Fast Distributed RSA Key Generation for Semi-Honest and Malicious Adversaries

Tore Kasper Frederiksen[1], Yehuda Lindell[2,3], Valery Osheter[3], and Benny Pinkas[2]

[1] Security Lab, Alexandra Institute, Denmark[⋆]
[2] Department of Computer Science, Bar-Ilan University, Israel[⋆⋆]
[3] Unbound Tech Ltd., Israel
tore.frederiksen@alexandra.dk, yehuda.lindell@biu.ac.il, valery.osheter@unboundtech.com, benny@pinkas.net

Abstract. We present two new, highly efficient, protocols for securely generating a distributed RSA key pair in the two-party setting. One protocol is semi-honestly secure and the other maliciously secure. Both are constant round and do not rely on any specific number-theoretic assumptions and improve significantly over the state-of-the-art by allowing a slight leakage (which we show to not affect security).
For our maliciously secure protocol our most significant improvement comes from executing most of the protocol in a "strong" semi-honest manner and then doing a single, light, zero-knowledge argument of correct execution. We introduce other significant improvements as well. One such improvement arrives in showing that certain, limited leakage does not compromise security, which allows us to use lightweight subprotocols. Another improvement, which may be of independent interest, comes in our approach for multiplying two large integers using OT, in the malicious setting, without being susceptible to a selective-failure attack.
Finally, we implement our malicious protocol and show that its performance is an order of magnitude better than the best previous protocol, which provided only *semi-honest* security.

1 Introduction

RSA [RSA78] is the oldest, publicly known, public key encryption scheme. This scheme allows a server to generate a public/private key pair, s.t. any client knowing the public key can use this to encrypt a message, which can only be decrypted using the private key. Thus the server can disclose the public key and

⋆ The majority of the work was done while at Bar-Ilan University, Israel.

⋆⋆ Tore, Yehuda and Benny were supported by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minsters Office. Yehuda and Benny were also been funded by the Israel Science Foundation (grant No. 1018/16). Tore has also received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583.

keep the private key secret. This allows anyone to encrypt a message, which only the server itself can decrypt. Even though RSA has quite a few years on its back, it is still in wide use today such as in TLS, where it keeps web-browsing safe through HTTPS. Its technical backbone can also be used to realize digital signatures and as such is used in PGP. However, public key cryptography, RSA in particular, is also a primitive in itself, widely used in more complex cryptographic constructions such as distributed signature schemes [Sho00], (homomorphic) threshold cryptosystems [HMRT12] and even general MPC [CDN01]. Unfortunately, these complex applications are not in the client-server setting, but in the setting of several distrusting parties, and thus require the private key to be secretly shared between the parties. This is known as *distributed key generation* and in order to do this, without a trusted third party, is no easy feat. Even assuming the parties act semi-honestly, and thus follow the prescribed protocol, it is a slow procedure as the fastest known implementation takes 15 minutes for 2048 bit keys [HMRT12]. For the malicious setting we are unaware of previous implementation. However, in many practical settings such a key sharing only needs to be done once for a static set of parties, where the key pair is then used repeatedly afterwards. Thus, a setup time of 15 minutes is acceptable, even if it is not desirable. Still, there are concrete settings where this is not acceptable.

Motivation. In the world of MPC there are many cases where a setup time of more than a few seconds is unacceptable. For example consider the case of a static server and a client, with a physical user behind it, wishing to carry out some instant, ad-hoc computation. Or the setting where several users meet and want to carry out an auction of a specific item. In these cases, and any case where a specific set of participating parties will only carry out few computations, it is not acceptable for the users to wait more than 15 minutes before they start computing. In such cases only a few seconds would be acceptable.

However, if a maliciously secure shared RSA key pairs could be generated in a few seconds, another possible application appears as well: being able to generate public key pairs in an enterprise setting, without the use of a Hardware Security Module (HSM). A HSM is a trusted piece of hardware pervasively used in the enterprise setting to construct and store cryptographic keys, guaranteed to be correct and leakage free. However, these modules are slow and expensive, and in general reflects a single point of failure. For this reason several companies, such as Unbound and Sepior have worked on realizing HSM functionality in a distributed manner, using MPC and secret-sharing. This removes the single point of failure, since computation and storage will be distributed between physically separated machines, running different operating systems and having different system administrators. Thus if one machine gets fully compromised by an adversary, the overall security of the generated keys will not be affected. This has been done successfully for the generation of symmetric keys, which usually does not need a specific mathematical structure. Unfortunately, doing this for RSA keys is not so easy. However, being able to generate a key pair with the private key secretly shared will realize this functionality. But for such a distributed system to be able to work properly in an enterprise setting such generation tasks must be completed in a matter of seconds.

In this paper we take a big step towards being able to generate a shared RSA key between two parties in a matter of seconds, *even* if one of the parties is acting maliciously and not following the prescribed protocol. Thus opening up for realizing the applications mentioned above.

The Setting. We consider two parties P_1 and P_2 whose goal is to generate an RSA modulus of a certain length, such that the knowledge of the private key is additively shared among them. Namely, the parties wish to compute the following:

Common input: A parameter ℓ describing the desired bits of the primes in an RSA modulus, and a public exponent e.

Common output: A modulus N of length 2ℓ bits.

Private outputs: P_1 learns outputs p_1, q_1, d_1, and P_2 learns outputs p_2, q_2, d_2, for which it holds that

– $(p_1 + p_2)$ and $(q_1 + q_2)$ are prime numbers of length ℓ bits.
– $N = (p_1 + p_2) \cdot (q_1 + q_2)$.
– $e \cdot (d_1 + d_2) = 1 \bmod \phi(N)$.
(Namely, $(d_1 + d_2)$ is the RSA private key for (N, e).)

Furthermore, we want the functionality to work (or abort) even if one of the parties is not following the protocol. That is, in the *malicious* setting.

Distributed RSA Key Generation. It turns out that all prior work follows a common structure for distributed RSA key generation. Basically, since there is no efficient algorithm for constructing random primes, what is generally done is simply to pick random, odd numbers, and hope they are prime. However, the Prime Number Theorem tells us that this is not very likely. In fact, for numbers of the size needed for RSA, the probability that a random odd number is prime is around one in 350. Thus to generate an RSA key, many random prime candidates must be generated and tested in some way. Pairs of prime candidates must then be multiplied together to construct a modulus candidate. Depending on whether the tests of the prime candidates involve ensuring that a candidate is prime except with negligible probability, or only that it is somewhat likely to be prime, the modulus candidate must also be tested to ensure that it is the product of two primes. We briefly outline this general structure below:

Candidate Generation: The parties generate random additive shares of potential prime numbers. This may involve ensuring that a candidate is prime except with negligible probability, insuring that the candidate does not contain small prime factors, or simply that it is just an odd number.

Construct Modulus: Two candidates are multiplied together to construct a candidate modulus.

Verify Modulus: This involves ensuring that the public modulus is the product of two primes. However, this is not needed if the prime candidates were guaranteed to be prime (except with negligible probability).

Construct Keys: Using the additive shares of the prime candidates, along with the modulus, the shared RSA key pair is generated.

With this overall structure in mind we consider the chronology of efficient distributed RSA key generation.

Related Work. Work on efficient distributed RSA key generation was started with the seminal result of Boneh and Franklin [BF01]. A key part of their result is an efficient algorithm for verifying biprimality of a modulus without knowledge of its factors. Unfortunately, their protocol is only secure in the semi-honest setting, against an honest majority. Several followup works handle both the malicious and/or dishonest majority setting [PS98,FMY98,Gil99,ACS02,DM10,HMRT12,Gav12]. First Frankel *et al.* [FMY98] showed how to achieve malicious security against a dishonest minority. Their protocol proceeds like Boneh and Franklin's scheme [BF01], but uses different types of secret sharing along with zero-knowledge arguments to construct the modulus and do the biprimality test in a malicious secure manner. Furthermore, for their simulation proof to go through, they also require that all candidate shares are committed to using an equivocable commitment. Poupard and Stern [PS98] strengthened this result to achieve security against a malicious majority (specifically the two-party setting) using 1-out-of-β OT, with some allowed leakage though. Later Gilboa [Gil99] showed how to get semi-honest security in the dishonest majority (specifically two-party) setting. Gilboa's approach follows along the lines of Boneh and Franklin's protocol [BF01], by using their approach for biprimality testing, but also introduces three new efficient approaches for computing the modulus from additive shares: one based on homomorphic encryption, one based on oblivious polynomial evaluation and one based on oblivious transfer. Both Algesheimer *et al.* [ACS02] and Damgård and Mikkelsen [DM10] instead do a full primality test of the prime candidates individually, rather than a biprimality test of the modulus. In particular the protocol of Algesheimer *et al.* [ACS02] is secure in the semi-honest setting (but can be made malicious secure) against a dishonest minority, and executes a distributed Rabin-Miller primality test using polynomial secret sharing with $\Theta(\log(N))$ round complexity, where N is the public modulus. On the other hand Damgård and Mikkelsen's protocol [DM10] is maliciously secure against a dishonest minority and also executes a distributed Rabin-Miller test, using a special type of verifiable secret sharing called replicated secret sharing which allows them to achieve constant round complexity. Later Hazay *et al.* [HMRT12] introduced a practical protocol maliciously secure against a dishonest majority (in the two-party setting), which is leakage-free. More specifically their protocol is based on the homomorphic encryption approach from Gilboa's work [Gil99], but adds zero-knowledge proofs on top of all the steps to ensure security against malicious parties. However, they conjectured that it would be sufficient to only prove correctness of a constructed modulus. This conjecture was confirmed correct by Gavin [Gav12]. In his work Gavin showed how to build a maliciously secure protocol against a dishonest majority (the two-party setting) by having black-box access to methods for generating a modulus candidate which might be incorrect, but is guaranteed to not leak info on the honest party's shares. The protocol then verifies the execution for every failed candidate and for the success modulus a variant of the Boneh and Franklin biprimality test [BF01] is carried out in a maliciously secure manner by using homomorphic encryption and zero-knowledge.

Contributions. We present two new protocols for distributed RSA key generation. One for the semi-honest setting and one for the malicious setting. Neither of our protocols rely on any specific number theoretic assumptions, but instead are based on oblivious transfer (OT), which can be realized efficiently using an OT extension protocol [KOS15,OOS17]. The malicious secure protocol also requires access to an IND-CPA encryption scheme, coin-tossing, zero-knowledge and secure two-party computation protocols. In fact, using OT extension significantly reduces the amount of public key operations required by our protocols. This is also true for the maliciously secure protocol as secure two-party computation (and thus zero-knowledge) can be done black-box based on OT.

We show that our maliciously secure protocols is more than an order of magnitude faster than its competitor. We achieve malicious security so cheaply mainly by executing a slightly stronger version of our semi-honest protocol and adding a new, lightweight zero-knowledge argument at the end, to ensure that the parties have behaved honestly. This overall idea has been hypothesized [HMRT12] and affirmed [Gav12]. However, unlike previous approaches in this paradigm [DM10,Gav12] our approach does not require rerunning and verifying the honesty of candidates that are discarded, thus increasing efficiency. We achieve this by introducing a new ideal functionality which gives the adversary slightly more (yet useless) power than normally allowed. This idea may be of independent interest as it is relevant for other schemes where many candidate values are constructed and potentially discarded throughout the protocol. We furthermore show how to eliminate much computation in the malicious setting by allowing a few bits of leakage on the honest party's prime shares. We carefully argue that this does not help an adversary in a non-negligible manner.

We also introduce a new and efficient approach to avoid selective failure attacks when using Gilboa's protocol [Gil99] for multiplying two large integers together. We believe this approach may be of independent interest as well.

Finally, we present an implementation of our maliciously secure protocol, showing it to be an order of magnitude faster than the most efficient previous *semi-honest* protocol [HMRT12]. In particular, a four thread implementation takes on average less than 40 seconds to generate a maliciously secure 2048 bit key, whereas the protocol of Hazay *et al.* [HMRT12] on average required 15 minutes for a *semi-honestly* secure 2048 bit key.

