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

(Full Version)

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.

* This article is based on an earlier article: [FLOP18], ©IACR 2018 10.1007/978-3-319-96881-0_12. This version provides more detail and fixes an issue with step 2 of the Biprimality Test sub-procedure, Fig 3.4.

** 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.

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 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.

1.1 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.

1.2 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.

1.3 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.

1.4 Related Work

Work on efficient [4] 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 equivocal 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

[4] In theory this could be achieved using any MPC protocol implementing regular RSA key generation and simply additively sharing the secret key at the end. However, this would be too inefficient in practice.

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.

1.5 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 protocol is more than an order of magnitude faster than its most efficient *semi-honest* competitor [HMRT12]. In particular, our eight threaded implementation takes on average less than 44 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. 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.

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 generation and construction of modulus phases. The functionality of random OT is described in Fig. 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. OT extension was suggested by Beaver [Bea96], followed by a practical construction by Ishai *et al.* [IKNP03]. Extremely efficient constructions in the semi-honest setting were described by Asharov *et al.* [ALSZ13] and Kolesnikov and Kumaresan [KK13]. Several works designed efficient protocols for 1-out-of-2 OT extension with security against malicious adversaries [NNOB12,ALSZ15,KOS15]. We are interested in the more general random 1-out-of-β OT extension, with security against malicious adversaries (this level of security will be needed in Section 3.3). A protocol for this task was described by Orrù *et al.* [OOS17], with an overhead which is just slightly larger than in the semi-honest case. 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, as the sender can simply use the random messages as one-time pads to its true messages. Thus if the sender has chosen messages $a_0, \ldots, a_{\beta-1}$ and receives random messages $m_0, \ldots, m_{\beta-1}$ from the functionality. It simply sends the encryptions $a_0 \oplus m_0, \ldots, a_{\beta-1} \oplus m_{\beta-1}$. The receiver can then decrypt the true message by computing $a_i = (a_i \oplus m_i) \oplus m_i$.

We will sometimes abuse notation and assume that $\mathcal{F}_{\mathsf{OT}}^{\ell,\beta}$ supports specific messages, using the approach above, by allowing the sender to input the message $(\texttt{transfer}, a_0, \ldots, a_{\beta-1})$, and the receiver receiving message $(\texttt{transfer}, a_i)$.

Finally we note that when $\beta > 2$ we do not need the receiver to learn *all* possible random messages from the OT functionality, but rather only a single one of his choice. This means that our protocol can use the 1-out-of-β OT extension of Orrù *et al.* [OOS17]. This gives us sublinear complexity in β for each OT.

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 (`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 (`transfer`, $m_0, \ldots, m_{\beta-1}$) to `snd` and (`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

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 Fig. 2.2.

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

Functionality interacts with P_1 and P_2. Upon receiving (`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 (`random`, x) to both parties.

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

Ideal functionality for coin-tossing

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 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 Fig. 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 [IKOS09,GMO16].

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 $M_L : \{0,1\}^* \times \{0,1\}^* \to \{\top, \bot\}$. It proceeds as follows:

- On input $(\texttt{prove}, x, w)$ from P and $(\texttt{verify}, x')$ 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

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 Fig. 2.4. Such a functionality can be implemented efficiently in constant rounds using a garbled circuit protocol [Lin16].

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

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

Output: After having received messages $(\texttt{input}, \cdot)$ 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 $(\texttt{output}, y_1)$ to P_1 and $(\texttt{output}, y_2)$ 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 $(\texttt{deliver})$ then send $y_{3-\mathcal{I}}$ to party $P_{\mathcal{I}}$, otherwise if $P_{3-\mathcal{I}}$ returns the message $(\texttt{abort})$ then output $(\texttt{abort})$ to $P_{\mathcal{I}}$.

Ideal functionality for two-party computation

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. Furthermore we abuse notation and assume that $\bmod$ computes the remainder when it is not in an equivalence.

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 considerably 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 1.4. 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 Fig. 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 Fig. 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.

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

$$N = (p_1 + p_2)(q_1 + q_2) = p_1q_1 + p_2q_2 + p_1q_2 + p_2q_1.$$

FIGURE 3.1 (Div-OT)

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 (`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

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

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

These will therefore be the two outputs of the multiplication protocol. The protocol can be implemented using *random* 1-out-of-2 OT:

1. The two parties run a random OT where the receiver's input is b_i. The sender learns $s_{i,0}, s_{i,1}$. The receiver learns s_{i,b_i}, where the strings $s_{i,0}, s_{i,1}$ are in $[0, 2^{2\ell} - 1]$.
2. The sender sends the string $\hat{s} = s_{i,0} + s_{i,1} + a \bmod 2^{2\ell}$ to the receiver.
3. The sender defines $r_i = s_{i,0}$.
4. If $b_i = 0$ then the receiver learns $s_{i,0}$ and sets its output to be $c_i = s_{i,0}$.
5. If $b_i = 1$ then the receiver learns $s_{i,1}$ and sets its output to be $c_i = \hat{s} - s_{i,1} \bmod 2^{2\ell}$. Note that this value is equal to $a + s_{i,0} \bmod 2^{2\ell}$.

The communication overhead of this protocol is that of the random OT protocol, plus sending a 2ℓ-bit string from the sender to the receiver. Since we run this protocol for each bit, we implement it using random OT extension.

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

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 Div-OT in Fig. 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 $(\texttt{transfer}, c_{0,\alpha,i}, c_{1,\alpha,i})$.
   (c) P_1 inputs $(\texttt{receive}, \alpha_{1,i})$ 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 $(\texttt{deliver}, c_{\alpha_{1,i},i})$ 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

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 3.3 (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 Fig. 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

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 Fig. 3.4. By slightly modified, we mean that step 2), which ensures that $\gcd(p + q - 1, N) = 1$, is computed without the need of doing operations in the group $(\mathbb{Z}_N[x]/(x^2+1))^*/\mathbb{Z}_N^*$. Specifically we leverage the steps of "Construct Modulus" for doing oblivious multiplication by computing an additive secret sharing s_1, s_2 of $s = r \cdot (p + q - 1)$ for a random $r \in \mathbb{Z}_N$ modulo $2^{3\ell}$. It is then possible for the parties to check $\gcd(s, N) = 1$ in plain which is sufficient to ensure $\gcd(p + q - 1, N) = 1$. An issue is however, that if $r \cdot (p + q - 1)$ is not reduced modulo N, then even for a large r, the value s might leak information about $p + q$. But it is not possible for the parties to simply reduce their shares modulo N before exchanging them since s is a 3ℓ bit number and thus $(s_1 \mod N) + (s_2 \mod N) \neq (s_1 + s_2 \mod 2^{3\ell}) \mod N$ since $2^{3\ell} > N$. Fortunately this is easy to handle; by simply ensuring that the secret sharing of s is over an s-bit *bigger* domain than needed for computing the product $r \cdot (p + q - 1)$. In this case it will hold that $s_1 + s_2 > 2^{3\ell+s}$ except with probability 2^{-s}. This is because $s_1 + s_2 \mod 2^{3\ell+s} < 2^{3\ell}$ and since s_1 and s_2 is a random sharing we get that the probability that the s most significant bits of, say s_1, are all 0 is at most 2^{-s}. If these bits are not all 0 we are necessarily in the

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]
   (a) P_1 chooses a random number $\bar{r}_1 \in \mathbb{Z}_N$ and P_2 chooses a random $\bar{r}_2 \in \mathbb{Z}_N$. (The parties will verify that $\gcd((\bar{r}_1 + \bar{r}_2) \cdot (p+q-1) \mod N, N) = 1$.)
   (b) The parties run a multiplication protocol (the first steps of "Construct Modulus" using modulo $2^{3\ell+s+2}$) where they compute shares α_1, α_2 (known to P_1, P_2 respectively) of $\bar{r}_1 \cdot (p_2+q_2-1) \mod 2^{3\ell+s+2}$, and shares β_1, β_2 of $\bar{r}_2 \cdot (p_1+q_1) \mod 2^{3\ell+s+2}$.
   (c) P_1 sends to P_2 the value $s_1 = \bar{r}_1(p_1+q_1) + (\alpha_1 + \beta_1 \mod 2^{3\ell+s+2}) - 2^{3\ell+s+2} \mod N$.
   (d) P_2 sends to P_1 the value $s_2 = \bar{r}_2(p_2+q_2-1) + \left(\alpha_2 + \beta_2 \mod 2^{3\ell+s+2}\right) \mod N$.
   (e) The parties verify that $\gcd(s_1+s_2, N) = 1$. If this is not the case then they discard the candidate N.

[a] This test is required in order not to run additional tests over the group $\mathbb{T}_N = (\mathbb{Z}_N[x]/(x^2+1))^\times/\mathbb{Z}_N^\times$, which proved to be very slow. See [BF01] Sect. 4.1.

The biprimality test inspired by Boneh and Franklin [BF01]

case where $s_1 + s_2 > 2^{3\ell+s}$ as otherwise the inequality $s_1 + s_2 \mod 2^{3\ell+s} < 2^{3\ell}$ cannot hold. Thus by computing $s_1 - 2^{3\ell+s}$ over the integers, we get an integer sharing of $r \cdot (p+q-1)$ which is then possible to reduce modulo N [5]. The idea of turning a modular secret sharing into an integer secret sharing was previously explained by Algesheimer *et al.* [ACS02].

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 $b = w^{-1} \mod e$. 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 Fig. 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 Fig. 3.5. The functionality closely reflects the specific

[5] We note that for technical reasons our protocol will do the sharing over $3\ell + s + 2$ bits instead of just $3\ell + s$ bits.

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.

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

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

Using this functionality we get the following theorem:

Theorem 3.6. *The protocol $\Pi_{\text{RSA-semi}}$ in Fig. 3.2 and 3.3 securely realizes the ideal functionality $\mathcal{F}_{\text{RSA-semi}}$ in Fig. 3.5 against a static and semi-honest adversary in the $\mathcal{F}_{OT}$-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. [6]
- 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 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

[6] This is in fact not strictly necessary. It is sufficient to only retain privacy under a malicious attack. However, for simplicity we simply assume that is fully maliciously secure.

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. Specifically P_1 computes

$$a_1 = p_1 q_1 + \left(\sum_{i \in [2\ell+3s]} c_{d_{q,i},q,i} \cdot h_{q,i} \right) + \left(\sum_{i \in [2\ell+3s]} c_{d_{p,i},p,i} \cdot h_{p,i} \right) \mod \mathcal{P}$$

and P_2 computes

$$a_2 = p_2q_2 + g_pq_2 + g_qp_2 - \left(\sum_{i\in[2\ell+3s]} c_{0,q,i}\cdot h_{q,i}\right) - \left(\sum_{i\in[2\ell+3s]} c_{0,p,i}\cdot h_{p,i}\right) \mod \mathcal{P}$$

The parties then exchange their values a_1 and a_2 and now the modulus $N = a_1 + a_2 \mod \mathcal{P}$. 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, to verify that the modulus is constructed from the values committed to in *candidate generation*. Second; to generate a random sharing of the private key. Finally, to do a maliciously secure execution of the full biprimality test of Boneh and Franklin [BF01]. 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 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.

To compute the shares of the secret key we let the proving party input some randomness which is used to randomize the verifying party's share of the private key.

To ensure a correctly executed biprimality test, two checks are performed: First to ensure correct execution of step 1) of Fig. 3.4 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. Second, to ensure correct execution of step 2) of Fig. 3.4 we verify that the parties executed the multiplication subprotocol correctly and sent the correct values s_1 and s_2 and in step c and d. This reduces to verifying the correctness of the computation $s_1 + s_2 \equiv (\bar{r}_1 + \bar{r}_2)\cdot(p_1 + p_2 + q_1 + q_2 - 1) \mod N$. Unfortunately we cannot simply release the value $(\bar{r}_1 + \bar{r}_2)\cdot(p_1 + p_2 + q_1 + q_2 - 1)$ in plain as it might leak some information about p_1, p_2, q_1, q_2 when it is not reduced modulo N. To fix this we introduce a new trick: The prover picks a large, random pad $\hat{r}_P$, and the verifier a smallish random pad $\hat{r}_V$. We then secure compute the value

$\delta = \hat{r}_P + \hat{r}_V\,((\bar{r}_1 + \bar{r}_2) \cdot (p_1 + p_2 + q_1 + q_2 - 1) - s_1 - s_2)$ and open this towards the verifier. Now the verifier can compute the modulo N operation in plain on δ, and the prover can do the same on $\hat{r}_P$ and send the remainder to the verifier. The verifier can then check that $\hat{r}_P = \delta \mod N$. If the values s_1, s_2 have been correctly construct it must hold that $s_1 + s_2 \equiv (\bar{r}_1 + \bar{r}_2) \cdot (p_1 + p_2 + q_1 + q_2 - 1) \mod N$ and hence $\hat{r}_V\,((\bar{r}_1 + \bar{r}_2) \cdot (p_1 + p_2 + q_1 + q_2 - 1) - s_1 - s_2) \equiv 0 \mod N$. Thus a random pad on either side of the congruence will give the same remainder when reduced modulo N.

