2009*, pages 425–442, 2009.

[Cle86] R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, pages 364–369, 1986.

[Coc97] C. Cocks. Split generation of RSA parameters with multiple participants. In *Proceedings of 6th IMA conference on Cryptography and Coding*, pages 200–212. LNCS 1355, 1997.

[Cop97] D. Coppersmith. Small Exponents to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10:233–260, 1997.

[CP92] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *CRYPTO*, pages 89–105, 1992.

[DeB] N. DeBruijn. On the number of uncanceled elements in the sieve of eratosthenes. In *In* Proc. Neder. Akad. Wetensh.*, (53),*, pages 803–812. (Reviewed in LeVeque Reviews in Number Theory, 4, N-28, page 221).

[Des94] Y. G. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–457, July 1994.

[DF02] I. Damgård and E. Fujisaki. A statistically-hiding integer commitment scheme based on groups with hidden order. In *ASIACRYPT*, pages 125–142, 2002.

[DH76] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[DJ01] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography*, pages 119–136, 2001.

[DJ02] I. Damgård and M. Jurik. Client/server tradeoffs for online elections. In *Public Key Cryptography*, pages 125–140, 2002.

[DM10] I. Damgård and G. L. Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In *TCC*, pages 183–200, 2010.

[DN02] I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In *CRYPTO*, pages 581–596, 2002.

[DN03] I. Damgård and J. B. Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO*, pages 247–264, 2003.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, pages 643–662, 2012.

[ECR11] Ecrypt ii, yearly report on algorithms and keysizes (2010). http://www.ecrypt.eu.org/documents, 2011.

[ElG85] T. ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Trans. Info. Theory*, IT 31:469–472, 1985.

[FFS88] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *J. Cryptology*, 1(2):77–94, 1988.

[FIP09] National institute of standards and technology, federal information processing standards: Digital signature standard. http://csrc.nist.gov/encryption, 2009.

[FMY98] Y. Frankel, P. D. Mackenzie, and M. Yung. Robust efficient distributed RSA-key generation. In *stoc98*, pages 663–672. ACM Press, 1998.

[FO97] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *CRYPTO*, pages 16–30, 1997.

[FPS00] P.A. Fouque, G. Poupard, and J. Stern. Decryption in the context of voting or lotteries. In *Financial Crypto '00*. Springer-Verlag, 2000.

[FS86] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *CRYPTO*, pages 186–194, 1986.

[Gil99] N. Gilboa. Two party RSA key generation. In *CRYPTO*, pages 116–129, 1999.

[GJKR01] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Robust Threshold DSS Signatures. *Information and Computation*, 164(1):54–84, 2001.

[GJKR07] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[GKR00] R. Gennaro, H. Krawczyk, and T. Rabin. Robust and Efficient Sharing of RSA Functions. *Journal of Cryptology*, 13(2):273–300, 2000.

[Gol04] O. Goldreich. *Foundations of Cryptography: Volume 2*. Cambridge University Press, 2004. Preliminary version *http://philby.ucsd.edu/cryptolib.html/*.

[INI99] National institute of standards and technology, recommended elliptic curves for federal government use. http://csrc.nist.gov/encryption, 1999.

[JL09] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *TCC*, pages 577–594, 2009.

[JS07] S. Jarecki and V. Shmatikov. Efficient two-party secure computation on committed inputs. In *EUROCRYPT*, pages 97–114, 2007.

[KMWZ98] Neal Koblitz, Alfred J. Menezes, Yi-Hong Wu, and Robert J. Zuccherato. *Algebraic aspects of cryptography*. Springer-Verlag New York, Inc., New York, NY, USA, 1998.

[Kob87] Neal Koblitz. *A course in number theory and cryptography*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO (2)*, pages 1–17, 2013.

[Lip03] H. Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. In *ASIACRYPT*, pages 398–415, 2003.

[LP11] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *TCC*, pages 329–346, 2011.

[MIR] Multiprecision integer and rational arithmetic c/c++ library. http://www.shamus.ie/.

[NS10] Takashi Nishide and Kouichi Sakurai. Distributed paillier cryptosystem without trusted dealer. In *WISA*, pages 44–60, 2010.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[PS98] G. Poupard and J. Stern. Generation of shared RSA keys by two parties. In *in Asiacrypt 98*, pages 11–24. Springer-Verlag, 1998.

[Rab80] M. O. Rabin. Probabilistic Algorithm for Testing Primality. *Journal of Number Theory*, 12:128–138, 1980.

[Rab81] M. O. Rabin. How To exchange Secrets with Oblivious Transfer. Techincal Report TR-81, Aiken Computation Lab, Harvard University, 1981.