2 Preliminaries

Our protocols use several standard building blocks, namely oblivious transfer, and for the maliciously secure protocol, coin-tossing, an IND-CPA encryption scheme, a zero-knowledge protocol along with secure two-party computation. We here formalize these building blocks.

Random OT. Our protocol relies heavily on random OT both in the candidate

FIGURE 2.1 ($\mathcal{F}_{\mathsf{OT}}^{\ell,\beta}$)

Functionality interacts with a sender `snd` and receiver `rec`. It is initialized with the public values $\ell, \beta \in \mathbb{N}$. It proceeds as follows:

- Upon receiving (`transfer`) from `snd` and ($\mathtt{receive}, i$) from `rec` with $i \in \{0, \ldots, \beta - 1\}$ the functionality picks uniformly random values $m_0, \ldots, m_{\beta-1} \in \{0,1\}^\ell$ and sends ($\mathtt{transfer}, m_0, \ldots, m_{\beta-1}$) to `snd` and ($\mathtt{transfer}, m_i$) to `rec`.
- If a party is maliciously corrupted then it will receive its output first and if it returns the message (`deliver`) then the functionality will give the honest party its output, otherwise if the corrupted party returns the message (`abort`), then output (`abort`) to the honest party.

Ideal functionality for random oblivious transfer

generation and construction of modulus phases. The functionality of random OT is described in Figure 2.1. Specifically we suffice with a functionality that samples the sender's messages at random and lets the receiver choose which one of these random messages it wishes to learn. Random OTs of this form can be realized highly efficiently based on an *OT extension*, which uses a small number of "base" OTs to implement any polynomial number of OTs using symmetric cryptography operations alone. The state-of-the-art 1-out-of-2 OT extension is given by Keller *et al.* [KOS15] and for 1-out-of-β OT, by Orrù *et al.* [OOS17]. In some cases we need the sender to be able to specifically choose its messages. However, this is easily achieved by using the random OT-model as a black box and we will sometimes abuse notation and assume that $\mathcal{F}_{\mathsf{OT}}^{\ell,\beta}$ supports specific messages, by allowing the sender to input the message ($\mathtt{transfer}, a_0, \ldots, a_{\beta-1}$), and the receiver receiving message ($\mathtt{transfer}, a_i$).

AES. Our maliciously secure scheme also requires usage of AES. However, any symmetric encryption scheme will do as long as it is a block-cipher (with blocks of at least κ bits) and can be assumed to be a pseudo-random permutation (PRP) and used in a mode that is IND-CPA secure. We will denote this scheme by $\mathsf{AES} : \{0,1\}^\kappa \times \{0,1\}^* \to \{0,1\}^*$ and have that $\mathsf{AES}_K^{-1}(\mathsf{AES}_K(M)) = M$ when $K \in \{0,1\}^\kappa, M \in \{0,1\}^*$.

Coin-tossing. We require a coin-tossing functionality several places in our maliciously secure protocols. Such a functionality samples a uniformly random element from a specific set and hands it to both parties. We formally capture the needed functionality in Figure 2.2.

Zero-Knowledge Argument-of-Knowledge. As part of the setup phase of our malicious protocol we need both parties to prove knowledge of a specific piece of information. For this purpose we require a zero-knowledge argument-of-knowledge. More formally, let $L \subset \{0,1\}^*$ be a publicly known language in NP and M_L be a language verification function of this language i.e. for all $x \in L$ there exist a string w of length polynomial in the size of x s.t. $M_L(x,w) = \top$ and for all $x \notin L, w \in \{0,1\}^*$ then $M_L(x,w) = \bot$. Thus this function outputs $\top$ if and only

FIGURE 2.2 ($\mathcal{F}_{\mathsf{CT}}$)

Functionality interacts with P_1 and P_2. Upon receiving $(\mathtt{toss}, \mathbb{R})$ from both parties, where $\mathbb{R}$ is a description of a ring, sample a uniformly random element $x \in \mathbb{R}$ and send $(\mathtt{random}, x)$ to both parties.

Corruption: If a party is corrupt, then send $(\mathtt{random}, x)$ to this party first, and if it returns the message $(\mathtt{deliver})$ then send $(\mathtt{random}, x)$ to the other party, otherwise if the corrupted party returns the message $(\mathtt{abort})$ then output $(\mathtt{abort})$ to the honest party.

Ideal functionality for coin-tossing

FIGURE 2.3 ($\mathcal{F}_{\mathsf{ZK}}^{M_L}$)

Functionality interacts with two parties P and V. It is initialized on a deterministic polytime language verification function $M_L : \{0,1\}^* \times \{0,1\}^* \rightarrow \{\top, \bot\}$. It proceeds as follows:

– On input $(\mathtt{prove}, x, w)$ from P and $(\mathtt{verify}, x')$ from V. If $x = x'$ and $M_L(x, w) = \top$ output $(\top)$ to V, otherwise output $(\bot)$.

Ideal functionality for zero-knowledge argument-of-knowledge

if w is a string that verifies that x belongs to the language L. We use this to specify the notion of a zero-knowledge argument-of-knowledge that a publicly known value $x \in L$. Specifically one party, P the prover, knows a witness w and wish to convince the other party, V the verifier, that $M_L(x, w) = \top$ without revealing any information on w.

We formalize this in 2.3 and note that such a functionality can be realized very efficiently using garbled circuits [JKO13] or using the "MPC-in-the-head" approach [GMO16].

Two-party Computation. We use a maliciously secure two-party computation functionality in our protocol. For completeness we here formalize the ideal functionality we need for this in 2.4. Such a functionality can be implemented efficiently in constant rounds using a garbled circuit protocol [Lin16].

Notation. We let κ be the computational security parameter and s the statistical security parameter. We use ℓ to denote the amount of bits in a prime factor of an RSA modulus. Thus $\ell \geq \kappa$. We use $[a]$ to denote the list of integers $1, 2, \ldots, a$. We will sometimes abuse notation and implicitly view bit strings as a non-negative integer.

3 Construction

This section details constructions of protocols for two-party RSA key generation. We first describe in Section 3.1 the general structure of our protocols. We describe in Section 3.2 a protocol for the semi-honest setting which is consider-

FIGURE 2.4 ($\mathcal{F}^{f}_{\mathsf{2PC}}$)

Functionality interacts with two parties P_1 and P_2. It is initialized on a deterministic polytime function $f : \{0,1\}^{n_1+n_2} \to \{0,1\}^{m_1+m_2}$. It proceeds as follows:

Input: On input $(\mathtt{input}, x_{\mathcal{I}})$ from $P_{\mathcal{I}}$ where $x_{\mathcal{I}} \in \{0,1\}^{n_{\mathcal{I}}}$, where no message $(\mathtt{input}, \cdot)$ was given by $P_{\mathcal{I}}$ before, store $x_{\mathcal{I}}$.

Output: After having received messages $(\mathtt{input}, \cdot)$ from both P_1 and P_2, compute $y_1 \| y_2 = y = f(x)$ where $x = x_1 \| x_2$ and $y_1 \in \{0,1\}^{m_1}$, $y_2 \in \{0,1\}^{m_2}$. Then return $(\mathtt{output}, y_1)$ to P_1 and $(\mathtt{output}, y_2)$ to P_2.

Corruption: If party $P_{\mathcal{I}}$ is corrupt, then it is given $y_{\mathcal{I}}$ from the functionality before $y_{3-\mathcal{I}}$ is given to $P_{3-\mathcal{I}}$. If $P_{\mathcal{I}}$ returns the message $(\mathtt{deliver})$ then send $y_{3-\mathcal{I}}$ to party $P_{\mathcal{I}}$, otherwise if $P_{3-\mathcal{I}}$ returns the message $(\mathtt{abort})$ then output $(\mathtt{abort})$ to $P_{\mathcal{I}}$.

Ideal functionality for two-party computation

ably more efficient than previous protocols for this task. Finally, we describe in Section 3.3 our efficient protocol which is secure against a malicious adversary.

3.1 Protocol Structure

Following previous protocols for RSA key generation, as described in Section 1, the key generation protocol is composed of the following phases:

Candidate Generation: In this step, the two parties choose random shares p_1 and p_2, respectively, with the hope that $p_1 + p_2$ is prime. For our maliciously secure protocol they also commit to their choices. The parties then run a secure protocol, based on 1-out-of-β OT, which rules out the possibility that $p_1 + p_2$ is divisible by any prime number smaller than some pre-agreed threshold B_1. We call this the first *trial division*.
If $p_1 + p_2$ is not divisible by any such prime then it passed on to the next stage, otherwise it is discarded.

Construct Modulus: Given shares of two candidate primes p_1, p_2 and q_1, q_2, the parties run a secure protocol, based on 1-out-of-2 OT, which computes the candidate modulus $N = (p_1 + p_2)(q_1 + q_2)$. The output N is learned by both parties.

Verify Modulus: This step consists of two phases in our semi-honest protocol and three phases in the malicious protocol. Both protocols proceeds s.t. once N is revealed and in the open, the parties run a second *trial division*, by locally checking that no primes smaller than a threshold B_2 ($B_1 < B_2$) are a factor of N. If N is divisible by such a number then N is definitely not a valid RSA modulus and is discarded. For an N not discarded, the parties run a secure *biprimality test* which verifies that N is the product of two primes. If it is not, it is discarded. For the malicious protocol, a *proof of honesty* phase is added to ensure that N is constructed in accordance with the commitments from *Candidate Generation* and that N is indeed a biprime, constructed using the "correct" shares, even if one party has acted maliciously.

Construct Keys: Up to this point, the parties generated the modulus N. Based on the value $\Phi(N) \mod e$ and their prime shares p_1, q_1, respectively p_2, q_2, the parties can locally compute their shares of the secret key d_1, respectively d_2 s.t. $e \cdot (d_1 + d_2) = 1 \bmod \phi(N)$.

In principle, the protocol could run without the first and second trial division phases. Namely, the parties could choose their shares, compute N and run the biprimality test to check whether N is the product of two primes. The goal of the trial division tests is to reduce the overall run time of the protocol: Checking whether p is divisible by β, filters out $1/\beta$ of the candidate prime factors, and reduces, by a factor of $1 - 1/\beta$, the number of times that the other phases of the protocol need to run. It is easy to see that trial divisions provide diminishing returns as β increases. The thresholds B_1, B_2 must therefore be set to minimize the overall run time of the protocol.

The phases of the protocol are similar to those in previous work that was described in Section 1. Our protocol has two major differences: (1) Almost all cryptographic operations are replaced by the usage of OT extension, which is considerably more efficient than public key operations was has been used previously. (2) Security against malicious adversaries is achieved efficiently, by observing that most of checks that are executed in the protocol can be run while being secure only against semi-honest adversaries, assuming privacy is kept against malicious attacks and as long as the final checks that are applied to the chosen modulus N are secure against malicious adversaries.

3.2 The Semi-honest Construction

The protocol consists of the phases described in Section 3.1, and is described in Figures 3.2 and 3.3. These phases are implemented in the following way:

Candidate Generation: The parties P_1 and P_2 choose private random strings p_1 and p_2, respectively, of length $\ell - 1$ bits, subject to the constraint that the two least significant bits of p_1 are 11, and the two least significant bits of p_2 are 0 (this ensures that the sum of the two shares is equal to 3 modulo 4).

The parties now check, for each prime number $3 \leq \beta \leq B_1$, that $(p_1 + p_2) \neq 0 \bmod \beta$. In other words, if we use the notation $a_1 = p_1 \bmod \beta$ and $a_2 = -p_2 \bmod \beta$, then the parties need to run a secure protocol verifying that $a_1 \neq a_2$.