In a bit more detail, letting P denote the prover and V the verifier, the *proof of honesty* proceeds as follows, keeping in mind that $p_1 = 4\tilde{p}_1$, $q_1 = 4\tilde{q}_1$ and $p_2 = 4\tilde{p}_2 + 3$, $q_2 = 4\tilde{q}_2 + 3$:

- Run the step 1) of biprimality test in Fig. 3.4 using a coin-tossing protocol to sample the random values γ_i.
- P picks s random values $t_1, \ldots, t_s \in \mathbb{Z}_{2^{\ell+s-2}}$ and computes an AES encryption of each of these.
- Based on each of the random values γ_i from the biprimality test, P computes all s^2 combinations $\gamma_i^{t_j} \mod N$ and sends these to V.
- The parties use coin-tossing to pick s bits $b_1, \ldots, b_s \in \{0, 1\}$.
- For each $j \in [s]$, if b_j is 0 then P sends $t_j \in \mathbb{Z}_{2^{\ell+s-2}}$ to V, otherwise it sends $-\tilde{p}_P - \tilde{q}_P + t_j \in \mathbb{Z}_{2^{\ell+s-2}}$.
- Using the values from the biprimality test V verifies that $\gamma_i^{t_j} \mod N$ or $\gamma_i^{-\tilde{p}_P - \tilde{q}_P + t_j} \mod N$ (depending on its choice b_j) is as expected.
- Finally they use the two-party computation to verify the multiplication carried out in step 2) of the biprimality test in Fig. 3.4, along with the AES-based commitments of $\tilde{p}_P, \tilde{q}_P, \bar{r}_P$, $\bar{r}_V$, t_j for $j \in [s]$ and that $\phi(N) \mod e \neq 0$.[7] This two-party computation furthermore computes a random sharing of the secret key d and gives each party their share.
- If any checks above fails then V aborts.

We formally describe the full protocol in Fig. 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 Fig. 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

[7] This means that $\gcd(e, \phi(N)) = 1$ since e is prime. This is done to ensure that it is possible to compute the private key.

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 Div-OT in Fig. 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.

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 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

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 $\mathcal{F}_{\mathsf{OT}}^{2\ell,2}$ with input $(\texttt{transfer}, c_{0,\alpha,i}, c_{1,\alpha,i})$.
   (c) P_1 inputs $(\texttt{receive}, d_{\alpha,i})$ to $\mathcal{F}_{\mathsf{OT}}^{2\ell,2}$. 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.

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

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 Fig. 3.4 using $3\ell + 4s + 2$ calls to $\mathcal{F}_{\mathsf{OT}}^{3\ell+s+2,2}$ where $P_{\mathcal{I}}$ acts as the receiver in all the OTs when it has input $\bar{r}_{\mathcal{I}} \in \mathbb{Z}_N$ when computing "Construct Modulus". After step 2.a $P_{\mathcal{I}}$ sends $H_{\bar{r}_{\mathcal{I}}} = \mathsf{AES}_{K_{\mathcal{I}}}(\bar{r}_{\mathcal{I}})$.

Proof of Honesty

The parties call $(\texttt{toss}, \mathbb{Z}_N^\times)$ 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 $(\texttt{toss}, \{0,1\})$ 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

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

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

Protocol for maliciously secure RSA key generation.

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 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.

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. P picks a random value $\hat{r}_P \in \{0,1\}^{3\ell+2s+2}$ and V picks a random value $\hat{r}_V \in \{0,1\}^s$. The parties define the following function f, where P gives private input $(\tilde{p}_P, \tilde{q}_P, K_P, \bar{r}_P, \rho_P, \hat{r}_P)$, V gives private input $(\tilde{p}_V, \tilde{q}_V, K_V, \bar{r}_V, \hat{r}_V)$. Let $\sigma = s_P + s_V \mod N$, based on the values from step 2 of the biprimality test. They both give public input $(N, e, c_P, r_P, c_V, r_V, \sigma, H_{\tilde{p}_P}, H_{\tilde{q}_P}, H_{\bar{r}_P}, H_{\tilde{p}_V}, H_{\tilde{q}_V}, H_{\bar{r}_V}, \{b_j, v_j, H_{t_j}\}_{i\in[s]})$:

$$
\begin{aligned}
\delta &:= \hat{r}_P + \hat{r}_V \cdot ((\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 5) - \sigma) \\
w &:= N - 5 - 4(\tilde{p}_P + \tilde{q}_P + \tilde{p}_V + \tilde{q}_V) \mod e\ , \\
\chi &:= (H_{\tilde{p}_P} =^? \mathsf{AES}_{K_P}(\tilde{p}_P)) \ \wedge\ (H_{\tilde{q}_P} =^? \mathsf{AES}_{K_P}(\tilde{q}_P)) \\
&\quad \wedge (c_P =^? \mathsf{AES}_{\mathsf{AES}_{r_P}(K_P)}(0)) \ \wedge\ (H_{\bar{r}_P} =^? \mathsf{AES}_{K_P}(\bar{r}_P)) \\
&\quad \wedge (H_{\tilde{p}_V} =^? \mathsf{AES}_{K_V}(\tilde{p}_V)) \ \wedge\ (H_{\tilde{q}_V} =^? \mathsf{AES}_{K_V}(\tilde{q}_V)) \\
&\quad \wedge (c_V =^? \mathsf{AES}_{\mathsf{AES}_{r_V}(K_V)}(0)) \ \wedge\ (H_{\bar{r}_V} =^? \mathsf{AES}_{K_V}(\bar{r}_V)) \\
&\quad \wedge (\forall j \in [s] : \mathsf{AES}_{K_P}(v_j + b_j \cdot (\tilde{p}_P + \tilde{q}_P)) =^? H_{t_j}) \\
&\quad \wedge (N =^? (4(\tilde{p}_V + \tilde{p}_P) + 3) \cdot (4(\tilde{q}_V + \tilde{q}_P) + 3)) \\
&\quad \wedge w \neq 0 \\
\text{if } V = P_1 : d_V &:= \left\lfloor \frac{-(w^{-1} \mod e) \cdot (N - 5 - 4\tilde{p}_1 - 4\tilde{q}_1) + 1}{e} \right\rfloor \\
\text{else} : d_V &:= \left\lfloor \frac{-(w^{-1} \mod e) \cdot (-4\tilde{p}_2 - 4\tilde{q}_2)}{e} \right\rfloor \\
\text{if } \chi = 1 : \bar{d}_V &:= d_V - \rho_P \\
\text{else} : \bar{d}_V &= \perp, \delta = \perp \\
&\text{Output } (\chi, \delta, \bar{d}_V) \text{ to } V \text{ and } (\perp) \text{ to } P
\end{aligned}
$$

13. The parties then use $\mathcal{F}_{\mathsf{2PC}}$ to compute the output of f and abort if $\chi = 0$.[a]
14. P sends $\hat{\delta} = \hat{r}_P \mod N$ to V.
15. V checks that $\hat{\delta} \equiv \delta \mod N$

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.

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 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]. We note that if desired, it is possible to force the most significant bit, of the prime shares, to be 1. Thus guaranteeing that the most significant or second to most significant bit of the 2ℓ bit modulus N is 1 as well. Remember that we suffice with semi-honest execution for most of our protocol and that we force correctness through the zero-knowledge argument in *proof of honesty*. Thus forcing the most significant bit of the shares to be 1 comes down to simply also checking this in the function f computed by $\mathcal{F}_{\mathsf{2PC}}$.

Regarding the allowed leakage we show in Appendix C.2 that (for standard parameters) at most $\log_2(e/(e-1)) + 2\log_2(B_1/2)$ bits are leaked to a malicious 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.[8] 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

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

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. [9]

Formally we prove the following theorem in Appendix C.2.

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

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 Fig. 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

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

[9] These values were calculated using the calculation done in the proof of Lemma C.1.

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.

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]. In effect of this optimization the ideal functionality must be expanded slightly, since now any malicious party can pick an arbitrary predicate to be executed on both its and the honest party's prime shares. This is reflected by allowing the adversary to

give this as input in the *select* command. In this way the simulator can emulate a 1-bit leaked two-party computation by passing the predicate of the adversary onto the ideal functionality and thus achieve an indistinguishable simulation.

Constant rounds. We note that the way our protocols $\Pi_{\mathsf{RSA\text{-}semi}}$ and Π_{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 the main competition lies in the protocols of Boneh and Franklin [BF01], Gilboa [Gil99], and Algesheimer *et al.* [ACS02]. With regards to Boneh and Franklin, the asymptotics are comparable and both schemes support trial division of both the individual candidates and the modulus. However, Boneh and Franklin require an honest majority and thus cannot be realized in the essential

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.

two-party setting. Gilboa supports a dishonest majority. Unfortunately it does not have the option of trial division on the prime candidates and thus requires testing more candidates. However our semi-honest protocol is very similar to one of the protocols introduced by Gilboa in [Gil99], *but* with the addition of efficient OT-based trial division of the prime candidates. Our greatest competition for the semi-honest protocol is with 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.

The only protocols relevant for malicious security are those of Frankel *et al.* [FMY98], Damgård and Mikkelsen [DM10], Poupard and Stern [PS98], and Hazay *et al.* [HMRT12]. Of these protocols only the latter two are secure against a dishonest majority. Comparing to Damgård and Mikkelsen we see that even though they have the most efficient primality test, they unfortunately require a very large computation for *every* potential candidate, even before trial division. This, along with the fact that it requires an honest majority, limits its desirability.

Next we see that on the surface Poupard and Stern looks very competitive to our scheme. However, there are two issues with that protocol that are not reflected in the table: First, 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. Second, the constants in constructing a modulus are significantly larger than ours. The reason is that they must compute several 1-out-of-m OT for different primes m to multiply two prime candidates. Since they need this to be a specific, rather than random, OT, it means that something must be communicated for each possible option.

With this discussion in mind we see that the only competition in the setting of a malicious dishonest majority is the protocol 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 every step compared to our maliciously secure protocol.

Finally, we also consider the protocol of Gavin [Gav12]. This paper does not give a concrete protocol but instead assumes access to different functionalities with certain requirements along with zero-knowledge arguments. Because of the lack of a concrete realization we have not been able to include it in the table. Using these abstract functionalities and arguments the author shows how to compile a "strong" semi-honest scheme into a scheme secure in the malicious model against a dishonest majority. His overall idea is the same as ours; construct a, possibly incorrect modulus and prove that is is correct in the end of the protocol. However, unlike our approach, where we ensure that the other party has used the same candidates in the proof as used to compute the modulus, Gavin instead requires opening and verifying the computation of each failed candidate. This leads to about a factor 2 overhead in the entire protocol. However,

the zero-knowledge argument the author presents uses the Boneh and Franklin biprimality test and thus has s iterations of a constant amount of expensive operations. However, each iterations also requires a zero-knowledge argument. No concrete implementation is given but it seems that a zero-knowledge argument presented by Hazay *et al.* can be used. However, this requires $O(s)$ expensive operations. Thus giving a total complexity of $O(s^2)$ expensive operations, like in our malicious protocol.

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 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 yet implement 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 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.

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 6.8 million. Since we use dual execution, and both parties need to execute the circuit, we get a total of 13.6 million AND gates that must be constructed and evaluated per party. We also measured a throughput of constructing about 3.2 million AND gates in Yao's protocol on the machines that we were using. Since construction takes longer than evaluation, we therefore estimate that implementing these computations using garbled circuits will contribute at most 8.4 wall-clock 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.

Acknowledgment

We would like to thank Muthuramakrishnan Venkitasubramaniam and Samuel Ranellucci for pointing out an issue in step 2 of the biprimality test present in the proceedings version of this paper [FLOP18].

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 Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 417–432. Springer, 2002.

ALSZ13. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *CCS*, pages 535–548. ACM, 2013.

ALSZ15. Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT*, volume 9056 of *Lecture Notes in Computer Science*, pages 673–701. Springer, 2015.

Bea96. Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In Gary L. Miller, editor, *STOC*, pages 479–488. ACM, 1996.

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 Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

DM10. Ivan Damgård and Gert Læssøe Mikkelsen. Efficient, robust and constant-round distributed RSA key generation. In Daniele Micciancio, editor, *TCC*, volume 5978 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2010.

FLOP18. Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO*, volume 10992 of *Lecture Notes in Computer Science*, pages 331–361. Springer, 2018.

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

HD03. Julian Havil and Freeman Dyson. *Gamma: Exploring Euler's Constant*. Princeton University Press, 2003.

HMRT12. Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA key generation and threshold paillier in the two-party setting. In Orr Dunkelman, editor, *CT-RSA*, volume 7178 of *Lecture Notes in Computer Science*, pages 313–331. Springer, 2012.

IKNP03. Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO*, volume 2729 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2003.

IKOS09. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge proofs from secure multiparty computation. *SIAM J. Comput.*, 39(3):1121–1152, 2009.

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 *ACM SIGSAC*, pages 955–966, 2013.

KK13. Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *CRYPTO*, volume 8043 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2013.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew Robshaw, editors, *CRYPTO*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 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.

NNOB12. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

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 Helena Handschuh, editor, *CT-RSA*, volume 10159 of *Lecture Notes in Computer Science*, pages 381–396. Springer, 2017.

PS98. Guillaume Poupard and Jacques Stern. Generation of shared RSA keys by two parties. In Kazuo Ohta and Dingyi Pei, editors, *ASIACRYPT*, volume 1514 of *Lecture Notes in Computer Science*, pages 11–24. Springer, 1998.

PVW08. Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In David A. Wagner, editor, *CRYPTO*, volume 5157 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 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), 4 encryptions of ℓ bits (for the prime shares), s encryptions of values of length $\ell + s$ bits (for step 1 of the maliciously secure biprimality tests) and 2 encryptions of 2ℓ bits (for step 2 of the maliciously secure biprimality test). For $\ell = 1024$ bits (RSA with a 2048 bit modulus) this translates to 412 AES blocks. With an AES circuit of size 6000 AND gates, this translates to $2.5M$ gates.
- Multiplications: The circuit computes one multiplication (for computing N) where each of the inputs is ℓ bits long. For $\ell = 1024$ using Karatsuba multiplication this takes 0.542M gates [Sch12]. We have another multiplication where the inputs are at most 2ℓ bits (the computation of δ) which gives at most 1.692M gates using Karatsuba, unoptimized for inputs of uneven length. Finally the computation of δ also requires another multiplications of a 3ℓ bit number with a s bit number; thus at most 0.122M gates using the school book method.
- Division: The circuit computes a division of an 2ℓ bit value by $e = 2^{16} + 1$. For $\ell = 1024$ the size of this component is about 900K AND gates.
- Computing the inverse of an 2ℓ 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 6.8M 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 constructed is less than 27M.

B The Running Times of the Different Components

Let us analyze the number of times that each phase of the protocol needs to be run, and the expected overhead. These values are obviously dependent on the parameters B_1, B_2 which define the number of tests in the first and second trial division phases.