[Rab98] T. Rabin. A simplified approach to threshold and proactive rsa. In *CRYPTO*, pages 89–104, 1998.

[Sch91] C. P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4:161–174, 1991.

[SEC00] Standards for efficient cryptography group, sec 2: Recommended elliptic curve domain parameters. SECG2, 2000.

[Sho00] V. Shoup. Practical threshold signatures. In *EUROCRYPT*, pages 207–220, 2000.

[Sil94] J.H. Silverman. *Advanced Topics in the Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer-Verlag, 1994.

[Sil09] J.H. Silverman. *The Arithmetic of Elliptic Curves*. Graduate Texts in Mathematics. Springer, 2009.

A Distributed Generation of an RSA Composite [PS98]

In [PS98], Poupard and Stern suggested a protocol that enables two parties to generate an RSA composite without the help of a trusted dealer. In particular, they showed how to evaluate any algebraic expression as follows: Let λ_0 and λ_1 denote the respective inputs of P_0 and P_1 sampled out of a *polynomially bounded* domain Λ, and let f denotes the public function that the parties wish to compute. Then, party P_0 chooses two random values α, β from a predetermined set and computes $\alpha \cdot f(\lambda_0, \lambda) + \beta$ for all $\lambda \in \Lambda$. Next, the parties run an oblivious transfer protocol [Rab81] where P_0 enters the above set and P_1 enters its input λ_1 to f. Upon completing this execution, P_1 learns $y_1 = \alpha \cdot f(\lambda_0, \lambda_1) + \beta$. The parties repeat the above with reversed roles. Denote by $y_0 = \alpha' \cdot f(\lambda_0, \lambda_1) + \beta'$, P_0's output from the later oblivious transfer execution. Then, in the final step, the parties reveal (α, β, y_0) and (α', β', y_1) *simultaneously* and verify their outputs.

This protocol can be utilized to compute $N = (p_0 + p_1)(q_0 + q_1)$, where party P_0 enters two random shares denoted by p_0, q_0 and P_1 enters its random shares, p_1, q_1. Nevertheless, the fact that it incurs linear costs in the size of the domain makes it impractical for super polynomial domains, as in most cases. In order to circumvent this problem, Poupard and Stern introduce the following solution. The parties agree first on M, as the smallest product of prime numbers greater than 2^n and compute the function in $\mathbb{Z}_M$. Now, since M can be factored into smaller relatively prime factors, the parties run the protocol for each prime factor m_i of M and then use the Chinese Remainder Theorem to combine these outcomes into the desired computation modulus M. Apart from the fact that the paper does not introduce a complete simulation based proof, this technique induces a problem when coping with malicious activity as the protocol does not have a mechanism which verifies that the parties use consistent inputs either for the multiple oblivious transfer executions, or even within a single invocation of the original construction. In particular, it is not clear how to build such a practical mechanism. In addition, the efficiency of the later protocol still relies on the number of integers in the primes factorization of M and the sizes of the domains that they induce.

B The [DM10] Biprimality Test: The Two-Party Case

This section describes how the biprimality test by Damgård and Mikkelsen [DM10] works, and how it is applicable in the two-party case in the honest-but-curious setting. It is currently unknown how to adapt the test to the malicious case without applying generic and rather inefficient zero knowledge proofs. As with the previous biprimality test the objective is to test a number $N = pq$, with $p \equiv q \equiv 3 \pmod 4$ to check whether p and q are both primes. We assume N to be public and p and q to be additively shared between the two parties. The test is basically the Miller Rabin [Rab80] primaility test, run first on p and afterward on q, exploiting the fact that N is public to do this efficient. We also assume a distributed ElGamal scheme as described in section 2.2.2 is set up, and that P_0 has generated a paillier key pair.

First we note that using the Miller Rabin [Rab80] primaility test on a number $p \equiv 3 \pmod 4$, all that is needed to test is whether $\gamma^{(p-1)/2} \equiv \pm 1 \pmod p$ for a randomly picked $\gamma \in \mathbb{Z}_p^*$. Next we note that it does not make a difference if we randomly pick $\gamma \in \mathbb{Z}_N^*$. Focusing on p, the test verifies that $\gamma^{(p-1)/2} \equiv \pm 1 \bmod p$ for a randomly picked $\gamma \in \mathbb{Z}_N^*$. First party P_i computes β_i:

$$\beta_0 = \gamma^{(p_0-1)/2} \bmod N, \qquad \beta_1 = \gamma^{-p_1/2} \bmod N.$$

We note that:

$$\gamma^{(p-1)/2} \equiv \pm 1 \pmod p \tag{2}$$

$$\Updownarrow$$

$$\gamma^{(p_0-1)/2} \equiv \pm\gamma^{-p_1/2} \pmod p \tag{3}$$

$$\Updownarrow$$

$$(\beta_0 \bmod p) = \pm\,(\beta_1 \bmod p) \tag{4}$$

where (3) to (4) follows from the fact that $p|N$; thus reducing β_i modulo p is equivalent to reducing $\gamma^{(p_i-(1-i))/2}$ modulo p directly. It remains to be shown how to perform the modulo reductions and check for equality in (4) securely.