Previous approaches for doing this involved using a modified BGW protocol [BF01], Diffie-Hellman based public key operations (which have to be implemented over relatively long moduli, rather than in elliptic-curve based groups) [HMRT12], and using a 1-out-of-β OT [PS98]. We take our point of departure in the latter approach, but improve the efficiency by having a lower level of abstraction and using an efficient random OT extension. We describe our approach by procedure Div-OT in Figure 3.1.

The parties run this test for each prime $3 \leq \beta \leq B_1$ in increasing order (where B_1 is the pre-agreed threshold). Note that the probability that the shares are filtered by the test is $1/\beta$ and therefore the test provides diminishing returns as β increases. The threshold B_1 is chosen to optimize the overall performance of the entire protocol.

FIGURE 3.1 (OT-divisibility Test)

The parties have common input $\beta \in \mathbb{N}$ and P_1 has $p_1 \in \mathbb{N}$ and P_2 has $p_2 \in \mathbb{N}$. The procedure returns $\perp$ iff $\beta | (p_1 + p_2)$, otherwise it returns $\top$.

1. P_2 inputs $(\texttt{transfer})$ to $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$ and learns random messages $\{m_i\}_{i \in [\beta]}$.
2. P_1 computes $a_1 = p_1 \mod \beta$ and inputs $(\texttt{receive}, a_1)$ to $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$ and gets output $(\texttt{deliver}, m_{a_1})$.
3. P_2 lets $a_2 = -p_2 \mod \beta$ and sends m_{a_2} to P_1.
4. P_1 checks whether $m_{a_1} = m_{a_2}$ and outputs $\perp$ and sends it to P_2 if this is the case, otherwise it outputs $\top$ and sends this to P_2.

The 1-out-of-β OT based trial division procedure

Construct Modulus: Once two numbers pass the previous test, the parties have shares of two candidate primes p_1, p_2 and q_1, q_2. They then run a secure protocol which computes the candidate modulus

$$N = (p_1 + p_2)(q_1 + q_2) = p_1 q_1 + p_2 q_2 + p_1 q_2 + p_2 q_1.$$

The multiplication $p_1 q_1$ (resp. $p_2 q_2$) is computed by P_1 (resp. P_2) by itself. The other two multiplications are computed by running a protocol by Gilboa [Gil99], which reduces the multiplication of $\ell - 1$ bit long numbers to $\ell - 1$ invocations of 1-out-of-2 OTs, implemented using an efficient OT extension. The protocol works as follows: Assume that the sender's input is a and that the receiver's input is b, and that they must compute shares of $a \cdot b$. Let the binary representation of b be $b = b_{\ell-1}, \ldots, b_2, b_1$. For each bit the two parties run a 1-out-of-2 OT protocol where the sender's inputs are $(r_i, (r_i + a) \bmod 2^{2\ell})$, and the receiver's input is b_i, where r_i is a random $2^{2\ell}$ bit integer. Denote the receiver's output as $c_i = r_i + a \cdot b_i \mod 2^{2\ell}$. It is easy to verify that

$$a \cdot b = \left(\sum\nolimits_{i \in [\ell-1]} 2^{i-1} \cdot c_i\right) + \left(\sum\nolimits_{i \in [\ell-1]} - 2^{i-1} \cdot r_i\right).$$

These will therefore be the two outputs of the multiplication protocol. We implement this protocol based on random OT.

After constructing the modulus, the parties verify that the public exponent e will work with this specific modulus. I.e. that $\gcd(\phi(N), e) = 1$. Namely, that $\gcd(N - p - q + 1, e) = 1$. This is done in the same manner as Boneh and Franklin [BF01], where P_1 computes $w_1 = N + 1 - p_1 - q_1 \mod e$ and P_2 computes $w_2 = p_2 + q_2 \mod e$. The parties exchange the values w_1 and w_2 and then verify that $w_1 \neq w_2$. If instead $w_1 = w_2$ it means that e is a factor of $\phi(N)$ and the parties discard the candidate shares.

Verify Modulus: As previously mentioned, for our semi-honest protocol the verification of the modulus consists of two phases in a pipelined manner; first a *trial division* phase and then a full *biprimality test*. Basically, the full biprimality test is significantly slower than the trial division phase, thus, the trial division phase weeds out unsuitable candidates much cheaper than the biprimality test. Thus, overall we expect to execute the biprimality test much fewer times when doing trial division first.

PROTOCOL 3.2 (Semi-honest Key Generation $\Pi_{\mathsf{RSA\text{-}semi}}$ - Part 1)

Candidate Generation

1. P_1 picks a uniformly random value $\tilde{p}_1 \in \mathbb{Z}_{2^{\ell-3}}$ and defines $p_1 = 4 \cdot \tilde{p}_1 + 3$.
2. P_2 picks a uniformly random value $\tilde{p}_2 \in \mathbb{Z}_{2^{\ell-3}}$ and defines $p_2 = 4\tilde{p}_2$.
3. Let $\mathcal{B} = \{\beta \leq B_1 | \beta \text{ is prime}\}$. The parties execute procedure $\mathsf{Div\text{-}OT}$ in Figure 3.1 for each $\beta \in \mathcal{B}$, where P_1 uses input p_1 and P_2 input p_2. If any of these calls output $\perp$, then discard the candidate pair p_1, p_2.

Construct Modulus

Let p_1, q_1, p_2, q_2 be two candidates that passed the generation phase above, where P_1 knows $p_1 = 4 \cdot \tilde{p}_1 + 3, q_1 = 4 \cdot \tilde{q}_1 + 3$ and P_2 knows $p_2 = 4 \cdot \tilde{p}_2, q_2 = 4 \cdot \tilde{q}_2$.

1. The parties execute the following steps for each $\alpha \in \{p, q\}$ and $i \in [\ell - 1]$:
   (a) P_2 chooses a uniformly random value $r_{\alpha,i} \in \mathbb{Z}_{2^{2\ell}}$ and sets $c_{0,\alpha,i} = r_{\alpha,i}$ and
   $$c_{1,\alpha,i} = \begin{cases} r_{\alpha,i} + q_2 \mod 2^{2\ell} & \text{if } \alpha = p \\ r_{\alpha,i} + p_2 \mod 2^{2\ell} & \text{if } \alpha = q \end{cases}$$
   (b) P_2 invokes $\mathcal{F}_{\mathsf{OT}}^{2\ell,2}$ with input $(\mathtt{transfer}, c_{0,\alpha,i}, c_{1,\alpha,i})$.
   (c) P_1 inputs $(\mathtt{receive}, \alpha_{1,i})$ to $\mathcal{F}_{\mathsf{OT}}^{2\ell,2}$, $\alpha_{1,i}$ is the i'th bit of α_1. P_1 thus receives the message $(\mathtt{deliver}, c_{\alpha_{1,i},i})$ from $\mathcal{F}_{\mathsf{OT}}^{2\ell,2}$ for $i \in [\ell - 1]$.
2. P_1 computes $z_1^{\alpha} = \sum_{i \in [\ell-1]} c_{\alpha_{1,i},i} \cdot 2^{i-1} \mod 2^{2\ell}$ and P_2 computes $z_2^{\alpha,i} = -\sum_{i \in [\ell-1]} r_{\alpha,i} \cdot 2^{i-1} \mod 2^{2\ell}$.
3. P_2 computes $a_2 = p_2 q_2 + z_2^p + z_2^q \mod 2^{2\ell}$ and sends this to P_1.
4. P_1 computes $a_1 = p_1 q_1 + z_1^p + z_1^q \mod 2^{2\ell}$ and sends this to P_2.
5. P_1 and P_2 then compute $(p_1 + p_2)(q_1 + q_2) = N = (a_1 + a_2 \mod \mathcal{P}) \mod 2^{2\ell}$.
6. P_1 computes $w_1 = N + 1 - p_1 - q_1 \mod e$ and sends this to P_2. Similarly P_2 computes $w_2 = p_2 + q_2 \mod e$ and sends this to P_1.
7. P_1 and P_2 checks if $w_1 = w_2$. If this is the case they discard the candidate N and its associated shares p_1, q_1, p_2, q_2. Otherwise they define the value $w = w_1 - w_2 \mod e$ for later use.

Protocol for semi-honestly secure RSA key generation in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model

The trial division phase itself is very simple: since both parties know the candidate modulus N, one party simply try to divide it by all primes numbers in the range $B_1 < \beta \leq B_2$. If successful, then N is discarded.

If N passes the trial division, we must still verify that it is in fact a biprime, except with negligible probability. To do this we use a slightly modified version of the biprimality suggested by Boneh and Franklin [BF01], which relies on number-theoretic properties of $N = pq$ where $p = 3 \bmod 4$ and $q = 3 \bmod 4$. (Note that in the prime-candidate generation, p and q were guaranteed to have this property.) The test is described in Figure 3.4. By slightly modified, we mean

PROTOCOL 3.3 (Semi-honest Key Generation $\Pi_{\mathsf{RSA\text{-}semi}}$ - Part 2)

Trial Division

Let $\mathcal{B} = \{B_1 < p \leq B_2 | p \text{ is prime}\}$ for some previously decided B_2. P_2 then executes trial division of the integers up to B_2. If a factor is found then send $\perp$ to P_1 and discard N and its associated prime shares p_1, q_1, p_2, q_2. Otherwise send $\top$ to P_1.

Biprimality Test

The parties execute the biprimality test described in Figure 3.4 and discard the candidate N if the test fails.

Generate Shared Key

1. Both parties use the value w computed in 7 in **Construct Modulus** associated with the candidate N to compute $b = w^{-1} \mod e$, and then finally P_1 computes $d_1 = \lfloor \frac{-b\cdot(N+1-p_1-q_1)+1}{e} \rfloor$. If $e| - p_2 - q_2$ then P_2 computes $d_2 = 1 + \lfloor \frac{-b\cdot(-p_2-q_2)}{e} \rfloor$, otherwise P_2 computes $d_2 = \lfloor \frac{-b\cdot(-p_2-q_2)}{e} \rfloor$.
2. P_1 outputs (N, p_1, q_1, d_1) and P_2 outputs (N, p_2, q_2, d_2).

Protocol for semi-honestly secure RSA key generation in the $\mathcal{F}_{\mathsf{OT}}$-hybrid model

FIGURE 3.4 (Biprimality test [BF01])

1. The parties execute following test s times.
   (a) P_1 samples a random value $\gamma \in \mathbb{Z}_N^\times$ with Jacobi symbol 1 over N.
   (b) P_1 sends γ to P_2.
   (c) P_1 computes $\gamma_1 = \gamma^{\frac{N+1-p_1-q_1}{4}} \mod N$ and sends this value to P_2.
   (d) P_2 checks if $\gamma_1 \cdot \gamma^{\frac{-p_2-q_2}{4}} \mod N \neq \pm 1$. In this case P_2 sends $\perp$ to P_1 and the parties break the loop and discard the candidate N.
2. The parties verify that $\gcd(N, p+q-1) = 1$.
   (a) P_1 chooses a random number $\bar{r}_1 \in \mathbb{Z}_{2^{\ell+s}}$ and P_2 chooses a random $\bar{r}_2 \in \mathbb{Z}_{2^{\ell+s}}$. (The parties will verify that $\gcd((\bar{r}_1+\bar{r}_2)\cdot(p+q-1), N) = 1$.)
   (b) The parties run a multiplication protocol (similar to that run in the "Construct Modulus" step modulo $2^{2\ell+s+2}$) where they compute shares α_1, α_2 (known to P_1, P_2 respectively) of $\bar{r}_1 \cdot (p_2 + q_2 - 1) \mod 2^{2\ell+s+2}$, and shares β_1, β_2 of $\bar{r}_2 \cdot (p_1 + q_1) \mod 2^{2\ell+s+2}$.
   (c) P_1 sends to P_2 the value $s_1 = \bar{r}_1(p_1 + q_1) + \alpha_1 + \beta_1 \mod 2^{2\ell+s+2}$.
   (d) P_2 computes $s_2 = \bar{r}_2(p_2 + q_2 - 1) + \alpha_2 + \beta_2 \mod 2^{2\ell+s+2}$, and verifies that $\gcd(s_1 + s_2, N) = 1$. If this is not the case then it sends $\perp$ to P_1 and discard the candidate N.