The fraction of prime numbers. First, we recall that the number of primes less than B, denoted $\pi(B)$ is bounded by $0.922 < \pi(B)/(B/\ln B) < 1.105$ [HD03].We are actually interested in the *fraction* of primes, namely the probability that a

random integer is a prime. Denote this probability as $Pr_{prime} = \pi(B)/B$. Thus, we have $0.922/\ln B < Pr_{prime} = \pi(B)/B < 1.105/\ln B$. Setting $B = 2^{1024}$ we have that $\ln B \approx 709$ and so

$$\frac{1}{769} < Pr_{prime} < \frac{1}{642} \ .$$

This means that in a naive run of the protocol, the expected number of runs until we obtain a single prime is between 642 and 769. Since the protocol verify that all numbers that are checked are odd, the expected number of trials is halved. The effective value of Pr_{prime} is therefore between 321 and 385. Note, however, that the vast majority of these will fail in the first trial division.

Passing the first trial division phase. The first trial division phase verifies that a candidate number is not divisible by any prime number up to a threshold value B_1. Let p_i be the sequence of prime numbers starting with $p_1 = 3$. The Chinese Remainder Theorem implies that a candidate odd number passes the first trial division phase with probability

$$Pr_1 = \prod_{p_i \leq B_1} \left(1 - \frac{1}{p_i}\right) \ .$$

The cost of the first trial division phase. The "fail-fast" optimization suggests terminating the first trial division phase as soon as any trial division fails. Therefore, with probability $1/3$ the phase terminates after the first trial division, with probability $(2/3) \cdot (1/5)$ it fails after the second trial division, etc. The expected number of trial divisions in a single run of this phase is therefore

$$T_1 = \sum_{p_i \leq B_1} \prod_{j=1}^{i-1} \left(1 - \frac{1}{p_j}\right) \cdot p_i \cdot i \ .$$

(This is a simplification of the run time since we know that in the last iteration of this phase all trial divisions were computed.) The first trial division phase terminates after finding two numbers which pass the test. The total run time of the test until this success is about

$$T_{\text{phase 1}} = 2 \cdot T_1 \cdot (1/Pr_1) \ .$$

Note that $T_{\text{phase 1}}$ is a function of B_1.

The cost of the second trial division phase. Once two numbers pass the first trial division phase the protocol constructs the modulus candidate N and computes trial divisions by all primes in the range $]B_1, B_2]$. The exact expression for expected cost of a single run of this phase, T_2, can be computed in exactly the same way as T_1. The value of T_2 is a function of B_1 and B_2.

Let Pr_2 be the probability that a prime candidate passes the second trial division phase given that it passed the first trial division phase, namely $Pr_2 =$

$\prod_{B_1 \leq p_i \leq B_2}(1 - \frac{1}{p_i})$. The probability that a modulus candidate that entered the second trial division phase passes it, is therefore $(Pr_2)^2$. In case of a failure the protocol needs to choose new candidate primes and start again from computing the first trial division phase.

The biprimality test. The probability that a number that passed the second trial division phase is indeed the product of two primes, is $Pr_3 = Pr_{prime}/(Pr_1 Pr_2)^2$.

The total runtime. Let us denote by $T_{\text{biprimality}}$ the overhead of running the biprimality test, and by T_{proof} the overhead of computing and verifying the proof along with the zero-knowledge argument (since they are both carried out only once). The total expected runtime is therefore

$$\left((T_{\text{phase 1}} + T_2) \cdot \frac{1}{Pr_2} + T_{\text{biprimality}}\right) \cdot \frac{1}{Pr_3} + T_{\text{proof}}$$

The values of B_1, B_2 should be set to minimize this total run time. Increasing B_1 and B_2 decreases the number of "false primes" which pass the trial divisions but are not prime and therefore decreases the number of times the protocol is repeated. On the other hand, increasing B_1, B_2 increases the run time of the trial division phases and has diminishing returns.

B.1 The Overhead of Malicious Security

We argue that the malicious protocol can be realized with only a constant overhead over the semi-honest protocol (and thus has the same asymptotic complexity). First, we analyze the overhead of all phases up to the proof of honesty:

- Computing the zero-knowledge argument can be done using a single garbled circuit implementing the verification function [JKO13]. To find the overall complexity first see that the computation of an AES block requires a constant amount of AND gates per block and each AND gate requires a constant amount of calls to the hash function used in the garbling scheme. Each block is bounded by $O(\kappa)$, thus we get $O(\kappa)$ complexity since the input-length is 2κ and we assume usage of OT extension.
- Computing the AES based commitments on each prime candidate only adds a constant overhead of light operations.
- Next note that a malicious OT extension adds only a constant overhead over the semi-honest one [KOS15,OOS17].
- The construction of the modulus uses a few more OTs. However, the number of these OTs is only $O(\ell + s)$, and since $\ell > s$ and we already do ℓ OTs in the semi-honest protocol this is only a constant additional overhead.
- Coin-tossing can be implemented in the non-programmable random oracle model using only a constant number of calls to a hash function and a PRG, and thus does not add more than constant overhead.

Finally, the *proof of honesty* requires a re-execution of the biprimality test, thus $O(s)$ exponentiations. Furthermore, since each of these tests gets challenged s times, the parties need to actually do $O(s^2)$ exponentiations in total. However, the amount of times the *biprimality test* is executed both in the semi-honest and malicious protocols will be $O(\ell^2/\ln^2(\ell))$ (based on the discussion in Section B). Notice that $O(\ell) \in o(\ell^2/\ln^2(\ell))$ and $\ell > s$, thus the *biprimality test* phase will be require $\Omega(s^2)$ exponentiations and thus the addition of a single execution of $O(s^2)$ exponentiations *once* will only yield a constant overhead.

However, the *proof of honesty* also requires the execution of a secure computation. Concretely assume that a dual-execution garbled circuit approach is used for this. That is, using the approach outlined in Section 4. Several computations are done in this garbled circuit. To find the overall complexity first see that the computation of AES requires a constant times the block length. Assume that the block length is κ and thus that $O(\ell/\kappa)$ blocks are required to encrypt ℓ bits. Thus an AES encryption requires $O(\ell)$ AND gates. Since $O(s)$ encryptions are computed, this gives $O(\ell \cdot s)$ AND gates. To verify the value H_{t_j} the circuit must compute addition on ℓ bit values, which requires $O(\ell)$ AND gates. However, this is also done s times, thus requiring $O(\ell \cdot s)$ AND gates. Next see that we carry out a few multiplications of numbers of at most 2ℓ bits, which (conservatively) requires $O(\ell^2)$ gates. Finally the computation of the modulo e operation along with integer division of e can be implemented using a constant amount of multiplications over values in $O(\ell)$, thus with $O(\ell^2)$ complexity in the worst case.

Each AND gate in a garbled circuit requires a constant amount of symmetric operations and since we use the dual execution approach, only a constant amount of garbled circuits are constructed. Thus, since $s < \ell$ we get total complexity of at most $O(\ell^2)$ symmetric operations for this execution. But since *construct modulus* requires $O(\ell)$ cheap operations and is executed $O(\ell^2/\ln^2(\ell))$ times we see that asymptotically the secure computation is *less* than what is already done and thus the overhead is only constant.

C Proof of Security

C.1 Correctness

We prove the correctness of the various parts of the protocol.

Candidate generation If the parties behave honestly then this phase constructs an additively shared $\ell - 1$ bit number, p, which does not contain any prime factor less than or equal to B_1.

To see this notice that 2 is not factor of p, since $p = p_1 + p_2 = 4\tilde{p_1} + 3 + 4\tilde{p_2} \mod 2 = 1$. Next see that if $a_1 \neq a_2$, then β is not a factor for $\beta \in \mathcal{B}$. This follows since if $m_{a_1} = m_{a_2}$, then $a_1 = a_2 \Leftrightarrow a_1 - a_2 = 0$. Thus $a_1 - a_2 = p_1 + p_2 \mod \beta = p \mod \beta$. This means that if $a_1 - a_2 = 0$ then β is a factor of p. Also note from the calculation that no p will be discarded if it does not have any factor $\beta \in \mathcal{B}$.

Construct modulus If the parties behave honestly then this phase constructs a product $N = pq$, of two additively shared values p and q. Furthermore, it will verify that e is not be a factor of $\phi(N)$ (under the assumption that p and q are primes).

For $\alpha = p$ we see that after step 4 it is the case that:

$$\begin{aligned} z_1^p &= \sum_{i\in[2\ell+3s]} c_{d_{p,i},i} \cdot h_{p,i} \mod \mathcal{P} = \sum_{i\in[2\ell+3s]} h_{p,i} \cdot (r_{p,i} + d_{p,i} \cdot q_2) \mod \mathcal{P} \\ &= \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot r_{p,i}\right) + \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot d_{p,i} \cdot q_2\right) \mod \mathcal{P} \\ &= \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot r_{p,i}\right) + q_2 \cdot \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot d_{p,i}\right) \mod \mathcal{P} \end{aligned}$$

and thus we have

$$\begin{aligned} z_1^p + z_2^p &= \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot r_{p,i}\right) + q_2 \cdot \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot d_{p,i}\right) \\ &\quad - \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot r_{p,i}\right) \mod \mathcal{P} \\ &= q_2 \cdot \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot d_{p,i}\right) \mod \mathcal{P} \ . \end{aligned}$$

The case for $\alpha = q$ follows trivially.

Next consider the following computation:

$$\begin{aligned} a_1 + a_2 &\equiv p_1 q_1 + z_1^p + z_1^q + p_2 q_2 + z_2^p + g_p \cdot q_2 + z_2^q + g_q \cdot p_2 \mod \mathcal{P} \\ &\equiv p_1 q_1 + q_2 \cdot \left(\sum_{i\in[2\ell+3s]} h_{p,i} \cdot d_{p,i}\right) + q_2 \cdot g_p \\ &\quad + p_2 q_2 + p_2 \cdot \left(\sum_{i\in[2\ell+3s]} h_{q,i} \cdot d_{q,i}\right) + g_q \cdot p_2 \mod \mathcal{P} \\ &\equiv p_1 q_1 + p_1 q_2 + p_2 q_1 + p_2 q_2 \mod \mathcal{P} \\ &\equiv (p_1 + p_2)(q_1 + q_2) \equiv N \mod \mathcal{P} \end{aligned}$$

Since $p_1, p_2, q_1, q_2 < 2^{\ell-1}$ we have have that $p_1 + p_2 < 2^\ell$ and $q_1 + q_2 < 2^\ell$ and thus $(p_1 + p_2)(q_1 + q_2) < 2^{2\ell}$ and thus $N < 2^{2\ell} < \mathcal{P}$, meaning $(N \mod \mathcal{P}) \mod 2^{2\ell} = N$.

Next see that if p and q are primes, then

$$\begin{aligned}\phi(N) &= (p-1)(q-1) = (p_1+p_2-1)(q_1+q_2-1)\\ &= p_1q_1 + p_1q_2 + p_2q_1 + p_2q_2 - p_1 - q_1 - p_2 - q_2 + 1\\ &= N + 1 - (p_1+q_1) - (q_1+q_2)\end{aligned}$$

Next see that if $N+1-(p_1+q_1)-(q_1+q_2) \mod e = 0$, then e is a factor of N. Thus, if $N+1-(p_1+q_1) \equiv p_2+q_2 \mod e$ then $e|\phi(N)$. In particular, since $w_1 = N+1-(p_1+q_1) \mod e$ and $w_2 = p_2+q_2 \mod e$, if $w_1 = w_2$ then $e|\phi(N)$. We see that checking if $r_{w_1} = r_{w_2}$ is equivalent to this since chance of collision between two values r_i and r_j for $i \neq j$ is negligible in the security parameter since the values r_i, r_j are randomly sampled.

Trial division If the parties behave honestly then this phase ensures that N does not contain any prime factor less than or equal to B_2. In particular this also ensures that p and q do not contain any prime factors less than or equal to B_2.