Modulo Reductions an Equality Testing in (4): We first note, that: $\beta_i \bmod p = \beta_i - \left\lfloor \frac{\beta_i}{p} \right\rfloor p$, and we assume an approximation $a^{(i)}$ of $\lfloor \beta_i/p \rfloor$, where $a^{(i)}$ is held additively shared, $a_0^{(i)} + a_1^{(i)}$; we show how to obtain this below. The rest of the calculation is done distributedly as: P_0 sends ElGamal encryptions of β_0,

p_0, $a_0^{(0)}$, $a_0^{(1)}$, $p_0 \cdot a_0^{(0)}$, and $p_0 \cdot a_0^{(1)}$ to P_1. Using the homomorphic property of the ElGamal scheme this allows P_1 to compute encryptions of:

$$\begin{aligned}\beta_i - (p_0 \cdot a_0^{(i)} + p_0 \cdot a_1^{(i)} + p_1 \cdot a_0^{(i)} + p_1 \cdot a_1^{(i)}) &= \beta_i - (a_0^{(i)} + a_1^{(i)})(p_0 + p_1) \\ &= \beta_i - \left\lfloor \frac{\beta_i}{p} \right\rfloor \cdot p \\ &= (\beta_i \bmod p) + k_i p\end{aligned}$$

for $i \in \{0, 1\}$, where the k_i's are small integer values due to the fact that the $a^{(i)}$ are only approximations.

If (and only if) $\beta_0 \equiv \beta_1 \bmod p$ (respectively $\beta_0 \equiv -\beta_1 \bmod p$), the difference (respectively sum) will be a *small* multiple of p. Hence, we may check whether $\beta_0 \equiv \pm\beta_1 \bmod p$ using a small number of equality tests ($\beta_0 + \beta_1 = 0$, $(\beta_0 + \beta_1) - p = 0$, $(\beta_0 + \beta_1) - 2p = 0$, etc): First the parties randomly permute the ciphertexts in question; then they raise each one to a random, non-zero exponent; finally the parties decrypt each cipher text – if one was an encryption of 0, they conclude that $\beta_0 \equiv \pm\beta_1 \bmod p$. This leaks no information due to the (secret) permutation and random exponentiations.

Computing $a^{(i)} \approx \lfloor \beta_i/p \rfloor$: The approximation, $a^{(i)}$, of $\lfloor \beta_i/p \rfloor$ is obtained by first computing

$$\tilde{a}^{(i)} = \left\lfloor \frac{2^m}{N} \right\rfloor \cdot (q_0 + q_1) \cdot \beta_i \approx \left\lfloor \frac{2^m \cdot (q_0 + q_1) \cdot \beta_i}{N} \right\rfloor = \left\lfloor \frac{2^m \cdot \beta_i}{p} \right\rfloor$$

which is an approximation of $2^m a^{(i)}$, where m is a bit-length ensuring that $a^{(i)}$ will be sufficiently accurate, making k, the number of tests, above sufficiently small. For a thoroughly analysis of the size of m, the reader is referred to [DM10]. An additive sharing of $\tilde{a}^{(i)}$ over the integers may be computed based on Paillier encryption with P_0's keys. The Paillier key, has to be sufficiently big, such that the following calculations, will not lead to an overflow modulo the modulus. This implies that the following calculations are done over the integers, although they are done modulo the modulus of the Paillier key.

The value $\lfloor \frac{2^m}{N} \rfloor$ is a public value, and the rest is integer computation, thus: P_0 sends encryptions of q_0, β_0 and $q_0 \cdot \beta_0$ to P_1, which for $i \in \{0, 1\}$ computes encryptions of the values:

$$\begin{aligned}\tilde{a}^{(i)} &= \left\lfloor \frac{2^m}{N} \right\rfloor \cdot (q_0\beta_i + q_1\beta_i) \\ &= \left\lfloor \frac{2^m}{N} \right\rfloor \cdot q \cdot \beta_i.\end{aligned}$$