The biprimality test of Boneh and Franklin [BF01]

that step 2), which ensures that $\gcd(N, p+q-1) = 1$, is computed without the need of doing operations in the group $\mathbb{Z}_N[x]/(x^2+1))^*/\mathbb{Z}_N^*$.

Construct Keys: This phase is a simplified version of what is done by Boneh and Franklin [BF01]. Using the values w_1 and w_2 defined in *construct modulus* the parties compute $w = w_1 - w_2 \mod e$ and then $b = w^{-1} \mod e$.

FIGURE 3.5 ($\mathcal{F}_{\mathsf{RSA\text{-}semi}}$)

Functionality interacts with parties P_1 and P_2. Upon query of an integer $\ell \in \mathbb{N}$ and a prime e from both parties the functionality proceeds as follows:

– Sample random values p_1, p_2, q_1, q_2 of $\ell - 1$ bits each s.t. $p_1 \equiv q_1 \equiv 3 \mod 4$ and $p_2 \equiv q_2 \equiv 0 \mod 4$, $p = p_1 + p_2$ and $q = q_1 + q_2$ are prime, and $\gcd((p-1)(q-1), e) = 1$.
– Compute $d = e^{-1} \mod (p-1)(q-1)$, let $b = ((p-1)(q-1))^{-1} \mod e$ and set $d_2 = \lfloor \frac{-b\cdot(-p_2-q_2)}{e} \rfloor$ and $d_1 = d - d_2$.
– Output $(N = pq, b, p_1, q_1, d_1)$ to P_1 and $(N = pq, b, p_2, q_2, d_2)$ to P_2.

Ideal functionality for generating a shared RSA key semi-honestly

P_1 defines its share of the private key as $d_1 = \lfloor \frac{-b\cdot(N+1-p_1-q_1)+1}{e} \rfloor$. P_2 defines its share of the private key as $d_2 = 1 + \lfloor \frac{-b\cdot(-p_2-q_2)}{e} \rfloor$ or $d_2 = \lfloor \frac{-b\cdot(-p_2-q_2)}{e} \rfloor$ or depending on whether $e|p_2+q_2$ or not.

We formally describe the full semi-honest protocol in Figure 3.2 and 3.3.

Ideal functionality. The exact ideal functionality, $\mathcal{F}_{\mathsf{RSA\text{-}semi}}$, our semi-honest protocol realizes is expressed in Figure 3.5. The functionality closely reflects the specific construction of the modulus and the shares of the private key of our protocol. In particular, we notice that both primes of the public modulus are congruent to 3 modulo 4, which is needed for the Boneh and Franklin biprimality test to work. Based on these shared primes, the shares of the private keys are generated and handed to the parties. This part of the functionality closely follows the previous literature [BF01,Gil99,ACS02,DM10,Gav12]. First notice using primes congruent to 3 modulo 4 does not decrease security. This follows since all primes suitable for RSA are odd this means that only about half of potential primes are not used. Thus the amount of possible moduli are reduced by around 75%. However, this is similar to all previous approaches. Furthermore, this does not give an adversary any noticeable advantage in finding primes used in key generation.

Next notice that the value $\phi(N) \mod e$ is leaked. This leakage comes implicitly from how the shares d_1 and d_2 are constructed (although it is made explicit in the ideal functionality). We note that since we use the method of Boneh and Franklin [BF01] for this computation, this leakage is also present in their work and any other protocol that uses this approach to generate the shared keys. Specifically this means that at most $\log(e)$ bits of information on the honest party's secret shares are leaked. Thus when e is small, this does not pose any issue. However, using the common value of $e = 2^{16}+1$ this could pose a problem. We show how to avoid leaking $\phi(N) \mod e$ in our maliciously secure protocol.

Using this functionality we get the following theorem:

Theorem 3.6. *The protocol $\Pi_{\mathsf{RSA\text{-}semi}}$ in Figure 3.2 and 3.3 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{RSA\text{-}semi}}$ in Figure 3.5 against a static and semi-honest adversary in the $\mathcal{F}_{\mathsf{OT}}^{\cdot,\cdot}$-hybrid model.*

We will not prove this theorem directly. The reason being that the following section will make it apparent that *all* steps of the semi-honest protocol is also part of the malicious protocol. Now remember that a simulator for a semi-honest protocol receives the output of the corrupt party. In our protocol this will in particular mean the prime shares. Thus our semi-honest simulator will proceed like the malicious one for the same steps, using the corrupt party's prime shares.

3.3 Malicious construction

The malicious protocol follows the semi-honest one with the following exceptions:

- The underlying OT functionality must be maliciously secure.
- An extractable commitment to each party's choice of shares is added to the *construct candidate* phase. This is needed since the simulator must be able to extract the malicious party's choice of shares in order to construct messages indistinguishable from the honest party, consistent with any cheating strategy of the malicious party.
- A new and expanded version of the Gilboa protocol is used to compute a candidate modulus. This is done since a malicious P_2 (the party acting as the sender in the OTs) might launch a *selective failure attack* (details below).
- We use OT to implement an equality check of $w_1 = N+1-p_1-q_1 \mod e$ and $w_2 = p_2+q_2 \mod e$ to ensure that $\gcd(\phi(N), e) = 1$ without leaking w_1 and w_2, and thus avoid leaking $\phi(N) \mod e$ which is leaked in the semi-honest protocol.
- A *proof of honesty* step is added to the *verify modulus* phase, which is used to have the parties prove to one another that they have executed the protocol correctly.
- The private key shares are randomized and computed using a secure protocol.

For OT we simply assume access to any ideal functionalities as described in Section 2. Regarding the AES-based commitments, the expanded Gilboa protocol and the proof of honesty, we give further details below.

AES-based commitments. We implement these "commitments" as follows: Before *Candidate Generation*, in a phase we will call *Setup* each party "commits" to a random AES key K by sending $c = \mathsf{AES}_{\mathsf{AES}_r(K)}(0)$ for a random r (chosen by coin-tossing). This unusual "double encryption" ensure that c is not only *hiding* K, through the encryption, but also *binding* to K. The key K is then used to implement a committing functionality. This is done by using K as the key in an AES encryption, where the value we want to commit to is the message encrypted. However, for our proof to go through we require this "commitment" to be extractable. Fortunately this is easily achievable if the simulator knows K and to ensure this we do a zero-knowledge argument of knowledge of K s.t. $c = \mathsf{AES}_{\mathsf{AES}_r(K)}(0)$. By executing this zero-knowledge argument the simulator can clearly extract K (assuming the zero-knowledge argument is an ideal functionality).

Expanded Gilboa Protocol. The usage of OT in the malicious setting is infamous for selective failure vulnerabilities [KS06,MF06] and our setting is no different.

Specifically, what a malicious P_2 can do is to guess that P_1's choice bit is 0 (or 1) in a given step of the Gilboa protocol. In this case, P_2 inputs the correct message for choice 0, i.e. the random string r. But for the message for a choice of 1 it inputs the 0-string. If P_2's guess was correct, then the protocol executes correctly. However, if its guess was wrong, then the result of the Gilboa protocol, i.e. the modulus, will be incorrect. If this happens then the protocol will abort during the proof of honesty. Thus, two distinct and observable things happen dependent on whether P_2's guess was correct or not and so P_2 learns the choice bit of P_1 by observing what happens. In fact, P_2 can repeat this as many times as it wants, each time succeeding with probability 1/2 (when P_1's input is randomly sampled). This means that with probability 2^{-x} it can learn x of P_1's secret input bits.

To prevent this attack we use the notion of *noisy encodings*. A noisy encoding is basically a linear encoding with some noise added s.t. decoding is *only* possible when using some auxiliary information related to the noise. We have party P_1 noisily encode its true input to the Gilboa protocol. Because of the linearity it is possible to retrieve the true output in the last step of the Gilboa protocol (where the parties send their shares to each other in order to learn the result N) without leaking anything on the secret shares of P_1, even in the presence of a selective failure attack.

In a bit more detail, we define a 2^{-s}-statistically hiding noisy encoding of a value $a \in \mathbb{Z}_{2^{\ell-1}}$ as follows:

– Let $\mathcal{P}$ be the smallest prime larger than $2^{2\ell}$.
– Pick random values $h_1, \ldots, h_{2\ell+3s}, g \in \mathbb{F}_{\mathcal{P}}$ and random bits $d_1, \ldots, d_{2\ell+3s}$ under the constraint that $g + \sum_{i \in [2\ell+3s]} h_i \cdot d_i \mod \mathcal{P} = a$.
– The noisy encoding is then $(h_1, \ldots, h_{2\ell+3s}, g)$ and the decoding info is $(d_1, \ldots, d_{2\ell+3s})$.

Now for each of its shares, p_1 and q_1, P_1 noisily encodes as described and sends the noisy encodings $(h_{p,1}, \ldots, h_{p,2\ell+3s}, g_p)$ and $(h_{q,1}, \ldots, h_{q,2\ell+3s}, g_q)$ to P_2. Next, when they execute the OT steps, P_1 uses the decoding info $(d_{p,1}, \ldots, d_{p,2\ell+3s})$ and $(d_{q,1}, \ldots, d_{q,2\ell+3s})$ of p_1 and q_1 respectively and uses this as input the OTs instead of the bits of p_1 and q_1. For each such bit of p_1, P_2 inputs to the OT a random value $c_{0,p,i} = r_i$ and the value $c_{1,p,i} = r_i + q_2$ (and also operates in a similar way for q). P_1 then receives the values $c_{d_{p,i},p,i}, c_{d_{q,i},q,i} \in \mathbb{Z}_{\mathcal{P}}$ and P_2 holds the values $c_{0,p,i}, c_{1,p,i}, c_{0,q,i}, c_{1,q,i} \in \mathbb{Z}_{\mathcal{P}}$. It turns out that leaking at most s bits of $(d_{p,1}, \ldots, d_{p,2\ell+3s})$ and $(d_{q,1}, \ldots, d_{q,2\ell+3s})$ to P_2 does not give more than a 2^{-s} advantage in finding the value encoded. Thus, even if P_2 launches s selective failure attacks it gains no significant knowledge on P_1's shares.

After having completed the OTs, the parties compute their shares of the modulus N by using the linearity of the encodings. We believe that this approach to thwart selective failure attacks, when multiplying large integers, could be used other settings as well. In particular, we believe that for certain choices of parameters our approach could make a protocol like MASCOT [KOS16] more efficient since it would be possible to eliminate (in their terminology) the *combining step*.

Proof of Honesty. The proof of honesty has three responsibilities: first, it is a maliciously secure execution of the full biprimality test of Boneh and Franklin [BF01]; second, it verifies that the modulus is constructed from the values committed to in *candidate generation*. Finally it generates a random sharing of the private key. The proof of honesty is carried out twice. Once where party P_1 acts as the prover and P_2 the verifier, and once where P_2 acts the prover and P_1 the verifier. Thus each party gets convinced of the honesty of the other party and learns their respective shares of the private key.

To ensure a correctly executed biprimality test, a typical zero-knowledge technique is used, where coin-tossing is used to sample public randomness and the prover randomizes its witness along with the statement to prove. The verifier then gets the option to decide whether he wants to learn the value used for randomizing or the randomized witness. This ensures that the prover can only succeed with probability 1/2 in convincing the verifier if it does not know a witness.

To ensure that the modulus was constructed from the values committed to, a small secure two-party computation is executed which basically verifies that this is the case. Since the commitments are AES-based, this can be carried out in a very lightweight manner. Furthermore, to ensure that the values used in the maliciously secure biprimality test are also consistent with the shares committed to, we have the prover commit to the randomization values as well and verify these, along with their relation to the shares. Finally, we let the proving party input some randomness which is used to randomize the verifying party's share of the private key. We formally describe the full protocol in Figures 3.7, 3.8,3.9 and 3.10.