Biprimality test If the parties behave honestly then this phase ensures that N is a product of two primes p and q, except with probability low probability. Furthermore, the phase ensures that if N *is* a product of two primes, then it will never be discarded. We will not go into correctness here, since this phase implements exactly the biprimality check of Boneh and Franklin [BF01]. Thus, we refer the reader to that paper for a proof of correctness. However, we note that what they call step 4) (we call step 2) is the check that $\gcd(p+q-1, N) = 1$. This is not exactly the same as what is checked in their work, but in fact our check, $\gcd(p+q-1, N) = 1$, is a generalization. The check (of Boneh and Franklin [BF01]) is done to remove false positives passing step 1, which happens when $p = r_p^{d_p}$ and $q = r_q^{d_q}$ for $d_p > 1$ and $q \equiv 1 \mod r_p^{d_p-1}$. We see that if this is the case then $\gcd(r_p^{d_p} + r_q^{d_q} - 1, r_p^{d_p} r_q^{d_q}) = r_p^{d_p-1}$ and thus it is sufficient to verify that $\gcd(p+q-1, N) = 1$ as this will always be true when $N = pq$ and p and q are primes as $p+q-1 \neq p \neq q$. This check is implemented by checking $\gcd(r(p+q-1) \mod N, N) = 1$ where r is a random number of $2\ell+1$ bits. We first show that if $\gcd(r(p+q-1), N) = 1$, then it also holds that $\gcd(r(p+q-1) \mod N, N) = 1$. To see this first consider writing $r(p+q-1)$ as $a \cdot N + b$ where $0 \leq b < N$, then it is sufficient to show that if $\gcd(b, N) = c$ then $c|1$ and hence $c = 1$. We notice that $c|N$ and $c|b$ and so $c|(a \cdot N + b)$. This implies that $c|\gcd(r(p+q-1), N)$ and hence $c = 1$ if $\gcd(r(p+q-1), N) = 1$. Next, it is easy to see that if p and q are primes then $\gcd(r(p+q-1), N) = 1$ except if $r = p$ or $r = q$ which only happens with negligible probability since $|p|, |q| = \ell > s$. Finally, we show that this is actually what gets computed by the protocol. First see that $\bar{r}_1(p_2+q_2-1) < 2^{3\ell}$ since $\bar{r}_1 \in \mathbb{Z}_N$ and thus less than $2^{2\ell}$ and $p_2, q_2 < 2^{\ell-1}$. Then notice that α_1, α_2 are additive secret shares of $\bar{r}_1(p_2+q_2-1)$ over $\mathbb{Z}_{2^{3\ell+s+2}}$ by the "Construct Modulus" computation. Thus considering a single of these values, say α_1, will be a uniformly random value

from $\mathbb{Z}_{2^{3\ell+s+2}}$. This means that except with probability 2^{-s-2} it will hold that $\alpha_1 > 2^{3\ell}$. However if that is the case we know that it holds that $\alpha_2 > 2^{3\ell}$ as well, since $\alpha_1 + \alpha_2 \mod 2^{3\ell+s+2} = \bar{r}_1(p_2 + q_2 - 1) < 2^{3\ell}$, meaning that the only way the sum of α_1 and α_2 can be a share of a number less than $2^{3\ell}$ is if a modular reduction occurs. Now since $\alpha_1, \alpha_2 \in \mathbb{Z}_{2^{3\ell+s+2}}$ we know that only a single modular reduction happens. So we can conclude that $\alpha_1 + \alpha_2 - 2^{3\ell+s+2} = \bar{r}_1(p_2 + q_2 - 1)$ except with probability at most 2^{-s-2}. The same argument holds for β_1, β_2 and thus we get that $\alpha_1 + \beta_1 \mod 2^{3\ell+s+2}$ is a sharing of the value $\bar{r}_1(p_2 + q_2 - 1) + \bar{r}_2(p_1 + q_1) < 2^{3\ell+1}$. By using the same argument again we get that $\left(\alpha_1 + \beta_1 \mod 2^{3\ell+s+2}\right) + \left(\alpha_2 + \beta_2 \mod 2^{3\ell+s+2}\right) > 2^{3\ell+s+2}$ except with probability $2^{-s-1} + 2^{-s-1} = 2^{-s}$ by the Union Bound. Thus we get that $s_1 + s_2 \mod N = (\bar{r}_1 + \bar{r}_2)(p_1 + p_2 + q_1 + q_2 - 1) = r(p + q_1)$ for some value r except with probability 2^{-s}.

Proof of honesty If the parties behave honestly then this phase does exactly the same as *Biprimality Test* (along with executing step 2) of the biprimality test in Fig. 3.4, along with computing the private key shares using an ideal secure computation.

To see correctness, we first notice that the first four steps are the same as in *Biprimality Test*. To see this, first notice that $N - 5$ is divisible by 4:

$$\begin{aligned} N &= (p_1 + p_2)(q_1 + q_2) = (4\tilde{p}_1 + 3 + 4\tilde{p}_2)(4\tilde{q}_1 + 3 + 4\tilde{q}_2) \\ &= 16(\tilde{p}_1\tilde{q}_1 + \tilde{p}_1\tilde{q}_2 + \tilde{p}_2\tilde{q}_1 + \tilde{p}_2\tilde{q}_2) + 12(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 9 \end{aligned}$$

Thus $\frac{N-5}{4} = 4(\tilde{p}_1\tilde{q}_1 + \tilde{p}_1\tilde{q}_2 + \tilde{p}_2\tilde{q}_1 + \tilde{p}_2\tilde{q}_2) + 3(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 1$. With this in mind we see that the check performed in step 3 of *Proof of Honesty* is the same as in *Biprimality Test* since the exponent of γ_i will take the value:

$$\begin{aligned} \frac{N + 1 - p_1 - q_1 - p_2 - q_2}{4} &= \frac{N + 1 - 4\tilde{p}_1 - 3 - 4\tilde{q}_1 - 3 - 4\tilde{p}_2 - 4\tilde{q}_2}{4} \\ &= \frac{N - 5 - 4\tilde{p}_1 - 4\tilde{q}_1 - 4\tilde{p}_2 - 4\tilde{q}_2}{4} \\ &= \frac{N - 5}{4} - \tilde{p}_1 - \tilde{q}_1 - \tilde{p}_2 - \tilde{q}_2 \end{aligned}$$

Next we see that the checks performed in step 9 will always pass:

$$\begin{aligned} \gamma_i^{v_j} &\equiv \bar{\gamma}_{i,j} \cdot \gamma_{j,P}^{b_j} \cdot \gamma_j^{-b_j \cdot \frac{N-5}{4}} \mod N \\ \gamma_i^{b_j \cdot (-\tilde{p}_P - \tilde{q}_P) + t_j} &\equiv \gamma_i^{t_j} \cdot \left(\gamma_i^{\frac{N-5}{4} - \tilde{p}_P - \tilde{q}_P}\right)^{b_j} \cdot \gamma_j^{-b_j \cdot \frac{N-5}{4}} \mod N \\ \gamma_i^{b_j \cdot (-\tilde{p}_P - \tilde{q}_P) + t_j} &\equiv \gamma_i^{t_j + b_j\left(\frac{N-5}{4} - \tilde{p}_P - \tilde{q}_P\right) - b_j \cdot \frac{N-5}{4}} \mod N \\ \gamma_i^{b_j \cdot (-\tilde{p}_P - \tilde{q}_P) + t_j} &\equiv \gamma_i^{t_j + b_j(-\tilde{p}_P - \tilde{q}_P)} \mod N \end{aligned}$$

Further, see that step 2) of Fig. 3.4 is verified by the computation

$$\hat{r}_P \mod N = \hat{r}_P + \hat{r}_V \cdot ((\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 5) - \sigma) \mod N \ .$$

Where $\sigma = (\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 5) \mod N$. Thus $((\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 5) - \sigma) \mod N = 0$ and hence $\hat{r}_P \mod N = \hat{r}_P + \hat{r}_V \cdot 0 \mod N$.

Finally we see that if the parties have been honest and e is not a factor of $\phi(N)$, then $\mathcal{F}_{\mathsf{2PC}}$ will always output $\chi = 1$. To see this first notice that $v_j + b_j \cdot (\tilde{p}_P + \tilde{q}_P) = b_j \cdot (-\tilde{p}_P - \tilde{q}_P) + t_j + b_j \cdot (\tilde{p}_P + \tilde{q}_P) = t_j$. Then see that the rest of the function computes d_V like in the semi-honest protocol by Boneh and Franklin, just with the value d_V subtractively randomized with a value ρ_P. Specifically we notice the following:

$$d_1' + d_2 = \bar{d}_1 + \rho_1 + \bar{d}_2 + \rho_2 = (d_1 - \rho_2) + \rho_1 + (d_2 - \rho_1) + \rho_2 = d_1 + d_2$$

Seeing that the shares are computed to add up to the same value as in the Boneh and Franklin protocol.

Furthermore, it is trivial to see that if a party gives correct input then the AES verifications done in $\mathcal{F}_{\mathsf{2PC}}$ will always be accepted.

Generate shared key In general, the correctness of the shares d_1 and d_2 follows from Boneh and Franklin [BF01], as they are constructed through the same calculation, $d_1 = \lfloor \frac{-(w^{-1} \mod e) \cdot (N+1-p_1-q_1)+1}{e} \rfloor$ and $d_2 = \lfloor \frac{-(w^{-1} \mod e) \cdot (-p_2-q_2)+1}{e} \rfloor$ respectively. However, this *can* results in an off-by-one error in the sum of the shares because of the rounding. In the case of Boneh and Franklin, this is discovered through a trial decryption. However, we argue that this is not necessary and that the parties can locally discover if there is an off-by-one error. To see this first notice that there will always be a rounding error if rounding occurs when either computing d_1 or d_2. This is so since, by correctness. the sum of the numerators in the computation of d_1 and d_2 must necessarily be divisible by e (see page 715 in [BF01]). This implies that if $e | -(w^{-1} \mod e) \cdot (N+1-p_1-q_1)$ and $e | -(w^{-1} \mod e) \cdot (-p_2-q_2)$ then there is no rounding error. However, if $e \nmid -(w^{-1} \mod e) \cdot (N+1-p_1-q_1)$ (or $e \nmid -(w^{-1} \mod e) \cdot (-p_2-q_2)$) then it must necessarily hold that $e \nmid -(w^{-1} \mod e) \cdot (N+1-p_1-q_1)$ *and* $e \nmid -(w^{-1} \mod e) \cdot (-p_2-q_2)$ since the remainders of $-(w^{-1} \mod e) \cdot (N+1-p_1-q_1) \mod e$ and $-(w^{-1} \mod e) \cdot (-p_2-q_2) \mod e$ must sum together to e and there will thus be an off-by-one. This means that if $e | -(w^{-1} \mod e) \cdot (-p_2-q_2)$ there will *never* be an off-by-one error, but if $e \nmid -(w^{-1} \mod e) \cdot (-p_2-q_2)$ there will *always* be an off-by-one error. Now, since we assume that e is prime and since $w^{-1} \mod e$ is obviously less than e it must be the case that $e | -(w^{-1} \mod e) \cdot (-p_2-q_2)$ iff $e | -p_2-q_2$. Substituting p_1 with $4\tilde{p}_1$, q_1 with $4\tilde{q}_1$, p_2 with $4p_2+3$ and q_2 with $4q_2+3$ correctness of the key generation follows.

C.2 Security

Proof. (of Theorem 3.11).

Simulator To make the simulation and indistinguishability argument simpler we consider an expanded version of $\mathcal{F}_{\mathsf{RSA}}$. Specifically we assume that in the case of corruption it has one more command:

Full-leak: On input $j \in J$ from $P_{\mathcal{I}}$ return $p_{3-\mathcal{I},j}$ to $P_{\mathcal{I}}$ and set $J = J \backslash \{j\}$.

We now show that this expanded $\mathcal{F}_{\mathsf{RSA}}$ functionality is statistically indistinguishable from the regular $\mathcal{F}_{\mathsf{RSA}}$ functionality. To see this, consider a simulator sitting on top of the regular $\mathcal{F}_{\mathsf{RSA}}$ functionality, using this to emulate the *full-leak* command (passing all other commands directly through to the regular $\mathcal{F}_{\mathsf{RSA}}$). It does so by picking $p_{3-\mathcal{I},j}$ uniformly at random but consistent with the constraints of the functionality and the info that might already be leaked.

To see that this simulator is sufficient, first notice that any leaked share from *full-leak*, cannot be used in the construction of a modulus that is given as output of the functionality. Next see that the prime shares in the regular $\mathcal{F}_{\mathsf{RSA}}$ are picked uniformly at random (under the constraint that their sum is congruent to 3 mod 4). This means no environment can distinguish whether the random share it receives from $\mathcal{F}_{\mathsf{RSA}}$ is the true, internal random share, or something sampled from the same distribution. Furthermore, this holds even if the adversary has already used the *leak* command on prime j. To see this notice that if P_1 is corrupt it is easy for the simulator to pick a random share consistent with what is leaked when $a_{j,\beta} = -p_{2,j} \mod \beta$, by plugging the leaked remainder into the Chinese Remainder Theorem and simply choosing the rest uniformly at random.

We now continue with the simulation proof, assuming this expanded version of the $\mathcal{F}_{\mathsf{RSA}}$ functionality.

We denote the corrupt $P_{3-\mathcal{I}}$ by $\hat{P}_{3-\mathcal{I}}$ and construct a simulator $\mathsf{Sim}_{\mathcal{I}}$ emulating $P_{\mathcal{I}}$ for $\mathcal{I} \in \{1, 2\}$. The simulator emulates each of the steps in $\mathcal{F}_{\mathsf{RSA}}$ as follows, inputting `abort` to $\mathcal{F}_{\mathsf{RSA}}$ if $\hat{P}_{3-\mathcal{I}}$ inputs this to any of the emulated subfunctionalities at any point.

Corrupt P_1 Consider the simulator Sim_2:

Setup.

1. Emulate $\mathcal{F}_{\mathsf{CT}}$ twice by sampling $r_1, r_2 \in \{0,1\}^{\kappa}$ uniformly at random.
2. $\mathcal{I} = 1$: Receive the message from $\hat{P}_1$, c_1.
2. $\mathcal{I} = 2$: Pick $K_2 \in \{0,1\}^{\kappa}$ uniformly at random on behalf of of P_2, compute $\mathsf{AES}_{\mathsf{AES}_{r_2}(K_2)}(0) = c_2$ and send c_2 to $\hat{P}_1$.
3. $\mathcal{I} = 1$: Extract the input K_1 from $\hat{P}_1$ and emulate $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ correctly, by verifying that $\mathsf{AES}_{\mathsf{AES}_{r_1}(K_1)}(0) = c_{\mathcal{I}}$, and output $\perp$ if that is not the case.
3. $\mathcal{I} = 2$: Emulate $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ by returning $\top$ to $\hat{P}_1$.
4. Output `abort` if $\perp$ was returned in step 3.

Candidate Generation.

1. Receive the value $H_{\tilde{p}_1}$ from $\hat{P}_1$. Use the key K_1 extracted from $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ to learn the value encrypted $\tilde{p}_1$ and define $p_1 = 4 \cdot \tilde{p}_1 + 3$.

2. Sim_2 picks a uniformly random value $\tilde{p}_2 \in \mathbb{Z}_{2^{\ell-3}}$. Define $p_1 = 4\tilde{p}_1 + 3$ and $p_2 = 4\tilde{p}_2$. Sim_2 computes the commitment $H_{\tilde{p}_2} = \mathsf{AES}_{K_2}(\tilde{p}_2) \in \{0,1\}^\kappa$ and sends $H_{\tilde{p}_2}$ to $\hat{P}_1$.
3. If $p = p_1 + p_2$ is a prime and $\gcd(e, p-1) = 1$ then it marks this shared candidate prime, queries the *Sample* method of $\mathcal{F}_{\mathsf{RSA}}$ with p_1 and proceeds with the simulation by following the steps below of Div-OT for each $\beta \in \mathcal{B}$. Otherwise if p is not prime, or $\gcd(e, p-1) \neq 1$, then Sim_2 proceeds with the simulation of the rest of the protocol by executing like the true P_2, using the value p_2.