Finally P_1 picks two uniformly random values $\tilde{a}_1^{(i)}$ κ bits longer than the $\tilde{a}^{(i)}$, where κ is a security parameter, and returns and encryptions of

$$\tilde{a}_0^{(i)} = \tilde{a}^{(i)} - \tilde{a}_1^{(i)}$$

to P_0 for $i \in \{0, 1\}$. P_0 decrypts and stores both $\tilde{a}_0^{(i)}$ as negative integers, i.e. views $N_{\mathrm{Pa}} - x \in \mathbb{Z}_{N_{\mathrm{Pa}}}$ as $-x \in \mathbb{Z}$, N_{Pa} denoting the modulus in the Paillier key. The parties then truncate their values (drop the least significant m bits); denote the truncated values $a_0^{(i)}$ and $a_1^{(i)}$, and note that their sum is the required approximation, $a^{(i)}$:

$$\begin{aligned}a_0^{(i)} + a_1^{(i)} &\approx \tilde{a}^{(i)}/2^m \\ &\approx \left\lfloor \frac{2^m \cdot \beta_i \cdot q}{2^m N} \right\rfloor \\ &\approx \lfloor \beta_i/p \rfloor .\end{aligned}$$

C Generalizing to the Multiparty Case

Our protocol may be generalized to the multiparty case in the setting with dishonest majority and a static adversary. Our construction is comprised of the following two parts: (1) securely determining the modulus N, and (2) obtaining a threshold Paillier key. To the best of our knowledge, this is the first multiparty protocol for a distributed RSA composite generation, which is actively secure against an adversary corrupting all but one of the parties. Moreover, despite Cramer et al. [CDN01] and Damgård and Nielsen [DN03] demonstrating efficient, general multiparty computation given only a public Paillier key with a shared secret key (provided by some third-party, which could be replaced by a secure protocol), previous papers on RSA composite generation have disregarded the shared Paillier decryption key and focused solely on generating the modulus. We are the first to present a protocol for generating a full, distributed Damgård-Jurik key[3] [DJ01]. For clarity, we take a high-level view of these protocols. Further, as the protocols are described with clarity, rather than efficiency, in mind, many straightfoward optimizations are possible.

In the following, let k denote the number of parties, $P_1, \ldots, P_k$, and let t denote the desired threshold for the shared Paillier key computation. For both parts, we ensure security against an adversary corrupting up to $t-1$ parties, though naturally, given the threshold key any t parties may perform a decryption or reconstruct the secret key.

A few remarks are in place here. We first assume that the parties have access to PKI setup. That is, each party has a public verification key which all parties hold a copy of. Based on this, we may construct a broadcast channel. In addition, we allow any party to halt the protocol at any time by broadcasting an abort message. When this occurs, all parties broadcast all messages of the entire protocol (including all signatures) as well as all randomness used in all key generations, encryptions, etc. At this point it is easy to verify the behavior of all parties and assign blame. We may do this as there are only random inputs to the protocol picked by the participants, i.e. leaking them does not compromise any privacy.[4]

We remark that care must be taken when running sub-protocols in parallel. For example, a rushing adversary could potentially break the entire protocol by using a ZK-proof of an honest party to fake a ZK-proof of its own. The issues can be removed, e.g. by using broadcasts to ensure complete synchronization, and having all parties commit to all messages in a given round before actually sending them.

C.1 Generating an RSA Composite

In this section we describe our protocol for generating as RSA composite $N = \left(\sum_{i=1}^{k} p_i\right)\left(\sum_{i=1}^{k} q_i\right)$, for p_i, q_i being the shares picked by the ith party. We note that most of the steps of our multiparty protocol translate directly from our two-party construction.

1. KEY-SETUP.
   - The parties run the k-party generalization of the shared ElGamal key generation: The parties agree on the group of sufficiently large order, Q, and a generator, g. Each party P_i picks a uniformly random $x_i \in \mathbb{Z}_Q$, broadcasts $h_i = g^{x_i}$, and proves knowledge of a DL of h_i to all others. The ElGamal key is now $h = \prod_{i=1}^{k} h_i$. Decryption is analogous to the two-party case.
   - Each party P_i generates and broadcasts a Paillier key, $N_i \gg Q^2$, and proves that it is well-formed to all other parties.
2. GENERATE CANDIDATES.

[3] Our key differs slightly, however, the principles and the construction are essentially the same.

[4] See Footnote 2.

- We employ the same idea as used by Boneh and Franklin, [BF01]: Each party P_i generates a random share p_i for $p = \sum_{i=1}^{k} p_i$. The parties broadcast ElGamal encryptions of their shares and prove to all others that they know the plaintexts and that these are of appropriately bounded size using ZK proofs π_{ENC} and π_{BOUND}. Similarly to above, the parties ensure that the share of P_1 is congruent to $3 \bmod 4$, while all others are congruent to 0.
- Trial division is analogous to the two-party protocol. The parties broadcast encryptions of $p_i \bmod \alpha$ for all primes $\alpha < B$. Using the homomorphic property, the parties compute an encryption of the sum, and check if this is an encryption of a multiple of α, i.e. whether it is one of the values within $[0, \alpha, 2\alpha, \ldots, (k-1)\alpha]$. Correct behavior is verified with π_{MOD}.
- Repeat the previous steps to generate the second candidate, q, as well.