Ideal Functionality. We express the ideal functionality that our protocol realizes in Fig. 3.12. When there is no corruption the functionality simply proceeds almost as the semi-honest functionality in 3.5. That is, making a shared key based on random primes congruent to 3 modulo 4, *but* where the shares of the secret key are sampled at random in the range between $-2^{2\ell+s}$ and $2^{2\ell+s}$. This means that the value $\phi(N) \mod e$ is *not* leaked. When a party is corrupted the adversary is allowed certain freedoms in its interaction with the ideal functionality. Specifically the adversary is given access to several commands, allowing it and the functionality to generate a shared RSA key through an interactive game.

The functionality closely reflects what the adversary can do in our protocol. Specifically we allow a malicious party to repeatedly query the functionality to learn a random modulus, based on its choice of prime shares. This is reflected by commands *sample* and *construct*. *Sample* lets the adversary input its desired share of a prime and the functionality then samples a random share for the honest party s.t. the sum is prime. This command also ensures that the primes work with the choice of public exponent e. I.e., that $\gcd(e, (p-1)(q-1)) = 1$. Specifically it verifies that the gcd of e and the prime candidate minus one is equal to one. This implies that no matter which two primes get paired to construct a

PROTOCOL 3.7 (Malicious Key Generation Π_{RSA} - Part 1)

Setup

1. The parties call $(\texttt{toss}, \{0,1\}^\kappa)$ on $\mathcal{F}_{\mathsf{CT}}$ twice to sample uniformly random bitvectors $r_1, r_2 \in \{0,1\}^\kappa$. (Note that these outputs are known to both parties.)
2. For $\mathcal{I} \in \{1,2\}$ party $P_\mathcal{I}$ picks a uniformly random value $K_\mathcal{I} \in \{0,1\}^\kappa$, computes and sends $\mathsf{AES}_{\mathsf{AES}_{r_\mathcal{I}}(K_\mathcal{I})}(0) = c_\mathcal{I}$ to $P_{3-\mathcal{I}}$.
3. Let M_L be the function outputting $\top$ on input $((r_\mathcal{I}, c_\mathcal{I}), K_\mathcal{I})$ if and only if $\mathsf{AES}_{\mathsf{AES}_{r_\mathcal{I}}(K_\mathcal{I})}(0) = c_\mathcal{I}$. For $\mathcal{I} \in \{1,2\}$ party $P_\mathcal{I}$ inputs $(\texttt{prove}, (r_\mathcal{I}, c_\mathcal{I}), K_\mathcal{I})$ on $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ and party $P_{3-\mathcal{I}}$ inputs $(\texttt{verify}, (r_\mathcal{I}, c_\mathcal{I}))$. (The simplest way of implementing these proofs is probably using garbled circuits [JKO13].)
4. If any of these calls output $(\perp)$ then the parties abort. Otherwise they continue.

Candidate Generation

1. P_1 picks a uniformly random value $\tilde{p}_1 \in \mathbb{Z}_{2^{\ell-3}}$, defines $p_1 = 4 \cdot \tilde{p}_1 + 3$, computes and sends $H_{\tilde{p}_1} = \mathsf{AES}_{K_1}(\tilde{p}_1)$ to P_2.
2. P_2 picks a uniformly random value $\tilde{p}_2 \in \mathbb{Z}_{2^{\ell-3}}$ and defines $p_2 = 4 \cdot \tilde{p}_2$, computes and sends $H_{\tilde{p}_2} = \mathsf{AES}_{K_2}(\tilde{p}_2)$ to P_1.
3. Let $\mathcal{B} = \{\beta \leq B_1 | \beta \text{ is prime}\}$. The parties execute procedure $\mathsf{Div\text{-}OT}$ in Figure 3.1 for each $\beta \in \mathcal{B}$, where P_1 uses input p_1 and P_2 input p_2. If any of these calls output $\perp$, then discard the candidate pair p_1, p_2.

Protocol for maliciously secure RSA key generation.

modulus, it will always hold that $\gcd(e, \phi(N)) = 1$. *Construct* lets the adversary decide on two primes (of which it only knows its own shares) that should be used to construct a modulus and generate shares of the secret key in the same manner as done by Boneh and Franklin [BF01]. Finally, the adversary can then decide which modulus it wishes to use, which is reflected in the command *select*.

However, the functionality does allow the adversary to learn a few bits of information of the honest party's prime shares. In particular, the trial division part of our *candidate generation* phase, allows the adversary to gain some knowledge on the honest party's shares, as reflected in command *leak*. Specifically the adversary gets to guess the remainder of the honest party's shares modulo β, for each $\beta \in \mathcal{B}$ and is informed whether its guess was correct or not. In case the malicious party is P_2 then if any of its guesses for a particular prime is wrong, the adversary loses the option of selecting the modulus based on this prime.

The functionality keeps track of the adversary's queries and what was leaked to it, through the set J and dictionary C. Basically the set J stores the unique ids of primes the simulator has generated and which the adversary can use to construct an RSA modulus. Thus the ids of primes already used to construct a modulus are removed from this set. The same goes for primes a malicious P_2 have tried to learn extra bits about, but failed (as reflected in *leak*). The dictionary C on the other hand, maps prime ids already used to construct an RSA modulus, into this modulus. This means that once two primes have been

PROTOCOL 3.8 (Malicious Key Generation Π_{RSA} - Part 2)

Construct Modulus

Let p_1, q_1, p_2, q_2 be two candidates that passed the generation phase above, where P_1 knows $p_1 = 4\tilde{p}_1 + 3, q_1 = 4\tilde{q}_1 + 3$ and $H_{\tilde{p}_2}, H_{\tilde{q}_2}$ and P_2 knows $p_2 = 4\tilde{p}_2, q_2 = 4\tilde{q}_2$ and $H_{\tilde{p}_1}, H_{\tilde{q}_1}$. Furthermore, let $\mathcal{P}$ be the smallest prime number greater than $2^{2\ell}$.

1. For each $\alpha \in \{p, q\}$ party P_1 picks a list of values $h_{\alpha,1}, \ldots, h_{\alpha,2\ell+3s}, g_\alpha \in \mathbb{Z}_\mathcal{P}$ and a list of bits $d_{\alpha,1}, \ldots, d_{\alpha,2\ell+3s} \in \{0,1\}$ uniformly at random under the constraint that $g_\alpha + \sum_{i\in[2\ell+3s]} h_{\alpha,i} \cdot d_{\alpha,i} \mod \mathcal{P} = \alpha_1$.
2. The parties execute the following steps for each $\alpha \in \{p, q\}$ and $i \in [2\ell + 3s]$:
   (a) P_2 chooses a uniformly random value $r_{\alpha,i} \in \mathbb{Z}_\mathcal{P}$ and sets
   $$c_{0,\alpha,i} = r_{\alpha,i} \quad \text{and} \quad c_{1,\alpha,i} = \begin{cases} r_{\alpha,i} + q_2 \mod \mathcal{P} & \text{if } \alpha = p \\ r_{\alpha,i} + p_2 \mod \mathcal{P} & \text{if } \alpha = q \end{cases}.$$
   (b) P_2 invokes $\mathcal{F}_{\mathsf{OT}}^{2\ell+3s,2}$ with input $(\texttt{transfer}, c_{0,\alpha,i}, c_{1,\alpha,i})$.
   (c) P_1 inputs $(\texttt{receive}, d_{\alpha,i})$ to $\mathcal{F}_{\mathsf{OT}}^{2\ell+3s,2}$. P_1 thus receives the message $(\texttt{deliver}, c_{d_{\alpha,i},i})$ from $\mathcal{F}_{\mathsf{OT}}^{2\ell+3s,2}$.
3. P_1 sends the values $h_{\alpha,1}, \ldots, h_{\alpha,2\ell+3s}, g_\alpha$ to P_2 for $\alpha \in \{p, q\}$.
4. P_1 computes $z_1^\alpha = \sum_{i\in[2\ell+3s]} c_{d_{\alpha,i},i} \cdot h_{\alpha,i} \mod \mathcal{P}$ and P_2 computes $z_2^\alpha = -\sum_{i\in[2\ell+3s]} r_{\alpha,i} \cdot h_{\alpha,i} \mod \mathcal{P}$.
5. P_2 computes $a_2 = p_2 q_2 + z_2^p + g_p \cdot q_2 + z_2^q + g_q \cdot p_2 \mod \mathcal{P}$ and sends this to P_1.
6. P_1 computes $a_1 = p_1 q_1 + z_1^p + z_1^q \mod \mathcal{P}$ and sends this to P_2.
7. P_1 and P_2 then compute $N = (a_1 + a_2 \mod \mathcal{P}) \mod 2^{2\ell}$.
8. P_1 computes $w_1 = N + 1 - p_1 - q_1 \mod e$ and similarly P_2 computes $w_2 = p_2 + q_2 \mod e$.
9. P_2 inputs $(\texttt{transfer})$ to $\mathcal{F}_{\mathsf{OT}}^{\kappa,\lceil\log(e)\rceil}$ and learns $r_0, \ldots, r_{\beta-1} \in \{0,1\}^\kappa$.
10. P_1 inputs $(\texttt{receive}, w_1)$ and thus learns r_{w_1}.
11. P_2 sends r_{w_2} to P_1.
12. If $r_{w_1} = r_{w_2}$ then P_1 informs P_2 of this and they both discard the candidate N and its associated shares p_1, q_1, p_2, q_2.

Protocol for maliciously secure RSA key generation.

used, using *construct*, to construct a modulus, their ids are removed from J and instead inserted into C. After the construction we furthermore allow the adversary $P_\mathcal{I}$ to pick a value $w'_\mathcal{I} \in [0, e[$ and learn if $w'_\mathcal{I} = w_{3-\mathcal{I}}$ when $w_{3-\mathcal{I}}$ is constructed according to the protocol using the honest party's shares. However, if the corrupt party is P_2 and it guesses *correctly* then it won't be allowed to use the candidate.

Finally we have the command *abort* which allows an adversary to abort the functionality at any point it wishes, as is the norm in maliciously secure dishonest majority protocols.

Security. If there is no corruption we have the same security as described for the

PROTOCOL 3.9 (Malicious Key Generation Π_{RSA} - Part 3)

Trial Division

Let $\mathcal{B} = \{B_1 < p \leq B_2 | p \text{ is prime}\}$. P_2 then executes trial division of the integers up to B_2. If a factor is found then send $\perp$ to P_1 and discard N and its associated prime shares p_1, q_1, p_2, q_2. Otherwise send $\top$ to P_1.

Biprimality Test

The parties execute the biprimality test described in Figure 3.4. However after step 2.a P_1 sends $H_{\bar{r}_1} = \mathsf{AES}_{K_1}(\bar{r}_1)$ and P_2 sends $H_{\bar{r}_2} = \mathsf{AES}_{K_2}(\bar{r}_2)$. Furthermore in step 2.b they use the maliciously secure version of the multiplication protocol from "Construct Modulus" (modulo the smallest prime larger than $2^{2\ell+s+2}$).