1-2. Emulate random $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$ by extracting the input choice a'_1 of $\hat{P}_1$ and returning a random value $m_{a'_1} \in \{0,1\}^\kappa$.

3. Sim_2 then uses *Leak* on $\mathcal{F}_{\mathsf{RSA}}$ to learn if $(-p_2 \mod \beta \stackrel{?}{\neq} a'_1)$. If so, then it picks a random value $m \in \{0,1\}^\kappa$ and returns this to $\hat{P}_1$ (as the value received from P_2). Otherwise it returns $m_{a'_1}$.

4. If $\hat{P}_1$ sends back $\perp$ then the simulator breaks the loop and discards the prime candidate p. If it instead sends back $\top$, it continues.

Construct Modulus. Consider candidate values p and q constructed in the previous phase, if none of them is marked as prime, then Sim_2 simulates the rest of the protocol like P_2 using the shares p_2 and q_2 it constructed in the simulation of *Candidate Generation*, and marks the resulting N bad. If one of the values (assume w.l.o.g. p) is marked as prime and the other is not, then Sim_2 uses *Full-Leak* on $\mathcal{F}_{\mathsf{RSA}}$ to learn p_2 and proceeds by simulating P_2 using the p_2 it received from $\mathcal{F}_{\mathsf{RSA}}$ and the value q_2 it constructed in *Candidate Generation*, and marks the resulting N as bad. Finally, if both p and q are marked as prime, it proceeds as below:

2. Sim_2 executes the following steps for each $\alpha \in \{p, q\}$ and $i \in [2\ell + 3s]$:
   (a) Sim_2 chooses a uniformly random value $r_{\alpha,i} \in \mathbb{Z}_{\mathcal{P}}$ and sets $c_{0,\alpha,i} = c_{1,\alpha,i} = r_{\alpha,i}$.

   (b-c) Sim_2 emulates $\mathcal{F}_{\mathsf{OT}}^{2\ell+3s,2}$, extracts the input $d_{\alpha,i} \in \{0,1\}$ of $\hat{P}_1$ and returns the message $c_{d_{\alpha,i},\alpha,i}$ to $\hat{P}_1$.
3. Receive the values $h_{\alpha,i}$ for $\alpha \in \{p, q\}$ and $i \in [2\ell + 3s]$ from $\hat{P}_1$.
4. Sim_2 computes $z_1^\alpha = \sum_{i \in [2\ell+3s]} c_{d_{\alpha,i},\alpha,i} \cdot h_{\alpha,i} \mod \mathcal{P}$ for $\alpha \in \{p, q\}$. It then uses the *Construct* method on $\mathcal{F}_{\mathsf{RSA}}$ to construct a modulus based on $\hat{P}_1$'s shares p_1 and q_1, and thus receives (N, d_1). It then uses N to compute:

$$a_2 = N - (p_1 q_1 + z_1^p + z_1^q) \mod \mathcal{P} \ .$$

5. Sim_2 sends a_2 to $\hat{P}_1$.
6. Sim_2 receives a'_1 from $\hat{P}_1$.
7. Compute $\bar{N} = (a'_1 + a_2 \mod \mathcal{P}) \mod 2^{2\ell} = \bar{N}$. If $\bar{N} \neq N$ then Sim_2 sets $N := \bar{N}$, thus using the incorrect value $\bar{N}$ in the rest of the simulation.

8-10. Sim_2 extracts $\hat{P}_1$'s input to the OT, w'_1. It then returns a uniformly random value $r'_{w_1} \in \{0,1\}^\kappa$ to $\hat{P}_1$.

11. It inputs the value w_1' to the ideal functionality (still during the *Construct* command) and learns if $w_1' = p_2 + q_2 \mod e$. If this is the case then Sim_2 returns r'_{w_1} to $\hat{P}_1$, otherwise it returns a uniformly random value $r \in \{0,1\}^\kappa$.
12. If $\hat{P}_2$ asks to discard the candidate, then do so. Otherwise continue.

Trial Division: Sim_2 locally executes the trial division like in the real protocol and sends $\perp$ and discards the candidate N if it finds a factor. If it doesn't it sends $\top$ to $\hat{P}_1$.

Biprimality Test:

1. Sim_2 repeats the steps below s times:
   a-c. Receive γ', γ_1' from $\hat{P}_1$.
   d. Sim_2 verifies that $\gamma_1' = \pm\gamma^{\frac{N+1-p_1-q_1}{4}} \mod N$, if this is not the case then it sends $\perp$ to $\hat{P}_1$, breaks the loop and discards the candidate N.
2. Sim_2 does as follows:
   (a) Store the value $H'_{\bar{r}_1}$ received from $\bar{P}_1$ and let $\bar{r}_1 = \mathsf{AES}_{K_1}(H'_{\bar{r}_1})$. Also pick a random value $\bar{r}_2 \in \mathbb{Z}_N$ and send $H_{\bar{r}_2} = \mathsf{AES}_{K_2}(\bar{r}_2)$ to $\hat{P}_1$.
   (b) Sim_2 simulates the multiplication protocol by extracting the messages $\hat{P}_1$ input to the $\mathcal{F}_{\mathsf{OT}}^{3\ell+s+2,2}$ functionality when the simulator is acting as the receiver. When it is acting as the sender it picks a random value $p_2' + q_2' - 1 \in \mathbb{Z}_{2^\ell}$ obeying the leaked bits on the p_2 and q_2 that is learned from $\mathcal{F}_{\mathsf{RSA}}$ though under the constraint that $\gcd(N, p_1+p_2'+q_1+q_2'-1) = 1$). Using this it emulates the OT messages an honest party would send. It then computes the values α_2 and β_2 like an honest P_2 would, using these choices.
   (c) Receive s_1' from $\hat{P}_1$.
   (d) Sim_2 computes $s_2 = \bar{r}_2 \cdot (p_2' + q_2' - 1) + \left(\alpha_2 + \beta_2 \mod 2^{3\ell+s+2}\right) \mod N$ and sends s_2 to $\hat{P}_1$.
   (e) If $\gcd(s_1' + s_2, N) = 1$.

Proof of Honesty. For each $i \in [s]$ Sim_2 picks random values $x_i \in \mathbb{Z}_N^\times$ and $c_i \in \{0,1\}$ and computes $\gamma_i = (-1)^{c_i} \cdot x_i^2 \mod N$. It then emulates $\mathcal{F}_{\mathsf{CT}}$ by picking random values $\gamma \in \mathbb{Z}_N^\times$ and sending each of these to $\hat{P}_1$. Before sending a value γ it checks if $\left(\frac{\gamma}{N}\right) = 1$. If this is the case, it sends a γ_i instead of γ until all s values γ_i have been sent to $\hat{P}_1$.

The simulator then completes the steps below, simulating both P and V.

Simulating P ($\hat{P}_1$ has the role of V):

1. For each $i \in [s]$, Sim_2 computes the values $\gamma_{i,P} = (-1)^{c_i} \cdot \gamma_i^{\tilde{p}_1+\tilde{q}_1} \mod N$.
2. It sends $\gamma_{1,P}, \ldots, \gamma_{s,P}$ to $\hat{P}_1$.

3-4. If $\hat{P}_1$ sends $\perp$ then Sim_2 inputs $\perp$ to $\mathcal{F}_{\mathsf{RSA}}$.

5. Internally emulate $\mathcal{F}_{\mathsf{CT}}$ to pick random bits $b_1, \ldots, b_s$. For each $j \in [s]$ Sim_2 picks two random values $t_j, t_j' \in \{0,1\}^{\ell-2+s}$ and sends $H_{t_j} = \mathsf{AES}_{K_2}(t_j)$ to $\hat{P}_1$

6. For each $i, j \in [s]$, Sim_2 sends the values $\bar{\gamma}_{i,j} = \gamma_i^{t_j} \cdot \left((-1)^{c_i} \cdot \gamma_i^{\frac{N-5}{4} - \tilde{p}_1 - \tilde{q}_1}\right)^{b_j} \mod N$ to $\hat{P}_1$.
7. Output the random bits $b_1, \ldots, b_s$ to $\hat{P}_1$ emulated in step 5.
8. For each $j \in [s]$ send $v_j = t_j$ to V if $b_j = 0$ and $v_j = t'_j$ if $b_j = 1$.
9. If $\hat{P}_1$ sends back $\perp$ then Sim_2 inputs $\perp$ to $\mathcal{F}_{\mathsf{RSA}}$ and aborts.

10-11. Do nothing.

12-13. Extract the input $(\tilde{p}'_V, \tilde{q}'_V, K'_V, \bar{r}'_V, \hat{r}_V)$ of $\hat{P}_1$. If $\tilde{p}'_V \neq \tilde{p}_V$, $\tilde{q}'_V \neq \tilde{q}_V$, $c_V \neq \mathsf{AES}_{\mathsf{AES}_{r_V}(K'_V)}(0)$, $H'_{\bar{r}_V} \neq \mathsf{AES}_{K_V}(\bar{r}_V)$, or N is the not same value as learned by Sim_2 when it queried *construct* on $\mathcal{F}_{\mathsf{RSA}}$ in *Construct Modulus* then output $(\chi = 0, \perp, \perp)$. Else, using the value d_1 the simulator got from $\mathcal{F}_{\mathsf{RSA}}$ in the *construct* command and ρ_1 from $\hat{P}_1$'s input to the call to $\mathcal{F}_{\mathsf{2PC}}$. It then computes the value $d_1 - \rho_1 = \bar{d}_1$. Finally it picks a random $\delta \in \{0,1\}^{3\ell+2s+2}$, emulates $\mathcal{F}_{\mathsf{2PC}}$ to output $(\chi = 1, \delta, \bar{d}_V)$.

14. Send $\delta \mod N$ to $\hat{P}_1$.

Simulating V ($\hat{P}_1$ has the role of P):

1-2. Receive $\gamma'_{i,P}$ from $\hat{P}_1$.

3-4. Sim_2 verifies that $\gamma'_{i,P} = \pm\gamma_i^{\frac{N-5}{4} - \tilde{p}_1 - \tilde{q}_1} \mod N$ for each $i \in [s]$. If this is not the case, then it sends $\perp$ to $\mathcal{F}_{\mathsf{RSA}}$ and $\hat{P}_1$ and aborts.

5. Receive H'_{t_j} from $\hat{P}_1$.
6. Receive the values $\bar{\gamma}'_{i,j}$ from $\hat{P}_1$.
7. Sim_2 emulates $\mathcal{F}_{\mathsf{CT}}$ to sample bits $b_1, \ldots, b_s$ uniformly at random.
8. For each $j \in [s]$, Sim_2 receives the value v'_j.
9. For each $i, j \in [s]$ Sim_2 verifies that $\gamma_i^{v'_j} \equiv \bar{\gamma}'_{i,j} \cdot {\gamma'_{i,P}}^{b_j} \cdot \gamma_i^{-b_j \cdot \frac{N-5}{4}} \mod N$. If not it sends $\perp$ to $\hat{P}_1$ and $\mathcal{F}_{\mathsf{RSA}}$ and aborts.

10-11. Do nothing.

12-13. Sim_2 extracts $\hat{P}_1$'s input, $(\tilde{p}'_P, \tilde{p}'_P, \rho'_P, K'_P, \bar{r}'_P, \hat{r}'_P)$ to $\mathcal{F}_{\mathsf{2PC}}$. If $\tilde{p}'_P \neq \tilde{p}_P$, $\tilde{q}'_P \neq \tilde{q}_P$, $c_P \neq \mathsf{AES}_{\mathsf{AES}_{r_P}(K'_P)}(0)$, $H'_{\bar{r}_P} \neq \mathsf{AES}_{K'_P}(\bar{r}_P)$, or $H'_{t_j} \neq \mathcal{H}(v'_j + b_j \cdot (\tilde{p}_P + \tilde{q}_P)$ (using the value $\tilde{p}_P$ and $\tilde{q}_P$ defined in *Candidate generation*), or N is the same value as learned by Sim_2 when it queried *construct* on $\mathcal{F}_{\mathsf{RSA}}$ in *Construct Modulus* then Sim_2 emulates $\mathcal{F}_{\mathsf{2PC}}$ to output abort and subsequently aborts the protocol. Otherwise it emulates $\mathcal{F}_{\mathsf{2PC}}$ to complete successfully.

14-15. Receive $\hat{\delta}$ from $\hat{P}_1$ and abort if $\hat{\delta} \neq \hat{r}'_P \mod N$ for the extracted $\hat{r}_P$ or if $\bar{r}'_P \cdot (4\tilde{p}'_P + 4\tilde{q}'_P + 6) + \alpha_P + \beta_P \mod N \neq \sigma - s_V \mod N$.

Generate Shared Key: Use the *select* command on $\mathcal{F}_{\mathsf{RSA}}$ to accept the candidate (N, d_1, p_1, q_1).

Malicious P_2 We construct a simulator Sim_1:

Setup.

1. Emulate $\mathcal{F}_{\mathsf{CT}}$ twice by sampling $r_1, r_2 \in \{0,1\}^\kappa$ uniformly at random.

2. $\mathcal{I} = 1$: Pick $K_1 \in \{0,1\}^\kappa$ uniformly at random on behalf of of P_1, compute $\mathsf{AES}_{\mathsf{AES}_{r_1}(K_1)}(0) = c_1$ and send c_1 to $\hat{P}_2$.
2. $\mathcal{I} = 2$: Receive the message from $\hat{P}_2$, c_2.
3. $\mathcal{I} = 1$: Emulate $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ by returning $\top$ to $\hat{P}_2$.
3. $\mathcal{I} = 2$: Extract the input K_2 from $\hat{P}_2$ and emulate $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ correctly, by verifying that $\mathsf{AES}_{\mathsf{AES}_{r_2}(K_2)}(0) = c_2$, and output $\perp$ if that is not the case.
4. Output `abort` if $\perp$ was returned in step 3.