3. COMPUTE PRODUCT ($N = pq$).

   - This step differs significantly; obtaining g^N could easily be done [CDN01], however, $\tilde{N}$ cannot be computed as in the two-party case, hence we would have to obtain the solution to the DL problem differently. The main idea here is to have every pair of parties, P_i and P_j, engage in a protocol to obtain an additive sharing over $\mathbb{Z}_Q$ of $p_i q_j = s_{i,j}^{(i)} + s_{i,j}^{(j)}$ instead.[5] Note that no other party receives shares of this value. Each party then locally adds all shares held, thereby obtaining a share from an additive sharing of

$$pq = \sum_{i=1}^{k} \sum_{j=1}^{k} p_i q_j = \sum_{i=1}^{k} \sum_{j=1}^{k} \left(s_{i,j}^{(i)} + s_{i,j}^{(j)} \right). \tag{5}$$

     Additionally, ZK proofs on ElGamal encryptions are used to ensure that parties are committed to their shares and behave as specified. More formally,

     – For $1 \leq i, j \leq k$, P_j broadcasts an ElGamal encryption of a uniformly random value, $-s_{i,j}^{(j)} \in \mathbb{Z}_Q$, and proves in ZK towards all others that it has known plaintext using π_{ENC}.
     – For $1 \leq i, j \leq k$, P_i sends a Paillier encryption $c_{p_i,j}$ of p_i to P_j under its own key, N_i.[6] Moreover, for each one, it proves plaintext knowledge using π_{ENC} and that the value is bounded using π_{BOUND}.
     – For $1 \leq i, j \leq k$, P_j computes a Paillier encryption under key N_i

$$\bar{c}_{i,j} = (c_{p_i,j})^{q_j} \cdot \mathsf{Enc}\left(-s_{i,j}^{(j)} + Q \cdot r_{i,j} \right),$$

       where $r_{i,j}$ is a uniformly random n-bit value which statistically masks any overflow modulo Q in the computation. $\bar{c}_{i,j}$ is then sent to P_i along with a ZK proof that the computation was done using known values. This can e.g. be done using π_{VERLIN} and an additional dummy encryption of 0. It is straightforward to construct a simpler protocol similar to π_{VERLIN}; naturally this would be more efficient.
     – For $1 \leq i, j \leq k$, P_i decrypts $\bar{c}_{i,j}$, reduces the resulting plaintext modulo Q, and denotes the result $s_{i,j}^{(i)}$. P_i then broadcasts an ElGamal encryption of $s_{i,j}^{(i)}$ and proves plaintext knowledge, π_{ENC}.
     – For $1 \leq i, j \leq k$, the parties verify that everyone is indeed committed to shares of the products, $p_i q_j$. Based on the encryption of q_j, party P_i computes and broadcasts a fresh

[5] For $i = j$, the party in question simply computes a dummy sharing of the known value $p_i q_j$.

[6] For efficiency, P_i may send the same encryption to all other parties; we do not demand this behavior, though.

encryption of $p_i q_j$; correctness is verified by executing π_{MULT}. All parties then compute an ElGamal encryption of

$$p_i q_j - \left(s_{i,j}^{(i)} + s_{i,j}^{(j)} \right), \tag{6}$$

using the homomorphic property. This encryption is then decrypted, and all parties verify that the obtained plaintext equals zero. This demonstrates that $s_{i,j}^{(i)} + s_{i,j}^{(j)} = p_i q_j$, i.e. that the sharing was indeed of the product. If any check fails, the parties abort the entire execution.

– For $1 \leq i \leq k$, all parties compute an ElGamal encryption of

$$s_i = \sum_{j=1}^{k} s_{i,j}^{(i)} + s_{j,i}^{(i)}$$

using the homomorphic property. P_i the broadcasts s_i, the parties decrypt the encryption of s_i (resulting in the value g^{s_i}), and finally verify that the broadcast value is correct, i.e. the DL of the decrypted value.

– The parties compute $N = \sum_{i=1}^{k} s_i \bmod Q$.

4. BIPRIMALITY TEST.

- P_1 has the share congruent to $3 \bmod 4$, and therefore behaves as P_0 in the two-party protocol, while the rest behave as P_1 in the two-party case. Each party broadcasts its γ_i; P_1 proves consistency towards an encryption of $(N - p_1 - q_1 + 1)/4$ using π_{EQ}, while P_i proves consistency towards $-(p_i + q_i)/4$ for $1 < i \leq k$.