Proof of Honesty

The parties call $(\mathtt{toss}, \mathbb{Z}_N^\times)$ on $\mathcal{F}_{\mathsf{CT}}$ enough times to get s distinct random elements, denoted by $\gamma_i \in \mathbb{Z}_N^\times$ s.t. $\mathcal{J}_N(\gamma_i) = 1$ for $i \in [s]$.
(Recall that $\tilde{p}$ denotes $p \gg 2$, i.e. p shifted to the right two steps.) Execute the following steps where $P = P_1, V = P_2$ with $\tilde{p}_P = \tilde{p}_1$ and $\tilde{p}_V = \tilde{p}_2$ *and* where $P = P_2, V = P_1$ with $\tilde{p}_P = \tilde{p}_2$ and $\tilde{p}_V = \tilde{p}_1$. Similarly for $\tilde{q}_1, \tilde{q}_2$:

1. For each $i \in [s]$, P computes $\gamma_{i,P} = \gamma_i^{\frac{N-5}{4} - \tilde{p}_P - \tilde{q}_P} \mod N$
2. P sends $\gamma_{1,P}, \ldots, \gamma_{s,P}$ to V.
3. For each $i \in [s]$, V then verifies that $\gamma_i^{-\tilde{p}_V - \tilde{q}_V} \cdot \gamma_{i,P} \equiv \pm 1 \mod N$.
4. If *any* of the checks do not pass then V sends $\perp$ to P, outputs $\perp$ and aborts.
5. For each $j \in [s]$, P picks a random value $t_j \in \{0,1\}^{\ell-2+s}$. It then computes $\mathsf{AES}_{K_P}(t_j) = H_{t_j}$ and sends this to V.
6. For each $i, j \in [s]$, P then sends the values $\bar{\gamma}_{i,j} = \gamma_i^{t_j} \mod N$ to V.
7. The parties call $(\mathtt{toss}, \{0,1\})$ on $\mathcal{F}_{\mathsf{CT}}$ s times to sample uniformly random bits $b_1, \ldots, b_s \in \{0,1\}$.
8. For each $j \in [s]$, P then sends $v_j = b_j \cdot (-\tilde{p}_P - \tilde{q}_P) + t_j$ to V.
9. For each $i, j \in [s]$ V checks that

$$\gamma_i^{v_j} \mod N \overset{?}{=} \bar{\gamma}_{i,j} \cdot \gamma_{i,P}^{b_j} \cdot \gamma_i^{-b_j \cdot \frac{N-5}{4}} \mod N$$

If this is not the case then it sends $\perp$ to P, outputs $\perp$ and aborts.

Protocol for maliciously secure RSA key generation.

semi-honest protocol in Section 3.2, except that $\phi(N) \mod e$ is *not* implicitly leaked by party's secret key share. Next we see that allowing the adversary to query the functionality for moduli before making up its mind does not influence security. Since the adversary is polytime bounded, it can only query for polynomially many moduli. Furthermore, as the honest party's shares are randomly sampled by the functionality, and since they are longer than the security parameter (e.g. 1021 vs. 128) the adversary will intuitively not gain anything from having this ability. Arguments for why this is the case have been detailed by Gavin [Gav12].

Regarding the allowed leakage we show in the full version that (for standard parameters) at most $\log_2(e/(e-1)) + 2\log_2(B_1/2)$ bits are leaked to a malicious

PROTOCOL 3.10 (Malicious Key Generation Π_{RSA} - Part 4)

Proof of Honesty *(continued)*

11. P picks uniformly at random a value $\rho_P \in \{0,1\}^{2\ell+s}$.
12. The parties define the following function f, where P gives private input $(\tilde{p}_P, \tilde{q}_P, K_P, \bar{r}_P, \rho_P)$, V gives private input $(\tilde{p}_V, \tilde{q}_V, K_V, \bar{r}_V)$. Let $\sigma = s_P + s_V \mod \mathcal{P}$, based on the values from step 2 of the biprimality test, where $\mathcal{P}$ is the smallest prime larger than $2^{2\ell+s+2}$. They both give public input $(N, e, c_P, r_P, c_V, r_V, \sigma, H_{\tilde{p}_P}, H_{\tilde{q}_P}, H_{\bar{r}_P}, H_{\tilde{p}_V}, H_{\tilde{q}_V}, H_{\bar{r}_V}, \{b_j, v_j, H_{t_j}\}_{i\in[s]})$:

$$
\begin{aligned}
w &:= N - 5 - 4(\tilde{p}_P + \tilde{q}_P + \tilde{p}_V + \tilde{q}_V) \mod e \ , \\
\chi &:= (H_{\tilde{p}_P} \overset{?}{=} \mathsf{AES}_{K_P}(\tilde{p}_P)) \ \wedge \ (H_{\tilde{q}_P} \overset{?}{=} \mathsf{AES}_{K_P}(\tilde{q}_P)) \\
&\quad \wedge (c_P \overset{?}{=} \mathsf{AES}_{\mathsf{AES}_{r_P}(K_P)}(0)) \ \wedge \ (H_{\bar{r}_P} \overset{?}{=} \mathsf{AES}_{K_P}(\bar{r}_P)) \\
&\quad \wedge (H_{\tilde{p}_V} \overset{?}{=} \mathsf{AES}_{K_V}(\tilde{p}_V)) \ \wedge \ (H_{\tilde{q}_V} \overset{?}{=} \mathsf{AES}_{K_V}(\tilde{q}_V)) \\
&\quad \wedge (c_V \overset{?}{=} \mathsf{AES}_{\mathsf{AES}_{r_V}(K_V)}(0)) \ \wedge \ (H_{\bar{r}_V} \overset{?}{=} \mathsf{AES}_{K_V}(\bar{r}_V)) \\
&\quad \wedge (\forall j \in [s] : \mathsf{AES}_{K_P}(v_j + b_j \cdot (\tilde{p}_P + \tilde{q}_P)) \overset{?}{=} H_{t_j}) \\
&\quad \wedge (N \overset{?}{=} (4(\tilde{p}_V + \tilde{p}_P) + 3) \cdot (4(\tilde{q}_V + \tilde{q}_P) + 3)) \\
&\quad \wedge \sigma \overset{?}{=} (\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 5) \\
&\quad \wedge w \neq 0
\end{aligned}
$$

$$
\begin{aligned}
\text{if } V = P_1 : d_V &:= \left\lfloor \frac{-(w^{-1} \mod e) \cdot (N - 5 - 4\tilde{p}_1 - 4\tilde{q}_1) + 1}{e} \right\rfloor \\
\text{else} : d_V &:= \left\lfloor \frac{-(w^{-1} \mod e) \cdot (-4\tilde{p}_2 - 4\tilde{q}_2)}{e} \right\rfloor \\
\text{if } \chi = 1 : \bar{d}_V &:= d_V - \rho_P \\
\text{else } : \bar{d}_V &= \perp \\
&\text{Output } (\chi, \bar{d}_V) \text{ to } V \text{ and } (\perp) \text{ to } P
\end{aligned}
$$

13. The parties then use $\mathcal{F}_{\mathsf{2PC}}$ to compute the output of f and abort if $\chi = 0$.[a]

Generate Shared Key

1. P_1 computes and outputs $d_1 = \bar{d}_1 + \rho_1$.
2. If $e| - 4\tilde{p}_2 - 4\tilde{q}_2$ then P_2 computes and outputs $d_2 = \bar{d}_2 + \rho_2$, else it computes and outputs $d_2 = \bar{d}_2 + \rho_2 + 1$.

[a] This step must be done in parallel for both the cases where P_1 is the prover and P_1 the verifier.

Protocol for maliciously secure RSA key generation.

party without the protocol aborting, no matter if P_1 or P_2 is malicious. If P_2 is malicious it may choose to *try* to learn $(1+\epsilon)x$ bits of the honest party's prime shares for a small $\epsilon \leq 1$ with probability at most 2^{-x}. However, if it is unlucky and does not learn the extra bits then it will not be allowed to use a modulus based on the prime it tried to get some leakage on.

We now argue that this leakage is not an issue, neither in theory nor in practice. For the theoretical part, assume learning some extra bits on the honest party's prime shares would give the adversary a non-negligible advantage in finding the primes of the modulus. This would then mean that there exists a polytime algorithm breaking the security of RSA with non-negligible probability by simply exhaustively guessing what the leaked bits are and then running the adversary algorithm on each of the guesses. Thus if the amount of leaked bits is $O(\text{polylog}(\kappa))$, for the computational security parameter κ, then such an algorithm would also be polytime, and cannot exist under the assumption that RSA is secure. So from a theoretical point of view, we only need to argue that the leakage is $O(\text{polylog}(\kappa))$. To do so first notice that B_1 is a constant tweaked for efficiency. But for concreteness assume it to be somewhere between two constants, e.g., 31 and 3181.[4] Since B_1 is a constant it also means that the leakage is constant and thus $O(1) \in O(\text{polylog}(\kappa))$.

However, if the concrete constant is greater than κ this is not actually saying much, since we then allow a specific value greater than 2^κ time to be polynomial in κ. However, it turns out that for $B_1 = 31$ the exact leakage is only 3.4 bits and for $B_1 = 3181$ it is 5.7 bits.

Formally we prove the following theorem in the full version.

Theorem 3.11. *The protocol Π_{RSA} in Figures 3.7, 3.8, 3.9 and 3.10 securely realizes the ideal functionality $\mathcal{F}_{\mathsf{RSA}}$ in Figure 3.12 against a static and malicious adversary in the $\mathcal{F}_{\ddot{\mathsf{O}}\mathsf{T}}$-, $\mathcal{F}_{\mathsf{CT}}$-, $\mathcal{F}_{\mathsf{ZK}}$-,$\mathcal{F}_{\mathsf{2PC}}$-hybrid model assuming* AES *is IND-CPA and a PRP on the first block per encryption.*

3.4 Outline of Proof

Efficient malicious security One of the reasons we are able to achieve malicious security in such an efficient manner is because of our unorthodox ideal functionality. In particular, by giving the adversary the power to discard some valid moduli, we can prove our protocol secure using a simulation argument *without* having to simulate the honest party's shares of potentially valid moduli discarded throughout the protocol. This means, that we only need to simulate for the candidate N and its shares p_1, q_1, p_2, q_2 that actually get accepted as an output of the protocol.

Another key reason for our efficiency improvements is the fact that almost all of the protocol is executed in a "strong" semi-honest manner. By this we mean that only privacy is guaranteed when a party is acting maliciously, but correctness is not. This makes checking a candidate modulus N much more efficient than if full malicious security was required. At the end of the protocol, full malicious security is ensured for a candidate N by the parties proving that they have executed the protocol correctly.

[4] We find it unrealistic that the a greater or smaller choice will be yield a more efficient execution of our protocol.

FIGURE 3.12 ($\mathcal{F}_{\mathsf{RSA}}$)

Upon query of bitlength 2ℓ and public exponent e from parties P_1 and P_2 proceed as functionality $\mathcal{F}_{\mathsf{RSA\text{-}semi}}$ in Figure 3.5. Otherwise, letting $P_{\mathcal{I}}$ for $\mathcal{I} \in \{1, 2\}$ denote the corrupt party, the functionality initializes an empty set J and a dictionary C mapping IDs to a tuple of elements. Allow the adversary to execute any combination of the following commands:

Sample. On input $(j, p_{\mathcal{I},j})$ where $j \notin J$, $p_{\mathcal{I},j} \leq 2^{\ell-1}$ and $p_{\mathcal{I},j} \equiv 3 \mod 4$ if $\mathcal{I} = 1$ or $p_{\mathcal{I},j} \equiv 0 \mod 4$ if $\mathcal{I} = 2$: select a random value $p_{3-\mathcal{I},j}$ of $\ell - 1$ bits, under the constraint that $p_j = p_{1,j} + p_{2,j} \equiv 3 \mod 4$ is prime and $\gcd(e, p_j - 1) = 1$. Add j to J.

Leak. For each $j \in J$ and for each $\beta \leq B_1$ where β is prime, let $P_{\mathcal{I}}$ input a value $a_{j,\beta}$ and if $\mathcal{I} = 1$ return a bit indicating if $a_{j,\beta} \neq -p_{2,j} \mod \beta$. If $\mathcal{I} = 2$ return a bit indicating if $a_{j,\beta} \neq p_{1,j} \mod \beta$ and set $J = J \backslash \{j\}$ if $a_{j,\beta} = p_{1,j} \mod \beta$.