Candidate Generation:

1. Receive the value $H_{\tilde{p}_2}$ from $\hat{P}_2$. Use the key K_2 extracted from $\mathcal{F}_{\mathsf{ZK}}^{M_L}$ to learn the value encrypted $\tilde{p}_2$ and define $p_2 = 4 \cdot \tilde{p}_2$.
2. Sim_1 picks a uniformly random value $\tilde{p}_1 \in \mathbb{Z}_{2^{\ell-3}}$. Define $p_1 = 4\tilde{p}_1 + 3$ and $p_2 = 4\tilde{p}_2$. Sim_1 computes the commitment $H_{\tilde{p}_1} = \mathsf{AES}_{K_1}(\tilde{p}_1) \in \{0,1\}^\kappa$ and sends $H_{\tilde{p}_1}$ to $\hat{P}_2$.
3. If $p = p_1 + p_2$ is a prime and $\gcd(e, p-1) = 1$ then it marks this shared candidate prime, queries the *Sample* method of $\mathcal{F}_{\mathsf{RSA}}$ with p_2 and proceeds with the simulation by following the steps below of Div-OT for each $\beta \in \mathcal{B}$. Otherwise if p is not prime, or $\gcd(e, p-1) \neq 1$, then Sim_1 proceeds with the simulation of the rest of the protocol by executing like the true P_1, using the value p_1.
   1-2. Emulate random $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$ by picking random values $m_1, \ldots, m_\beta \in \{0,1\}^\kappa$ and giving these to $\hat{P}_2$.
   3. Receive m'_{a_2} from $\hat{P}_2$.
   4. If m'_{a_2} is not equal to one of the emulated values then return $\top$. Otherwise if m'_{a_2} is equal to one of the values above, then recover the index a_2. Sim_1 then uses *Leak* on $\mathcal{F}_{\mathsf{RSA}}$ to learn if $p_1 \mod \beta \stackrel{?}{\neq} a_2$. If it holds then return $\top$ to $\hat{P}_2$. Otherwise return $\perp$, break the loop and discard the candidate p.

Construct Modulus: Consider candidate values p and q constructed in the previous phase, if none of them is marked as prime, then Sim_1 simulates the rest of the protocol like P_1 using the shares p_1 and q_1 it constructed in simulation of *Candidate Generation*. If one of the values (assume w.l.o.g. p) is marked as prime and the other is not, then Sim_1 uses *Full-Leak* on $\mathcal{F}_{\mathsf{RSA}}$ to learn p_1 and proceeds by simulating P_1 using the p_1 it received from $\mathcal{F}_{\mathsf{RSA}}$ and the value q_1 it constructed in *Candidate Generation*, and marks the resulting N as bad. Finally, if both p and q are marked as prime, it uses the method *Construct* on $\mathcal{F}_{\mathsf{RSA}}$ and receives (N, d_2). It then proceeds as below:

1. Sim_1 picks $d_{\alpha,i} \in \{0,1\}$ uniformly at random for $\alpha \in \{p,q\}$ and $i \in [2\ell + 3s]$.
2. For each $\alpha \in \{p,q\}$ and $i \in [2\ell + 3s]$ Sim_1 extracts the inputs $c'_{1,\alpha,i}$ and $c'_{0,\alpha,i} = r'_{\alpha,i}$ from $\hat{P}_2$.
   Define E_p to be the set of indexes in $i \in [2\ell+3s]$ where $c'_{1,p,i} - c'_{0,p,i} \mod \mathcal{P} \neq q_2$. Define E_q similarly.

3. For $\alpha \in \{p, q\}$ and $i \in [2\ell+3s]$ Sim_1 picks the values $g_\alpha, h_{\alpha,i} \in \mathbb{Z}_{\mathcal{P}}$ uniformly at random and sends these to $\hat{P}_2$.
4. Sim_1 computes $\bar{z}_2^\alpha = -\sum_{i \in [2\ell+3s] \setminus E_\alpha} c'_{0,\alpha,i} \cdot h_{\alpha,i} \mod \mathcal{P}$ for $\alpha \in \{p, q\}$ along with:

$$\begin{aligned} a_1 = N &- \left(\sum_{i \in E_p} h_{p,i} \cdot d_{p,i} \cdot q_2 \right) - \left(\sum_{i \in E_q} h_{q,i} \cdot d_{p,i} \cdot p_2 \right) \\ &- (p_2 q_2 + \bar{z}_2^p + g_p \cdot q_2 + \bar{z}_2^q + g_q \cdot p_2) \\ &+ \left(\sum_{\alpha \in \{p,q\}} \sum_{i \in E_\alpha} c'_{d_{\alpha,i},\alpha,i} \cdot h_{\alpha,i} \right) \mod \mathcal{P} \ . \end{aligned}$$

5. The simulator receives a'_2 from $\hat{P}_2$.
6. Sim_1 then sends a_1 to $\hat{P}_2$.
7. Compute $\bar{N} = (a_1 + a'_2 \mod \mathcal{P}) \mod 2^{2\ell}$. If $\bar{N} \neq N$ then Sim_2 sets $N := \bar{N}$, thus using the incorrect value $\bar{N}$ in the rest of the simulation.

8-9. Emulate random $\mathcal{F}_{\mathsf{OT}}^{\kappa, \lceil \log(e) \rceil}$ by picking random values $r_0, \ldots, r_{\lceil \log(e) \rceil} \in \{0,1\}^\kappa$ and giving these to $\hat{P}_2$.

10-12. Receive a value r'_{w_2} from $\hat{P}_2$. If $r'_{w_2} = r_i$ for any $i \in [0, \lceil \log(e) \rceil[$ then give i as input to $\mathcal{F}_{\mathsf{RSA}}$ as part of the *Construct* command and thus finds out whether $w_2 = N + 1 - p_1 - q_1 \mod e$ or not. If the equality holds then the simulator informs $\hat{P}_2$ of this and discard the candidate N.

Trial Division: Sim_1 waits for $\hat{P}_2$ to input $\nu \in \{\top, \bot\}$. If $\nu = \bot$ then it discards the candidate N. If instead $\nu = \top$ it continues.

Biprimality Test:

1. Sim_1 repeats the steps below s times:
   (a) Sim_1 picks a random $x \in \mathbb{Z}_N^\times$, a random bit $b \in \{0,1\}$. It then computes $\gamma = (-1)^b \cdot x^2 \mod N$.
   (b) It then sends γ to $\hat{P}_2$.
   (c) Sim_1 then computes $\gamma_1 = (-1)^b \cdot \gamma^{\frac{p_2+q_2}{4}} \mod N$ and sends γ_1 to $\hat{P}_2$.
   (d) If $\hat{P}_2$ returns $\bot$ then Sim_1 breaks the loop and discards the candidate N. Otherwise it continues.
2. Sim_1 does as follows:
   (a) Store the value $H'_{\bar{r}_2}$ received from $\hat{P}_2$ and let $\bar{r}_2 = \mathsf{AES}_{K_2}(H'_{\bar{r}_2})$.
   (b) Sim_1 simulates the multiplication protocol by extracting the messages $\hat{P}_2$ input to the $\mathcal{F}_{\mathsf{OT}}^{3\ell+s+2,2}$ functionality when the simulator is acting as the receiver and picking a random input $\bar{r}_1 \in \mathbb{Z}_N$. When it is acting as the sender it picks a random value $p'_1 + q'_1 \in \mathbb{Z}_{2^{\ell+1}}$ obeying the leaked bits on the p_1 and q_1 that is learned from $\mathcal{F}_{\mathsf{RSA}}$ though under the constraint that $\gcd(N, p'_1 + p_2 + q'_1 + q_2 - 1) = 1$). Using this it emulates the OT messages an honest party would send. Next it picks a random input $\bar{r}_1 \in \mathbb{Z}_N$. It then computes the values α_1 and β_1 like an honest P_1 would, using these choices.

(c) Sim_1 computes $s_1 = \bar{r}_1 \cdot (p'_1 + q'_1) + \left(\alpha_1 + \beta_1 \mod 2^{3\ell+s+2}\right) - 2^{3\ell+s+2} \mod N$ and sends this to $\hat{P}_2$.
(d) Receive s'_2 from $\hat{P}_2$.
(e) Discard the candidate N if $\gcd(s_1 + s'_2, N) \neq 1$).

Proof of Honesty: Execute the simulation like Sim_2 with the exception that when simulating P, except in step 12 it outputs $d_2 - \rho_2 - 1 = \bar{d}_2$ instead of $d_2 - \rho_2 = \bar{d}_2$.

Generate Shared Key: Use the *select* command on $\mathcal{F}_{\mathsf{RSA}}$ to accept the candidate (N, p_2, q_2, d_2).

Leakage Before continuing with the indistinguishability argument we need the following Lemma, which intuitively shows that after $\mathsf{Sim}_{\mathcal{I}}$ has completed the *Construct Modulus* phase then, for a candidate $N = (p_1 + p_2)(q_1 + q_2)$ which is not marked as bad, the leakage on p_1, q_1 towards $\hat{P}_2$ and p_2, q_2 towards $\hat{P}_1$ is not too much.

Lemma C.1. *Assume* $13 \leq B_1 \leq 6000$, $512 \leq \ell$, *and* $28 \leq s$ *and let* $N = (p_1 + p_2)(q_1 + q_2)$ *be a candidate modulus not marked as* bad, *then leakage on* $(\tilde{p}_{\mathcal{I}}, \tilde{q}_{\mathcal{I}})$ *is at most* $2\log_2(B_1/2)$ *bits towards* $P_{3-\mathcal{I}}$ *in phase* Candidate Generation. *If* $\mathcal{I} = 2$ *then the leakage on* (p_2, q_2) *in the view of* P_1 *will further be increased by at most* $2\log_2(\beta)$ *bits with probability at most* $2/(\beta - 1)$ *for each* $\beta \in \mathcal{B}$ *used in phase* Candidate Generation.

Proof. First notice that both $\tilde{p}_{\mathcal{I}}$ and $\tilde{q}_{\mathcal{I}}$ are picked uniformly at random and consists of $\ell - 3$ bits. Thus each of them have $\ell - 3$ bits of entropy to start with. Furthermore see that $p_{\mathcal{I}}$ and $q_{\mathcal{I}}$ are computed as $4\tilde{p}_{\mathcal{I}} + 3$, $4\tilde{q}_{\mathcal{I}} + 3$ or $4\tilde{p}_{\mathcal{I}}$, $4\tilde{q}_{\mathcal{I}}$. Thus they still have $\ell - 3$ bits of entropy. Next, see that if $\mathcal{I} = 1$ then the two least significant bits of $p_{\mathcal{I}}$ will always be 0 and if $\mathcal{I} = 2$ they will always be 1, the rest will be the same random values as $\tilde{p}_{\mathcal{I}}$, respectively $\tilde{q}_{\mathcal{I}}$.

Consider $\mathcal{I} = 1$: See that the simulator Sim_1 only gets leakage in *Candidate Generation* when it learns if $\alpha_1 \mod \beta \neq a_2$ for all $\beta \in \mathcal{B}$ and $\alpha \in \{p, q\}$. However if it does not hold the candidate is rejected. Now since $a_2 \in [\beta]$ there are $\beta - 1$ possible values for which the inequality $\alpha_1 \mod \beta \neq a_2$ holds. This means that what is leaked, could in the worst case, allow the adversary to reduce the set of possible values of p_1 from $2^{\ell-3}$ to $\frac{\beta-1}{\beta} \cdot 2^{\ell-3} = 2^{\ell-3-\log_2(\beta-1/\beta)}$. I.e. leaking at *most* $\log_2(\beta/(\beta-1))$ bits. This means that in total at most $\sum_{\beta\in\mathcal{B}} \log_2(\beta/(\beta-1))$ bits for each $\alpha \in \{p, q\}$, so $2\sum_{\beta\in\mathcal{B}} \log_2(\beta/(\beta-1))$ in total. Thus we have that

$$
\begin{aligned}
2\sum_{\beta\in\mathcal{B}} \log_2\left(\frac{\beta}{\beta-1}\right) &= 2\log_2\left(\prod_{\beta\in\mathcal{B}} \frac{\beta}{\beta-1}\right) \\
&\leq 2\log_2\left(\prod_{3\leq\beta\leq B_1} \frac{\beta}{\beta-1}\right) \leq 2\log_2\left(\frac{B_1}{2}\right)
\end{aligned}
$$

Now consider $\mathcal{I} = 2$: See that the simulator Sim_2 only gets leakage in *Candidate Generation* to learn if $-\tilde{\alpha}_2 \mod \beta \neq a_1$ for all $\beta \in \mathcal{B}$ and $\alpha \in \{p, q\}$. See that if the inequality holds, then the same arguments go as when $\mathcal{I} = 1$. If the inequality does *not* hold, then notice that the set of possible values of α_2 gets reduced by β, as there is only one case out of β when the inequality does not hold, i.e. when $-\alpha_2 \mod \beta = a_1$ since $a_1 \in [\beta]$. This means that what is leaked, could in the worst case, allow the adversary to reduce the set of possible values of p_2 from $2^{\ell-3}$ to $\frac{1}{\beta} \cdot 2^{\ell-3} = 2^{\ell-3-\log_2(\beta)}$. This means that in total at most $\sum_{\beta\in\mathcal{B}} \log_2(\beta)$ bits for each $\alpha \in \{p, q\}$, so $2\sum_{\beta\in\mathcal{B}} \log_2(\beta)$ in total.