Correctness: Except for the computation of $N = pq$, all steps are essentially the same as in the two-party protocol. Focusing solely on this step, we note that it is easily verified that the right result is obtained, by Equation (5), since all parties explicitly verify that the encrypted shares indeed sum to the products, $p_i q_j$.

Security: Again, as all steps are analogous to the two-party protocol except for the computation of N, the security of these is shown (essentially) in the same way as security for the two-party protocol. Regarding the computation of N, we must ensure that no party can deviate from the protocol in any way without being detected. The computation of the additive sharing of N based on the parties' Paillier keys can be viewed as a number of two-party computations (of additive secret sharings of products), which must be globally verifiable. This is achieved, since it is verified using Equation (6) that the sum of the encrypted shares of the product *equals* the encrypted product; the latter is guaranteed to be correct due to the use of π_{MULT}. Thus, even if both P_i and P_j are corrupt, they will be committed to a sharing of the product. Further, a corrupt P_i can obtain no information about q_j as the addition of $-s_{i,j}^{(j)} + Q \cdot r_{i,j}$ statistically masks any information.[7] A corrupt P_j on the other hand learns nothing about p_i, as it only sees semantically secure encryptions. Similarly, if both P_i and P_j are honest, then the attacker only sees the semantically secure encryptions transferred, which leaks no information. Formal simulation is possible by giving the adversary either fresh, random encryptions *or* encryptions of random values distributed as the statistical mask depending on whether it knows the secret key.

[7] Note that the application of π_{BOUND} on ciphertext $c_{p_i,j}$ is critical, as this guarantees an honest P_j that its masking will hide q_j.

C.2 Computing the Threshold Key

In this section we present a protocol for generating a threshold key for (a slight variation of) the Damgård-Jurik generalization of Paillier encryption [DJ01]. Recall that decryption consists of raising to the power of d, where

$$d \equiv \begin{cases} 0 \bmod \phi(N) \\ 1 \bmod N \end{cases}.$$

Constructing a threshold key essentially consists of computing a Shamir sharing of this. Our solution consists of two steps: 1) First, compute an additive sharing of d. 2) Then, compute Shamir shares of this and decrypt these toward the relevant parties.

Computing an additive sharing of d: First note that

$$\phi(N) \cdot \left(\phi(N)^{-1} \bmod N\right) \equiv \begin{cases} 0 & \bmod \phi(N) \\ 1 & \bmod N \end{cases} \tag{7}$$

where $\phi(N)$ and $(\phi(N)^{-1} \bmod N)$ are viewed as integers. The key primitive of the construction is to add secure multiplication and full decryption to the ElGamal scheme by maintaining a secret state based on additive secret sharing: The parties implicitly hold additive secret sharings of $\phi(N)$: P_1 holds $N + 1 - (p_1 + q_1)$, while P_i holds $-(p_i + q_i)$ for $1 < i \leq k$. Further, the primary goal here is to compute an additive sharing of $\phi(N)^{-1} \bmod N$. This will then be multiplied with the shared $\phi(N)$. To invert $\phi(N)$, the parties utilize the inversion protocol of Bar-Ilan and Beaver, [BB89], simulating $\mathbb{Z}_N$ arithmetic in $\mathbb{Z}_Q$.

Secure multiplication as well as decryption is achieved, by utilizing the ElGamal encryptions as commitments to the shares of the parties. The crucial observation is that when the parties hold additive sharings (over $\mathbb{Z}_Q$) and are committed to those shares – through public ElGamal encryptions of each share – they may obtain an additive sharing of the product as well as ElGamal encryptions of these shares. The protocol, which we denote π_{Π}, is essentially the same as the one used for computing $N = pq$ above. Note that this construction is similar in structure to the protocols of Bendlin et al. [BDOZ11a].

- For $1 \leq i \leq k$, party P_i picks t_i uniformly at random from $\mathbb{Z}_N$ and r_i uniformly at random from $\mathbb{Z}_{N \cdot k \cdot 2^n}$, where n is a security parameter. Each P_i then broadcasts ElGamal two encryptions, c_{t_i} of t_i and c_{r_i} of r_i, and demonstrates plaintext knowledge and that they belong to the specified domains using π_{ENC} and π_{BOUND}. These will be viewed as sharings of random values, $t = \sum_{i=1}^{k} t_i$ and $r = \sum_{i=1}^{k} r_i$.
- The parties execute π_{Π}, obtaining shares $u_1, \ldots, u_n$ of $t \cdot \phi(N)$ as well as encryptions c_{u_i} of those shares.
- For $1 \leq i \leq k$, party P_i broadcasts $u_i + N \cdot r_i$; the parties then decrypt $c_{u_i} \cdot (c_{r_i})^N$, and verify that P_i broadcast the share correctly. If all checks succeed, the parties compute

  $$v = \sum_{i=1}^{k} u_i + N \cdot r_i.$$

  Note that due to the restrictions on the t_i and r_i, $v \equiv t \cdot \phi(N) \bmod N$.
- Each party locally computes the public value[8]

  $$\bar{v} = v^{-1} \bmod N;$$

[8] v is invertible except with negligible probability.