Construct. On input $(j, j', w'_{\mathcal{I}})$ from $P_{\mathcal{I}}$ where $j, j' \in J$ but $j, j' \notin C$ then compute
$d = e^{-1} \mod (p_j - 1)(p_{j'} - 1)$. Pick a random integer $d_1 \in [2^{2\ell+s}]$ and set $d_2 = d - d_1$. Return $(N = p_j \cdot p_{j'}, p_{\mathcal{I}}, d_{\mathcal{I}})$ to $P_{\mathcal{I}}$ and set $C[j] = C[j'] = (N, d_{3-\mathcal{I}}, j, j')$ and $J = J \backslash \{j, j'\}$. If $P_{\mathcal{I}}$ returns a value $w'_{\mathcal{I},j,j'} \in [0, e-1]$, proceed as follows: If $\mathcal{I} = 1$ notify P_1 whether $w'_{1,j,j'} = p_{2,j} + p_{2,j'} \mod e$ or not. If instead $\mathcal{I} = 2$ then notify P_2 whether $w'_{2,j,j'} = N + 1 - (p_{1,j} + p_{1,j'}) \mod e$ or not. Furthermore, if $\mathcal{I} = 2$ and $\top$ was returned set $C = C \backslash \{j, j'\}$.

Select. On the first input (j, j') with $j, j' \in C$ from $P_{\mathcal{I}}$, where $C[j] = (N, d_{3-\mathcal{I}}, j, j')$ send $(N, p_{3-\mathcal{I},j}, q_{3-\mathcal{I},j}, d_{3-\mathcal{I}})$ to $P_{3-\mathcal{I}}$ and stop accepting commands.

Abort. If $P_{\mathcal{I}}$ inputs $\perp$ at any time then output $\perp$ to both parties and abort.

Ideal functionality for generating shared RSA key

The simulator. With these observations about the efficiency of the protocol in mind, we see that the overall strategy for our simulator is as follows, assuming w.l.o.g. that $P_{\mathcal{I}}$ is the honest party and $P_{3-\mathcal{I}}$ is corrupted.

For the *Setup* phase the simulator simply emulates the honest party's choice of key $K_{\mathcal{I}}$ by sampling it at random. The reason this is sufficient is that because AES is a permutation and $K_{\mathcal{I}}$ is random, thus $\mathsf{AES}_{r_{\mathcal{I}}}(K_{\mathcal{I}})$ is random in the view of the adversary. The crucial thing to notice is that nothing is leaked about this when using it as key in the second encryption under the PRP property. We do strictly need the second encryption since the encryption key $r_{\mathcal{I}}$ is public, thus if we didn't have the second encryption an adversary could decrypt and learn $K_{\mathcal{I}}$! Regarding the zero-knowledge proof we notice that the simulator can extract the adversary's input $K'_{3-\mathcal{I}}$. We notice that the simulator can emulate $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ by verifying $K'_{3-\mathcal{I}}$ in the computation of $c_{3-\mathcal{I}}$. Again we rely on AES being a PRP to ensure that if $K'_{3-\mathcal{I}}$ is not the value used in computing $c_{3-\mathcal{I}}$ then the check will always fail because it would require the adversary to find $K'_{3-\mathcal{I}} \neq K_{3-\mathcal{I}}$ s.t. $\mathsf{AES}_{r_{3-\mathcal{I}}}(K'_{3-\mathcal{I}}) = \mathsf{AES}_{r_{3-\mathcal{I}}}(K_{3-\mathcal{I}})$. Thus the adversary is committed to some specific key $K_{3-\mathcal{I}}$ extracted by the simulator.

For *Candidate Generation* the simulator starts by sampling a random value $\tilde{p}_{\mathcal{I}}$ and extracts the malicious party's share $\tilde{p}_{3-\mathcal{I}}$ from its "commitment" $H_{\tilde{p}_{3-\mathcal{I}}}$, since the simulator knows the key $K_{3-\mathcal{I}}$. Furthermore, since we use AES in a mode s.t. it is IND-CPA secure the adversary cannot distinguish between the values $H_{\tilde{p}_{\mathcal{I}}}$ simulated or the values sent in the real protocol.

Next see that if $4(\tilde{p}_1 + \tilde{p}_2) + 3$ is *not* a prime, then the simulator will emulate the rest of the protocol using the random value it sampled. This simulation will be statistically indistinguishable from the real world since the simulator and the honest party both sample at random and follow the protocol. Furthermore, since the shares don't add up to a prime, any modulus based on this will never be output in the real protocol since the *proof of honesty* will discover if N is not a biprime.

On the other hand, if the shares *do* sum to a prime then the simulator uses *sample* on the ideal functionality $\mathcal{F}_{\mathsf{RSA}}$ to construct a prime based on the malicious party's share $\tilde{p}_{3-\mathcal{I}}$. It then simulates based on the value extracted from the malicious party. Specifically for the OT-based trial division, the simulator extracts the messages of the malicious party and uses these as input to *leak* on the ideal functionality. This allows the simulator to learn whether the adversary's input to the trial division plus the true and internal random value held by the ideal functionality is divisible by β.

To simulate construction of a modulus, we first consider a hybrid functionality, which is the same as $\mathcal{F}_{\mathsf{RSA}}$, except that a command *full-leak* is added. This command allows the simulator to learn the honest party's shares of a prime candidate, under the constraint that it is not used in the RSA that key the functionality outputs. It is easy to see that adding this method to the functionality does not give the adversary more power, since it can only learn the honest party's shares of primes which are independent of the output.

With this expanded, hybrid version of $\mathcal{F}_{\mathsf{RSA}}$ in place, the simulator emulates the construction of a modulus by first checking if one of the candidate values were prime and the other was not. In this case it uses *full-leak* to learn the value that is prime and then simulates the rest of the protocol like an honest party. This will be statistically indistinguishable from the real execution since the modulus will never be used as output since it is not a biprime and so will be discarded, at the latest, in the *proof of honesty* phase.

However, if both candidate values are marked as prime the simulator simulates the extended Gilboa protocol for construction of the modulus. It does so by extracting the malicious party's input to the calls to $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$. Based on this it can simulate the values of the honest party. This is pretty straightforward, but what is key is that no info on the honest party's prime shares is leaked to the adversary in case of a selective failure attack (when the adversary is the sender in $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$). To see this, first notice that selecting $h_1, \ldots, h_{2\ell+2s}, g$ at random and computing $g + \sum_{i \in [2\ell+2s]} h_i \cdot d_i \mod \mathcal{P}$ is in fact a 2-universal hash function. This implies, using some observations by Ishai *et al.* [IPS09], that whether these values are picked at random s.t. they hash to the true input or are just random, is 2^{-s} indistinguishable. Thus extending the function with $h_{2\ell+2s+1}, \ldots g_{2\ell+3s}$, allows the adversary to learn s of the bits d_i without affecting the indistinguishable result. Since s is the statistical security parameter and each d_i is picked at random, this implies that the adversary cannot learn anything non-negligible.

The proof of security of the remaining steps is quite straightforward. For *trial division* the simulator basically acts as an honest party since all operations are local. For the *biprimality test* the simulation is also easy. For step 1, it is simply following the proof by Boneh and Franklin [BF01], for step 2, simulation can be done using the same approach as for the Gilboa protocol. Regarding *proof of honesty* the simulation follows the steps for the simulation of the biprimality test and uses the emulation of the coin-tossing functionality to learn what challenge it needs to answer and can thus adjust the value sent to the verifier that will make the proof accept. Finally the *key generation* is also unsurprising as all the computations are local.

4 Instantiation

Optimizations. *Fail-fast.* It is possible to limit the amount of tests carried out on composite candidates, in all of the OT-based trial division, the second trial division and the biprimality test, by simply employing a *fail-fast* approach. That is, to simply break the loop of any of these tests, as soon as a candidate fails. This leads to significantly fewer tests, as in all three tests a false positive is more likely to be discovered in the beginning of the test. (For example, a third of the candidates are likely to fail in the first trial division test, which checks for divisibility by 3.)

Maximum runtime. In the malicious protocol an adversary can cheat in such a way that a legitimate candidate (either prime or modulus) gets rejected. In particular this means that the adversary could make the protocol run forever. For example, if he tries to learn 1024 bits of the honest party's shares by cheating then we expect to discard 2^{1023} good candidate moduli! Thus, tail-bounds for the choice of parameters should be computed s.t. the protocol will abort once it has considered more candidate values than would be needed to find a valid modulus e.w.p. 2^{-s}. In fact, it is strictly needed in order to limit the maximum possible leakage from selective failure attacks.

Synchronous execution. To ensure that neither party P_1 nor party P_2 sits idle at any point in time of the execution of the protocol, we can have them exchange roles for every other candidate. Thus, every party performs both roles, but on two different candidates at the same time, throughout the execution of the protocol. For example, while Alice executes as P_1 in the *candidate generation* on one candidate, she simultaneously executes the candidate generation as P_2 for another candidate. Similarly for Bob. The result of this is that no party will have to wait for the other party to complete a step as they will both do the same amount of computation in each step.

Leaky two-party computation. We already discussed in Section 3.3 how leaking a few bits of information on the honest party's prime shares does not compromise the security of the protocol. Along the same line, we can make the observation that learning a predicate on the honest party's share will in expectation not give more than a single bit of information to the adversary. In particular it can learn at most x bits of information with probability at most 2^{-x}. This is the same leakage that is already allowed to P_2, and thus allowing this would not yield any significant change to the leakage of our protocol. This means that it can suffice to construct only two garbled circuits to implement $\mathcal{F}_{\mathsf{2PC}}$ by using the *dual-execution* approach [MF06]. This is compared to the s garbled circuits needed in the general case where no leakage is allowed [Lin16].

Constant rounds. We note that the way our protocols $\mathit{\Pi}_{\mathsf{RSA\text{-}semi}}$ and $\mathit{\Pi}_{\mathsf{RSA}}$ are presented in Fig. 3.2 and Fig. 3.3, respectively Fig. 3.7, Fig. 3.8, Fig. 3.9 and Fig. 3.10 does not give constant time. This is because they are expressed iteratively s.t. candidate primes are sampled until a pair passing all the tests is found. However it is possible to simply execute each step of the protocols once for many candidates in parallel. This is because, based on the Prime Number Theorem, we can find the probability of a pair of candidates being good. This allows us to compute the amount of candidate values needed to ensure that a good modulus is found, except with negligible probability. Unfortunately this will in most situations lead to many candidate values being constructed unnecessarily. For this reason it is in practice more desirable to construct batches of candidates in parallel instead to avoid doing a lot of unnecessary work, yet still limit the amount of round of communication.

Efficiency Comparison. We here try to compare the efficiency of our protocol with previous work. This is done in Table 1.

With regards to more concrete efficiency we recall that both our protocols and previous work have the same type of phases, working on randomly sampled candidates in a pipelined manner. Because of this feature, all protocols limit the amount of unsuitable candidates passing through to the expensive phases, by employing trial division. This leads to fewer executions of expensive phases and thus to greater concrete efficiency. In some protocols this filtering is applied both to individual prime candidates *and* to candidate moduli, leading to minimal executions of the expensive phases. Unfortunately this is not possible in all protocols. For this reason we also show in Table 1 which protocols manage to improve the expected execution time by doing trial division of the prime candidates, respectively the moduli.

To give a proper idea of the efficiency of the different protocols we must also consider the asymptotics. However, because of the diversity in primitives used in the previous protocols, and in the different phases, we try to do this by comparing the computational bit complexity. Furthermore to make the comparison as fair as possible we assume the best possible implementations available *today* are used for underlying primitives. In particular we assume an efficient OT extension is used for OTs [KOS15].