The adversary has no info on $\tilde{p}_2$ and $\tilde{q}_2$ a priori. Next see that $\tilde{p}_2$ and $\tilde{q}_2$ are randomly picked, it means that $\tilde{p}_2 \mod \beta$ is "almost" uniformly randomly distributed in $[\beta]$. There is a slight bias towards the numbers from 0 to $2^{\ell-3} - 1 \mod \beta$. This means that in the most lucky case it is only 0 that is biased. However, it only means that the probability is $\lfloor (2^{\ell-3} - 1)/\beta \rfloor / 2^{\ell-3}$ for non-zero choices and $\lceil 2^{\ell-3}/\beta \rceil / 2^{\ell-3}$ for 0. Thus the probability that the inequality does *not* hold and so that the most information is leaked, is

$$\frac{\lceil \frac{2^{\ell-3}}{\beta} \rceil}{2^{\ell-3}} \leq \frac{\frac{2^{\ell-3}+\beta}{\beta}}{2^{\ell-3}} = \frac{\frac{2^{\ell-3}}{\beta+1}}{2^{\ell-3}} = \beta^{-1} + 2^{-\ell+3}$$

Furthermore, see that $2^{-\ell+3} \leq 2^{-s} \leq 2^{-\log(\beta)}$. Thus we get $\beta^{-1} + 2^{-\ell+3} = 2^{-\log(\beta)} + 2^{-\ell+3} \leq 2^{-\log((\beta-1)/2)} = 2/(\beta - 1)$ since we assume $\ell \geq 512$ and $B_1 < 6000$.

Corollary C.2. *The leakage on $(p_\mathcal{I}, q_\mathcal{I})$ towards $P_{3-\mathcal{I}}$ is at most $\log_2(e/(e-1)) + 2\log_2(B_1/2)$. If $\mathcal{I} = 2$ with probability at most 2^{-x}, $(1+\epsilon)x$ bits further are leaked for $\epsilon \leq 1$ and x picked by $P_{3-\mathcal{I}}$.*

Proof. From Lemma C.1 we get the value $2\log_2(B_1/2)$. However, $1/e$ bits are further leaked. This leakage comes from the last couple of steps of *Construct Modulus* as the parties will learn whether $w_1 = w_2$ or not. If they are not equal the candidate is passed on, otherwise it is rejected. However, we notice that nothings prevents a malicious party $P_\mathcal{I}$ from probing equality on an arbitrary $w'_\mathcal{I}$ instead of $w_\mathcal{I}$. When this party is P_1 it can at most learn that $w_2 \neq w'_1$ which reduces the of possible values of w_1 from e to $e-1$, thus leaking at most $\log_2(e/(e-1))$ bits of information. Furthermore, notice that if $w'_1 = w_2$ then the candidate is discarded by an honest P_2 and thus the malicious P_1 learns nothing on the candidate that will eventually be used. If instead P_2 is malicious it will learn whether $w'_2 = w_1$, but will be allowed the option of continuing with the candidate. In which case it learns at most $\log_2(e)$ bits of information, but with probability at most $1/e$. Furthermore, for $\mathcal{I} = 2$, we get from Lemma C.1 that with probability at most $2^{-\log((\beta-1)/2)}$, $\log(\beta)$ bits are leaked to P_1. The maximum payoff is for as small $\beta > 2$ as possible, i.e. $\beta = 3$. The adversary can do this twice for each β, once for $\tilde{p}_2$ and once for $\tilde{q}_2$. Thus its optimal strategy is to do this first for $\beta = 3$, then $\beta = 5$ and so on. In any case it is easy to see that there exists some ϵ fulfilling the equation. In particular for $\beta = 3$ this

$\epsilon = 1$. Finally see that we can disregard the $\log_2(e)$ bits that might be leaked with probability $1/e$ since this gives a lower payoff as it reflects the case of $\epsilon = 1$ since $e \geq 3$.

Indistinguishability Using these simulators we prove security with a hybrid argument. Intuitively we construct a series of hybrids. Based on these hybrids we iteratively argue indistinguishability of each of the phases using the simulator for P_2, P_1 respectively, until we have showed that the simulators can simulate the honest parties in the protocol.

Our hybrid argument is based on the following hypothetical simulators working with the ideal functionality $\mathcal{F}_{\mathsf{RSA}}$ for $\mathcal{I} \in \{1, 2\}$:

$\mathsf{Sim}_{\mathcal{I}}^0$: Receive the internal randomness of $P_{\mathcal{I}}$ as auxiliary input and use this to generate all potential candidates, $\tilde{p}_{\mathcal{I}}$ exactly like the real party $P_{\mathcal{I}}$. However, let the simulator pick a uniformly random value $K'_{\mathcal{I}} \in \{0,1\}^{\kappa}$ and use this instead of $K_{\mathcal{I}}$ picked in *setup* in the entire protocol.

$\mathsf{Sim}_{\mathcal{I}}^1$: Act like $\mathsf{Sim}_{\mathcal{I}}^0$ except that in *Candidate Generation*, if p is marked as prime then execute *Candidate Generation* like $\mathsf{Sim}_{\mathcal{I}}$. Otherwise execute the rest of the simulation like $P_{\mathcal{I}}$ would, using its shares.

$\mathsf{Sim}_{\mathcal{I}}^2$: Acts like $\mathsf{Sim}_{\mathcal{I}}^1$ except that it simulates phase *Construct Modulus* like $\mathsf{Sim}_{\mathcal{I}}$.

$\mathsf{Sim}_{\mathcal{I}}^3$: Acts like $\mathsf{Sim}_{\mathcal{I}}^2$ except that it simulates all phases up to and including *Biprimality Test* like $\mathsf{Sim}_{\mathcal{I}}$.

$\mathsf{Sim}_{\mathcal{I}}^4$: Acts like $\mathsf{Sim}_{\mathcal{I}}^3$ except that it simulates all phases up to and including *Proof of Honesty* like $\mathsf{Sim}_{\mathcal{I}}$ and outputs $\perp$ in the end of *Proof of Honesty* iff the function f would output $\chi = 0$ when queried by the extracted values from the adversary and the true values of $P_{\mathcal{I}}$.

$\mathsf{Sim}_{\mathcal{I}}^5$: Acts like $\mathsf{Sim}_{\mathcal{I}}^3$ except that it simulates all phases up to and including *Generate Shared Key* like $\mathsf{Sim}_{\mathcal{I}}$.

Setup - $\mathsf{Sim}_{\mathcal{I}}^0$ We now argue indistinguishability between the real execution and $\mathsf{Sim}_{\mathcal{I}}^0$. We start by noticing if the environment can distinguish between execution in the real world and $\mathsf{Sim}_{\mathcal{I}}^0$ then it must necessarily mean that it can distinguish between $c_{\mathcal{I}}$ and $c'_{\mathcal{I}} = \mathsf{AES}_{\mathsf{AES}_{r_{\mathcal{I}}}(K'_{\mathcal{I}})}(0)$. However, since we assume AES is a PRP, we have that $\mathsf{AES}_{r_{\mathcal{I}}}(K_{\mathcal{I}}) = k_{\mathcal{I}}$ is indistinguishable from a random string if the adversary was given this value since $K_{\mathcal{I}}$ is uniformly random (remember the simulator acts as the honest party). Thus the same goes for $\mathsf{AES}_{r_{\mathcal{I}}}(K'_{\mathcal{I}}) = k'_{\mathcal{I}}$. Now to argue that $c_{\mathcal{I}}$ is also indistinguishable from random we see since AES is a PRP then $\mathsf{AES}_{k_{\mathcal{I}}}(0) = c_{\mathcal{I}}$ is indistinguishable from random since $k_{\mathcal{I}}$ is. The same for $\mathsf{AES}_{k'_{\mathcal{I}}}(0) = c'_{\mathcal{I}}$. Next see that the only place where $K_{\mathcal{I}}$ is used (besides as input to ideal functionalities) is as the encryption key. However, repeating the same argument again, since both $K_{\mathcal{I}}$ and $K'_{\mathcal{I}}$ are random (since they are honestly picked) it means that the first block of the encryption of $\tilde{p}_{\mathcal{I}}$ under two different, random keys, would be indistinguishable from random and so we can replace $K_{\mathcal{I}}$ with $K'_{\mathcal{I}}$.

Setup - $\mathsf{Sim}^1_{\mathcal{I}}$. We now argue indistinguishability between $\mathsf{Sim}^0_{\mathcal{I}}$ and $\mathsf{Sim}^1_{\mathcal{I}}$. We start by noticing that it is sufficient to only consider adversaries which compute $c_{3-\mathcal{I}} = \mathsf{AES}_{\mathsf{AES}_{r_{3-\mathcal{I}}}(K_{3-\mathcal{I}})}(0)$ correctly and uses the same $K_{3-\mathcal{I}}$ in the zero-knowledge proof $\mathcal{F}^{M_L}_{\mathsf{ZK}}$ in *Setup*. To see this notice that if $\mathcal{F}^{M_L}_{\mathsf{ZK}}$ outputs $\top$ then it must hold that $c_{\mathcal{I}} = \mathsf{AES}_{\mathsf{AES}_{r_{3-\mathcal{I}}}(K'_{3-\mathcal{I}})}(0)$ for some $K'_{3-\mathcal{I}}$. However, if $K'_{3-\mathcal{I}} \neq K_{3-\mathcal{I}}$ then $\mathsf{AES}_{r_{3-\mathcal{I}}}(K_{3-\mathcal{I}}) \neq \mathsf{AES}_{r_{3-\mathcal{I}}}(K'_{3-\mathcal{I}})$ since AES is a PRP. Thus also means that $c_{3-\mathcal{I}} \neq c'_{3-\mathcal{I}}$. This means that the value $K_{3-\mathcal{I}}$ extracted by the simulator from $\mathcal{F}^{M_L}_{\mathsf{ZK}}$ will match the double encryption when $\mathcal{F}^{M_L}_{\mathsf{ZK}}$ is emulated to output $\top$ (if not the simulator makes $\mathcal{F}^{M_L}_{\mathsf{ZK}}$ output $\perp$ just like in the real execution).

Candidate Generation. In *Candidate Generation* we notice that for the value $\tilde{p}_{3-\mathcal{I}}$ the simulator decrypts from $H_{\tilde{p}_{3-\mathcal{I}}}$ if $p = 4(\tilde{p}_1 + \tilde{p}_2) + 3$ is not marked as prime then the execution will be indistinguishable to the execution with $P_{\mathcal{I}}$, since the simulation will directly execute like $P_{\mathcal{I}}$ using its true randomness. Next see that if p is marked as prime then $\mathsf{Sim}^1_{\mathcal{I}}$ will not use the true value $\tilde{p}_{\mathcal{I}}$ to compute $H_{\tilde{p}_{\mathcal{I}}} = \mathsf{AES}_{K'_{\mathcal{I}}}(\tilde{p}_{\mathcal{I}})$ but instead sample a random value $\tilde{p}'_{\mathcal{I}}$ instead. See that $H_{\tilde{p}'_{\mathcal{I}}} = \mathsf{AES}_{K'_{\mathcal{I}}}(\tilde{p}'_{\mathcal{I}})$ is indistinguishable from $H_{\tilde{p}_{\mathcal{I}}}$ follows directly from the IND-CPA assumption as we have already argued that $K'_{\mathcal{I}}$ is unknown in the view of the adversary and randomly sampled.

Next consider the first trial division:

For $\mathcal{I} = 1$: Sim^1_1 emulates $\mathcal{F}^{\kappa,\beta}_{\mathsf{OT}}$ exactly like the real world and the only remaining message sent to $\hat{P}_2$ in *Construct Candidate* is $\nu = \{\top, \perp\}$. We know that p is a prime (otherwise the simulation would exactly follow that of the honest P_1), so it follows from correctness that $\top$ *should* be returned to $\hat{P}_2$ for each $\beta \in \mathcal{B}$. Next, after receiving m'_{a_2} sent in step b, Sim^1_1 extracts the value a_2, by comparing m'_{a_2} with the list $m_0, \ldots, m_{\beta-1}$. First note that no two values m_i, $m_{i'}$ for $i \neq i'$ are equal e.w.n.p. in κ, so m'_{a_2} will match at most a single item. If there is a match then Sim^1_1 uses *Leak* on $\mathcal{F}_{\mathsf{RSA}}$ to learn whether $a_2 \neq p_1 \mod \beta$. If this does *not* hold, then Sim^1_1 returns $\perp$. Otherwise it returns $\top$ exactly like what would happen in the real execution. Thus $\top$ is returned even if m'_{a_2} does not match any of the messages in $\mathcal{F}^{\kappa,\beta}_{\mathsf{OT}}$. However, this is exactly like it would happen in the execution with P_1.

For $\mathcal{I} = 2$: Sim^1_2 emulates $\mathcal{F}^{\kappa,\beta}_{\mathsf{OT}}$ by selecting a random value $m_{a_1} \in \{0,1\}^{\kappa}$ and returning this to $\hat{P}_1$ and extracting its input message a'_1. This m_{a_1} is clearly distributed the same as in the real execution with P_1, since this is also randomly sampled. Next we notice that if $a_1 = -p_2 \mod \beta$, then Sim^1_2 returns the same $m = m_{a_1}$ again, exactly like what would happen in the real execution. Otherwise it returns a random m which is clearly indistinguishable from what would be sent in the real execution since there it would also be a random value completely unknown to $\hat{P}_1$.

Finally notice that no abort is issued by the honest party in *Candidate Generation*. Thus then abort probability remains the same between $\mathsf{Sim}^1_{\mathcal{I}}$ and the real execution. Finally, notice that all further phases of the protocol is executed

in exactly the same way, using exactly the same values, whether using $\mathsf{Sim}^1_{\mathcal{I}}$ or $P_{\mathcal{I}}$.

Construct Modulus. First see that if both candidate values are not marked as prime, then there is nothing to show. This follows since the simulator $\mathsf{Sim}^2_{\mathcal{I}}$ then executes like $P_{\mathcal{I}}$ using the simulated values, or one simulated value and one true value (which it learns using *Full-leak*).