this is then used to compute an additive sharing of $w = t \cdot \bar{v}$; P_i locally multiplies t_i by $\bar{v}$ to compute w_i, and all parties raise the encryptions of the t_i to $\bar{v}$ to obtain encryptions of the w_i.

- Finally, the parties execute π_Π on the shared values $\phi(N)$ and w, thereby obtaining shares d_i of d along with encryptions c_{d_i} of the d_i.

Correctness follows from the fact that the shared $w = \sum_{i=1}^{k} w_i$ equals

$$\Big((t \cdot \phi(N) + r \cdot N)^{-1} \bmod N\Big) \cdot t = \big(\phi(N)^{-1} \bmod N\big) + zN$$

for some integer z of at most $\lceil \log k \rceil + \lceil \log N \rceil$ bits. This implies that

$$d = \big(\big(\phi(N)^{-1} \bmod N\big) + zN\big) \cdot \phi(N) = (\big(\phi(N)^{-1} \bmod N\big)\,\phi(N) + z\phi(N)N.$$

Note that the (at most) $\lceil \log k \rceil + 3\lceil \log N \rceil$-bit value, d, is a proper decryption exponent, as it clearly satisfies

$$d \equiv \begin{cases} 0 \bmod \phi(N) \\ 1 \bmod N \end{cases}$$

Assuming that π_Π securely computes shares of products, then security follows from the fact that the only possible leak is

$$v = \sum_{i=1}^{k} u_i + N \cdot r_i.$$

However, this may be simulated as it is statistically close to a large, random value:

- $v \bmod N \in \mathbb{Z}_N^*$ is the product of t and $\phi(N)$. Since t is the sum of uniformly random values $t_i < N$, $t \bmod N$ is (except with negligible probability) uniformly random in $\mathbb{Z}_N^*$, and – as Paillier encryption requires $\gcd(N, \phi(N)) = 1$ – therefore so is $v \bmod N$. Note that if $t \bmod N \notin \mathbb{Z}_N^*$, then we abort.
- Since $\lfloor v/N \rfloor = \lfloor u/N \rfloor + r$ and r is random and n bits longer than $\lfloor u/N \rfloor$, then $\lfloor v/N \rfloor$ is statistically indistinguishable from a random value distributed as r, even if all but one of the parties are corrupt.

Convert the additive sharing of d to a Shamir sharing: Given the additive sharing of d modulo Q, i.e. d_i held by party P_i and the public encryptions c_{d_i} of the d_i, the parties may compute a "Shamir sharing" of d. Damgård and Jurik do this over the ring $\mathbb{Z}_{N\phi(N)}$, however, here this must be done over the integers, as $N\phi(N)$ is unknown. The parties do this by converting the initial sharing of d to an additive sharing over the integers. In the following, let $\ell_d = \lceil \log n + 3 \log N \rceil$ be the bit-length of d.

1. Conversion to an integer sharing is done by adding a statistically hiding mask to d; the shares then depend on the masked value and the shares of the mask.
   - For $1 \leq i \leq k$, party P_i picks r_i uniformly at random from $\mathbb{Z}_{2^{\ell_d+n}}$, where n is a security parameter. P_i then broadcasts an ElGamal encryption c_{r_i} of r_i, and proves in ZK that it knows the plaintext and that r_i is of the specified size using π_{ENC} and π_{BOUND}.
   - For $1 \leq i \leq k$, party P_i broadcasts $m_i = d_i + r_i$. Moreover, the parties decrypt $c_{d_i} \cdot c_{r_i}$ and verifies the correctness of the share.
   - Party P_1 sets its integer share of d to be

     $$d_{1,\mathbb{Z}} = \left(\sum_{i=1}^{k} m_i \bmod Q\right) - r_1;$$

     all parties compute an encryption $c_{d_{1,\mathbb{Z}}}$ of $d_{1,\mathbb{Z}}$.

- For $1 < i \leq k$, party P_i computes its integer share

$$d_{i,\mathbb{Z}} = -r_1;$$

all parties compute an encryption $c_{d_{i,\mathbb{Z}}}$ of $d_{i,\mathbb{Z}}$.