Based on the table we can make the following conclusions regarding the efficiency of our protocols. First, considering the semi-honest protocol; we see that

Scheme	Assumptions	Dishonest majority	Malicious secure	Prime trial division	Modulus trial division	Rounds	Amount of candidates	Candidate generation	Construct modulus	(Bi)primality test	Leakage
Our result*	IND-CPA, $\mathcal{F}_{\mathsf{OT}}$, $\mathcal{F}_{\mathsf{CT}}$	✓	✓	✓	✓	$\boldsymbol{O(1)}$	$O(\ell^2/\log^2(\ell))$	$\boldsymbol{O(\ell)}$	$\boldsymbol{O(\ell^2)}$	$O(s \cdot \ell^3)$	$\tau+2$
[BF01]	**None**	✗	✗	✓	✓	$\boldsymbol{O(1)}$	$O(\ell^2/\log^2(\ell))$	$\boldsymbol{O(\ell)}$	$\boldsymbol{O(\ell^2)}$	$O(s \cdot \ell^3)$	**2**
[FMY98]	DL	✗	✓	✗	✓	$\boldsymbol{O(1)}$	$O(\ell^2/\log^2(\ell))$	$O(\ell^3)$	$O(\ell^3)$	$O(s^2 \cdot \ell^3)$	**2**
[PS98]†	$\mathcal{F}_{\mathsf{OT}}$	✓	✓	✓	✓	$\boldsymbol{O(1)}$	$O(\ell^2/\log^2(\ell))$	$\boldsymbol{O(\ell)}$	$\boldsymbol{O(\ell^2)}$	?	$\tau+2$
[Gil99]	PRG, $\mathcal{F}_{\mathsf{OT}}$	✓	✗	✗	✓	$\boldsymbol{O(1)}$	$O(\ell^2/\log^2(\ell))$	$\boldsymbol{O(\ell)}$	$\boldsymbol{O(\ell^2)}$	$O(s \cdot \ell^3)$	**2**
[ACS02]	**None**	✗	✗	✓	✗	$O(\ell)$	$\boldsymbol{O(\ell/\log(\ell))}$	$\boldsymbol{O(\ell)}$	$\boldsymbol{O(\ell^2)}$	$O(s \cdot \ell^3)$	**2**
[DM10]	CRS, Strong RSA	✗	✓	✓	✓	$\boldsymbol{O(1)}$	$O(\ell^2/\log^2(\ell))$	$O(\ell^3)$	$O(\ell^3)$	$\boldsymbol{o(s \cdot \ell^3)}$	**2**
[HMRT12]	DCR, DDH	✓	✓	✓	✓	$O(1)^{\ddagger}$	$O(\ell^2/\log^2 \ell)$	$O(\ell^3)$	$O(\ell^3)$	$O(s \cdot \ell^3)$	**2**

Table 1. Comparison of the different protocols for distributed RSA key generation. The best possible values are highlighted in bold. All values assume a constant, and minimal, amount of participating parties; i.e. 2 or 3. The column *Amount of candidates* expresses the expected amount of random candidates that must be generated before finding a suitable modulus. The column *Candidate generation* expresses the computational bit complexity required to construct a *single* candidate prime. The column *Construct modulus* expresses the computational bit complexity required to construct a *single* potential modulus, based on two prime candidates. The column *(Bi)primality test* expresses the computational bit complexity required to verify that a *single* prime candidate is prime except with negligible probability or (depending on the protocol) to verify that a *single* modulus is the product of two primes except with negligible probability. The column *Leakage* expresses how many bits of information of the honest party's shares of the primes that is leaked to the adversary. Here τ means that $\sum_{\beta \in B_1} \log\left(\frac{\beta}{\beta-1}\right)$ bits can be leaked to a malicious adversary. Furthermore, the adversary is allowed to pick a probability x with which it learns $(1+\epsilon)x$ extra bits. However, if the adversary does not learn the extra bits then the honest party learns that the adversary has acted maliciously.

*: For the malicious protocol $O(s^2 \cdot \ell^3)$ operations are executed *once* per successful key pair generation.

†: The authors do not describe how to ensure biprimality in case of a malicious adversary.

‡: Constant round on average.

the only real competition is Algesheimer *et al.* as they expect to test asymptotically fewer prime candidates than us. However, this comes at the price of a large amount of rounds requires, making it challenging to use efficiently over the Internet. Furthermore, unlike our protocol, they require an honest majority making it possible for them to leverage efficient information theoretic constructions.

For the malicious security model we see that the only real competition lies in the work of Poupard and Stern [PS98]. However, we note that they don't provide a full maliciously secure protocol. In particular they do not describe how to do a biprimality test secure against a malicious and dishonest majority. Thus the only other protocol considering the same setting as us is the one by Hazay *et al.*. This is also the newest of the schemes and is considered the current state-of-the-art in this setting. However, this protocol requires asymptotically more operations for candidate generation, construction of modulus and the biprimality test.

Implementation. Below we outline the concrete implementation choices we made. We implement AES in counter mode, using AES-NI, with $\kappa = 128$ bit keys. For 1-out-of-2 OT (needed during the *Construct Modulus* phase) we use the maliciously secure OT extension of Keller *et al.* [KOS15]. For the base OTs we use the protocol of Peikert *et al.* [PVW08] and for the internal PRG we use AES-NI with the seed as key, in counter mode. For the random 1-out-of-β OT we use the random 1-out-of-2 OT above using the protocol of Naor and Pinkas [NP99]. For the coin-tossing we use the standard "commit to randomness and then open" approach.

In the *Construct Modulus* phase, instead of having each party sample the values $h_{\alpha,1}, \ldots, h_{\alpha.2\ell+3s}, g_\alpha \in \mathbb{Z}_\mathcal{P}$ and send them to the other party, we instead have it sample a seed and generate these values through a PRG. The party then only needs to send the seed to the other party. This saves a large amount of communication complexity without making security compromises.

Our implementation implemented **OT extension** in batches of 8912 OTs. Whenever a batch of OTs is finished, the program calls a procedure which generates a new block of 8192 OTs. Most of the cryptographic operations were implemented using OpenSSL, but big-integer multiplication was implemented in assembler instead of using the OpenSSL implementation for efficiency reasons.

We did not implement yet the **zero-knowledge argument** or the two-party computation since they can be efficiently realized using existing implementations of garbled circuits (such as JustGarble [BHKR13] or TinyGarble [SHS+15]) by using the protocol of Jawurek *et al.* [JKO13] for zero-knowledge and the dual-execution approach [MF06] for the two-party computation. These protocols are only executed *once* in our scheme using well tested implementations, and thus, as is described later in this section, we can safely estimate that the effect on the total run time is marginal.

Experiments. We implemented our maliciously secure protocol and ran experiments on Azure, using Intel Xeon E5-2673 v.4 - 2.3Ghz machines with 64Gb RAM, connected by a 40.0 Gbps network. are pretty strong servers).

We used the code to run 50 computations of a shared 2048 bit modulus, and computed the average run time. The results are as follows:

– With a single threaded execution, the average run time was 134 seconds.
– With four threads, the average run time was 39.1 seconds.
– With eight threads, the average run time was 35 seconds.

The run times showed a high variance (similar to the results of the implementation reported by Hazay *et al.* [HMRT12] for their protocol). For the single thread execution, the average run time was 134 seconds while the median run time was 84.9 seconds (the fastest execution took 8.2 sec and the slowest execution took 542 sec).

Focusing on the single thread execution, we measured the time consumed by different major parts of the protocol. The preparation of the OT extension tables took on average 12% of the run time, the multiplication protocol computing N took 66%, and the biprimality test took 7%. (These percentages were quite stable across all executions and showed little variance.) Overall these parts took 85% of the total run time. The bulk of the time was consumed by the secure multiplication protocol. In that protocol, most time was spent on computing the values z_1^α, z_2^α (line 4 in Fig. 3.8). This is not surprising since each of these computations computes $2\ell + 3s = 2168$ bignum multiplications.

Note that these numbers exclude the time required to do the zero-knowledge argument of knowledge in *Setup* and the two-party computation in *Proof of Honesty*. The zero-knowledge argument of knowledge requires about 12,000 AND gates (for two AES computations), and our analysis in Appendix A shows that the number of AND gates that need to be evaluated in the circuits of the honesty proof is at most 22 million. We also measured a throughput of computing about 3.2 million AND gates in Yao's protocol on the machines that we were using. Therefore we estimate that implementing these computations using garbled circuits will contribute about 7 seconds to the total time.

Comparing to previous work, the only other competitive protocol (for 2048 bit keys) with implementation work is the one by Hazay *et al.* [HMRT12]. Unfortunately their implementation is not publicly available and thus we are not able to make a comparison on the same hardware. However, we do not that the fastest time they report is 15 min on a 2.3 GHz dual-core Intel desktop, for their *semi-honestly secure* protocol.

References

ACS02. Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO*, pages 417–432, 2002.

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

BHKR13. Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy*, pages 478–492. IEEE Computer Society, 2013.

CDN01. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In *EUROCRYPT*, pages 280–299, 2001.

DM10. Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In *TCC*, pages 183–200, 2010.

FMY98. Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed rsa-key generation. In *STOC*, pages 663–672, 1998.

Gav12. Gérald Gavin. RSA modulus generation in the two-party case. *IACR Cryptology ePrint Archive*, 2012:336, 2012.

Gil99. Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 116–129. Springer, 1999.

GMO16. Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In Thorsten Holz and Stefan Savage, editors, *USENIX Security Symposium*, pages 1069–1083. USENIX Association, 2016.

HMRT12. Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA key generation and threshold paillier in the two-party setting. In *CT-RSA*, pages 313–331, 2012.

IPS09. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In *TCC*, pages 294–314, 2009.

JKO13. Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM SIGSAC*, pages 955–966. ACM, 2013.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In *CRYPTO*, pages 724–741, 2015.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM SIGSAC*, pages 830–842. ACM, 2016.

KS06. Mehmet S. Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of Yao's garbled circuit construction. In *In Proceedings of 27th Symposium on Information Theory in the Benelux*, pages 283–290, 2006.

Lin16. Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *J. Cryptology*, 29(2):456–490, 2016.

MF06. Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

NP99. Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *STOC*, pages 245–254. ACM, 1999.

OOS17. Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n OT extension with application to private set intersection. In *CT-RSA*, pages 381–396, 2017.

PS98. Guillaume Poupard and Jacques Stern. Generation of shared RSA keys by two parties. In *ASIACRYPT*, pages 11–24, 1998.

PVW08. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, pages 554–571, 2008.

RSA78. Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

Sch12. Thomas Schneider. *Engineering Secure Two-Party Computation Protocols: Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer Publishing Company, Incorporated, 2012.

Sho00. Victor Shoup. Practical threshold signatures. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000.

SHS$^+$15. Ebrahim M. Songhori, Siam U. Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *IEEE Symposium on Security and Privacy*, pages 411–428. IEEE Computer Society, 2015.

A The Size of the Circuit for the Proof of Honesty

The circuit that is evaluated in the honesty proof contains the following components:

- AES computation: Four AES encryptions of single blocks (for verifying the commitments to the keys in Step 2), and $s+6$ encryptions of values of length $|N|/2+s$ bits. For $|N| = 2048$ bits this translates to $4 + 46 \cdot 9 = 418$ AES blocks. With an AES circuit of size 6000 AND gates, this translates to $2.5M$ gates.
- Multiplications: The circuit computes two multiplications where each of the inputs is $|N|/2$ bits long. For $|N| = 2048$ using Karatsuba multiplication this takes 1.040M gates [Sch12].
- Division: The circuit computes a division of an $|N|$ bit value by $e = 2^{16} + 1$. For $|N| = 2048$ the size of this component is about 900K AND gates.
- Computing the inverse of an $|N|$ bit number modulo e: This operation is done by first reducing the number modulo e and then raising the result to the power of $e - 2$ modulo e. The first step takes about 900K AND gates, and the second step takes 64K AND gates.
- Comparisons, additions, and multiplications by small numbers (smaller than e): These operations are implemented with a number of gates that is linear in the size of their inputs, and is therefore quite small. We estimate an upper bound of 100K for the number of AND gates in all these operations.

The total size of the circuit is therefore less than 5.5M AND gates. The honesty proof should be carried out by each of the parties, and dual execution requires computing the circuit twice. Therefore the total number of AND gates computed is less than 22M.