2. Next, the $d_{i,\mathbb{Z}}$ are threshold shared. Each party P_i threshold-shares $d_{i,\mathbb{Z}}$ and demonstrates that it has done this correctly

- For $1 \leq i \leq k$ and $1 \leq j < t$, party P_i picks $a_{i,j}$ uniformly at random in $\mathbb{Z}_{2^{2\log(N)+n}}$ where n is the statistical security parameter, and broadcasts an ElGamal encryption $c_{a_{i,j}}$ of this. Moreover, it demonstrates knowledge of the plaintext $a_{i,j}$ and that $a_{i,j} < 2^{2\log(N)+n}$ using π_{ENC} and π_{BOUND}. These will be the coefficients used to share $d_{i,\mathbb{Z}}$; let

$$f_i(X) = d_{i,\mathbb{Z}} + \sum_{j=1}^{t-1} a_{i,j} \cdot X^j.$$

Note that $a_{i,j} \bmod N\phi(N)$ is statistically close to uniformly random.

- For $1 \leq i, j \leq k$, the parties compute encryptions $c_{f_i(j)}$ of $f_i(X)$ evaluated at point j. Further, P_i additively shares $f_i(j)$ over the integers among the players; denote P_ι's share $s_{f_i(j),\iota}$. P_i broadcasts ElGamal encryptions $c_{s_{f_i(j),\iota}}$ of these and demonstrate that the plaintexts are known and of bounded bit-length, ℓ_s:

$$\ell_s = \max\left(\ell_d; (\log^t n)(2\log(N)+n)(\log t)\right) + n$$

Moreover, the parties decrypt $\left(c_{f_i(j)}\right)^{-1} \cdot \prod_{\iota=1}^{k} c_{s_{f_i(j),\iota}}$ and verify that the plaintext is 0, i.e. that P_i actually shared the evaluation at $f_i(j)$.

- For $1 \leq j \leq k$, the parties compute additive shares of

$$f(j) = \sum_{i=1}^{k} f_i(j);$$

in addition, they use the homomorphic property to compute encryptions, $c_{f(j),\iota}$ of these values.

3. Finally, for $1 \leq j \leq k$, the parties must reveal the jth threshold share to party P_j. Damgård and Jurik reduce modulo $N\phi(N)$ first, which we cannot do. Hence to ensure no additional information is revealed, a large, random multiple of $N\phi(N)$ is added to each share.

- Since N is public and an integer-sharing of $\phi(N)$ is given, it is simple for the parties to obtain an integer-sharing of $N\phi(N)$: each party simply multiplies its share by N. Further, the parties compute ElGamal encryptions of these new shares, by raising the encryptions of the shares of $\phi(N)$ to the power of N.
- For $1 \leq i, j \leq k$, party P_i broadcast encryptions of uniformly random, $(\ell_s + \log k - 2\log N + n)$-bit values $r_{i,j}$ along with proofs that they are known and of bounded size.
- For $1 \leq j \leq k$, the parties execute π_{II} on the set of $r_{i,j}$ and the sharing of $N\phi(N)$. Each party P_i then adds its share of $f(j)$ to this, and denote this $\sigma_{i,j}$. Moreover, the parties compute ElGamal encryptions $c_{\sigma_{i,j}}$ for all these shares using the homomorphic property.

- For $1 \leq j \leq k$, party P_i sends the share $\sigma_{i,j}$ to P_j. Moreover, the parties decrypt the $c_{\sigma_{i,j}}$ towards P_j, who verifies that it has received the correct shares; finally each P_j computes its share of d, $\sigma_j = \sum_{i=1}^{k} \sigma_{i,j} \equiv f(j) \bmod N\phi(N)$.

Correctness is straightforward: The polynomial f has been constructed over the integers. Clearly $f(0) \equiv d \bmod N\phi(N)$. Security follows from the fact that all messages received are either encrypted *or* random shares, and thus simulatable. The final values, σ_j are statistically indistinguishable from points on a random polynomial over $\mathbb{Z}_{N\phi(N)}$ ***plus*** a sum of uniformly random, $(\ell_s + \log n - 2\log N + n)$-bit multiples of $N\phi(N)$.

Paillier decryption To perform a Paillier decryption with the new threshold key, the parties must raise the ciphertext in question to the power of the share of the key. To prevent malicious behavior, each party must prove in ZK that this has been done correctly – i.e. prove in ZK that the exponent used is also stored in some commitment. Hence, to conclude the key generation, the parties compute and decrypt the ElGamal encryptions $\prod_{i=1}^{k} \sigma_{i,j}$ for $1 \leq j \leq k$. They thereby obtain g^{σ_j}. Using π_{EQ} they parties may show equality of two exponents, i.e. show that they have raised a ciphertext to their part of the key. This differs from [DJ01] who perform this secondary exponentiation modulo a power of N.