Before we continue to argue indistinguishability, we need the following definitions and helper lemma:

Definition C.3. *Let $\mathcal{H}$ define a family of hash functions $H_{h_1,\ldots,h_n,g} : \{0,1\}^n \to \mathbb{Z}_{\mathcal{P}}$ where $g, h_i \in \mathbb{Z}_{\mathcal{P}}$ for $i \in [n]$ and $H_{h_1,\ldots,h_n,g}(d) := g + \sum_{i\in[n]} h_i \cdot d_i \mod \mathcal{P}$ where $d \in \{0,1\}^n$ with d_i denoting the i'th bit of d.*

Lemma C.4. *The family of hash functions, $\mathcal{H}$, is 2-universal.*

Proof. Remember the definition of 2-universality says it must hold that for any pair of two, distinct inputs and a randomly sampled hash function from the family, the probability of collision is at most one over the size of the range. Or more formally

$$\forall x, y \in \{0,1\}^n, x \neq y \ : \ \Pr_{H_{h_1,\ldots,h_n,g} \in_R \mathcal{H}}(H_{h_1,\ldots,h_n,g}(x) = H_{h_1,\ldots,h_n,g}(y)) \leq \mathcal{P}^{-1}$$

We wish to show that this holds for $\mathcal{H}$ defined above in Def. C.3. First notice that a collision occurs when $g + \sum_{i\in[n]} h_i \cdot x_i \equiv g + \sum_{i\in[n]} h_i \cdot y_i \mod \mathcal{P}$. In particular this means that a collision occurs when $\sum_{i\in[n]} h_i \cdot (x_i - y_i) \equiv 0 \mod \mathcal{P}$. Since $x \neq y$ there must be at least one position $i \in [n]$ where $x_i \neq y_i$ and thus $x_i - y_i \neq 0$. If there is only one such position then the probability that $\sum_{i\in[n]} h_i \cdot (x_i - y_i) \equiv 0 \mod \mathcal{P}$ is at most $\mathcal{P}^{-1}$, since h_i is sampled uniformly at random from $\mathbb{Z}_{\mathcal{P}}$. If there are more than 1 such position then we will have a sum of uniformly random picked numbers of $\mathbb{Z}_{\mathcal{P}}$. Thus the probability that they sum to 0 (or any other value in $\mathbb{Z}_{\mathcal{P}}$ for that matter) is at most P^{-1}. Hence it follows that the hash function is 2-universal.

Next we define the following distribution reflecting the choice of P_1 in the protocol:

Definition C.5. *For $x \in \mathbb{Z}_{\mathcal{P}}$ let $\mathcal{D}^n_x$ be the distribution induced by $(h_1, \ldots, h_n, g)$ s.t. $g + \sum_{i\in[n]} h_i \cdot d_i \mod \mathcal{P} = x$ where $d_i \in \{0,1\}$ is a randomly picked pattern.*

We can now prove the following:

Lemma C.6. *Let $n \geq \lceil \log_2(\mathcal{P}) \rceil + 2s > \log_2(\mathcal{P}) + 2s$. Then, for all $x \in \{0,1\}^{\lceil \log_2(\mathcal{P}) \rceil + 2s}$ the statistical distance between the distribution of $\mathcal{D}^n_x$ and the uniform distribution over $\mathcal{H}$ is at most 2^{-s}.*

Proof. For this proof we piggyback on the proof of Lemma 1 in [IPS09], as this proves the same statement, but for a different hash function.

First see that we have from Lemma C.4 that $\mathcal{H}$ in accordance with Def. C.3 is 2-universal. Thus the Leftover Hash Lemma tells us that:

$$\mathrm{SD}(H_{h_1,\ldots h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}(x), \mathcal{U}_{\lceil \log_2(\mathcal{P})\rceil+2s}) \leq 2^{-s} \ ,$$

when x is uniformly random sampled from $\{0,1\}^{\lceil \log_2(\mathcal{P})\rceil+2s}$. This follows since the min-entropy of x is $\lceil \log_2(\mathcal{P})\rceil + 2s$ and thus

$$\log_2(\mathcal{P}) + 2\log(\epsilon^{-1}) \leq \lceil \log_2(\mathcal{P})\rceil + 2s \Rightarrow \log_2(\mathcal{P}) + 2s \leq \lceil \log_2(\mathcal{P})\rceil + 2s,$$

when $\epsilon = 2^{-s}$.

To finish the proof we must argue a symmetry between the different outcomes of the hash function, since we need the distance to hold for all possible inputs and not only a randomly sampled input. To do so, consider a family of permutations $\{\pi_a | a \in \mathbb{Z}_{\mathcal{P}}\}$ s.t. for all $z \in \mathbb{Z}_{\mathcal{P}}$, $\Pr(z|H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}) = \Pr(z + a|\pi_a(H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}))$. Here $\Pr(z|H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g})$ means the probability of getting output value z when applying the specific hash function $H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}$ on a randomly sampled input. More formally

$$\Pr(z|H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}) := \Pr_{x \in_R \{0,1\}^{\lceil \log_2(\mathcal{P})\rceil+2s}}(H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}(x) = z) \ .$$

Next notice that we have $\pi_a(H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g'}) = H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}$ for any choice of $h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s}, g, g' \in \mathbb{Z}_{\mathcal{P}}$ by simply defining $a = g - g' \mod \mathcal{P}$. Now observe the following:

$$\begin{aligned}
&\mathrm{SD}((\mathcal{H}, H_{h_1,\ldots h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}(\cdot)), (\mathcal{H}, \mathcal{U}_{\lceil \log_2(\mathcal{P})\rceil+2s}) \\
&\quad =^{def} \frac{1}{2}\frac{1}{|\mathcal{H}|} \sum_{H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g} \in \mathcal{H}} \sum_{z \in \mathbb{Z}_{\mathcal{P}}} |\Pr(z|H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}) - \Pr(z|\mathcal{U})| \\
&= \frac{1}{2|\mathcal{H}|} \cdot \sum_{H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g} \in \mathcal{H}} \sum_{z \in \mathbb{Z}_{\mathcal{P}}} \left(|\Pr(z|H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g}) - \mathcal{P}^{-1}| \right) \\
&= \frac{\mathcal{P}}{2|\mathcal{H}|} \cdot \sum_{z \in \mathbb{Z}_{\mathcal{P}}} \sum_{H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g} \in \mathcal{H}} \left(|\Pr(x|\pi_{x-z}(H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g})) - \mathcal{P}^{-1}| \right) \\
&= \frac{\mathcal{P}}{2|\mathcal{H}|} \cdot \sum_{H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g'} \in \mathcal{H}} \left(|\Pr(x|H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g'}) - \mathcal{P}^{-1}| \right)
\end{aligned}$$

For the first step notice that the sum is multiplied with $\frac{1}{|\mathcal{H}|}$ since the first part of the two distributions we compare, the sampling of $H_{h_1,\ldots,h_{\lceil \log_2(\mathcal{P})\rceil+2s},g} \in \mathcal{H}$, will be the same in both distributions. This means that the difference in probability of each possible value of z is counted $|\mathcal{H}|$ times, once for each possible $H_{h_1,\ldots h_{\lceil \log_2(\mathcal{P})\rceil+2s},g} \in \mathcal{H}$. So multiplying with $|\mathcal{H}|^{-1}$ gives us the average. By applying Bayes' Theorem, $\Pr(x|H) = \frac{\Pr(H|x)\Pr(x)}{\Pr(H)}$, we then have that

$\Pr(x|H_{h_1,\ldots,h_{\lceil\log_2(\mathcal{P})\rceil+2s},g}) = \Pr(H_{h_1,\ldots,h_{\lceil\log_2(\mathcal{P})\rceil+2s},g}|x) \cdot \frac{\mathcal{P}}{|\mathcal{H}|}$ and thus:

$$\frac{\mathcal{P}}{2|\mathcal{H}|} \cdot \sum_{z\in\mathbb{Z}_{\mathcal{P}}} \sum_{H_{h_1,\ldots,h_{\lceil\log_2(\mathcal{P})\rceil+2s},g'}\in\mathcal{H}} \left(|\Pr(x|H_{h_1,\ldots,h_{\lceil\log_2(\mathcal{P})\rceil+2s},g'}) - \mathcal{P}^{-1}|\right)$$

$$= \frac{1}{2} \cdot \sum_{H_{h_1,\ldots,h_{\lceil\log_2(\mathcal{P})\rceil+2s},g'}\in\mathcal{H}} \left(|\Pr(H_{h_1,\ldots,h_{\lceil\log_2(\mathcal{P})\rceil+2s},g'}|x) - |\mathcal{H}|^{-1}|\right)$$

Now notice that this is in fact the statistical distance between distribution $\mathcal{D}_x^{\lceil\log_2(\mathcal{P})\rceil+2s}$ from Def. C.5 and $\mathcal{U}_{|\mathcal{H}|}$ (sampling a hash function from the family), thus finishing the proof.

With this we can now argue indistinguishability. First considering the case of $\mathcal{I} = 1$:

Consider an intermediate hybrid where Sim_1^1 picks $d_{\alpha,i}$ for $i \in [2\ell + 3s]$ uniformly at random in step 1. Then in step 3 it picks $h_{\alpha,i} \in \mathbb{Z}_{\mathcal{P}}$ uniformly at random for $i \in E_\alpha$, where E_α is an arbitrary set of positions. Next it picks $h_{\alpha,i}, g \in \mathbb{Z}_{\mathcal{P}}$ for $i \in [2\ell + 3s]\backslash E_\alpha$ uniformly at random under the constraint that $g + \sum_{i\in[2\ell+3s]\backslash E_\alpha} h_{\alpha,i} \cdot d_{\alpha,i} \mod \mathcal{P} = \alpha_1 - \sum_{i\in E_\alpha} h_{\alpha,i} \cdot d_{\alpha,i} \mod \mathcal{P}$. It is easy to see that this is clearly possible, for example by simply choosing g appropriately, no matter what values of $d_{\alpha,i}$ were picked. It is furthermore easy to see that the intermediate hybrid is indistinguishable from the hybrid induced by Sim_1^1 since the values are still picked at random, under the same constraint as in Sim_1^1.

Next assume that at most s bits are leaked on d_p in the intermediate hybrid and Sim_1^2 (we show this in a moment). Split d_p up into the bits which are leaked and the bits which are not leaked, letting E_p denote the set of indexes of bits leaked. Thus the bits $d_{p,i}$ for $i \in [2\ell+3s]\backslash E_p$ are uniformly random sampled and completely unknown to the adversary in both hybrids. Thus, by Def. C.3 and Lemma C.4 we have that $h_{\alpha,i}, g$ for $i \in [2\ell+3s]\backslash E_p$ defines a two-universal hash function from the family $\mathcal{H}$ where the digest will have value $p_1 - \sum_{i\in E_p} h_{p,i} \cdot d_{p,i} \mod \mathcal{P}$ in the intermediate hybrid. Thus from Lemma C.6 we get that the statistical distance between the output of this hash function in the intermediate hybrid and the random one defined for Sim_1^2 is 2^{-s} indistinguishable from random. This follows since in the intermediate hybrid the hash function is picked to hit $a = p_1 - \sum_{i\in E_p} h_{p,i} \cdot d_{p,i} \mod \mathcal{P}$ (and thus taken from the distribution $\mathcal{D}_a^{2\ell+2s}$) and in Sim_1^2 it is simply picked uniformly at random.

Thus the last thing we need to argue to get the argument above to go through, is that at most s bits on $d_{\alpha,i}$ is leaked. To do so notice that it is only used to compute z_1^α and thus a_1. Thus any leakage on $d_{\alpha,i}$ will be on a_1 as the OT functionality is perfect. This follows from a standard selective failure attack argument, as the adversary can guess x bits on $(d_{p,1}, d_{q,1})$ by sending incorrect values for one of the choices in the OT. It will notice whether it guessed correct in *Proof of Honesty*, since in this case N will be correct based on the hash digests. Since they are random it can only succeed with probability 2^{-x} and because we assume statistical security s, it cannot succeed in guessing more than s positions with probability greater than 2^{-s}. Furthermore, since each bit is random and

independent, this gives exactly s bits of entropy. This finishes the argument of indistinguishability of the first 3 steps.

Next consider the values computed and sent in step 4-6. We must argue that in step 4, the a_1 value simulated by Sim_1^2 is indistinguishable from the true value a_1 that P_1 would send, i.e. what is sent by Sim_1^1. First notice from correctness that if $|E_p| + |E_q| = 0$ then $\bar{z}_2^\alpha = z_2^{\alpha\prime}$ and so we have that $pq = N = p_1q_1 + z_p^1 + z_q^1 + p_2q_2 + z_2^{p\prime} + g_p \cdot q_2 + z_2^{q\prime} + g_q \cdot p_2 \mod \mathcal{P}$. Thus setting $a_1 = N - (p_2q_2 + z_2^{p\prime} + g_p \cdot q_2 + z_2^{q\prime} + g_q \cdot p_2) \mod \mathcal{P}$ means that $a_1 = p_1q_1 + z_1^p + z_1^q \mod \mathcal{P}$, i.e. the same in both Sim_1^1 and Sim_1^2.

Now note that if $c'_{1,p,i} - c'_{0,p,i} \mod \mathcal{P} = q_2$ then $c'_{d_{p,i},p,i} = r_{0,p,i} + q_2 \cdot d_{p,i} \mod \mathcal{P}$. Thus $c'_{d_{p,i},p,i} - c'_{0,p,i} \mod \mathcal{P} = q_2 \cdot d_{p,i}$. Thus if $\hat{P}_2$ sends correct messages we have that for any subset E_α of $[2\ell + 3s]$, we can still simulate the value of P_1. We see from the following, by starting to consider the result of $a_1 + a_2'$, when the positions in E_α are excluded by both parties:

$$
\begin{aligned}
&N - \left(\sum_{i \in E_p} h_{p,i} \cdot d_{p,i} \cdot q_2\right) - \left(\sum_{i \in E_q} h_{q,i} \cdot d_{q,i} \cdot p_2\right) \\
&\quad \equiv (p_1q_1 + z_1^p + z_1^q) + (p_2q_2 + z_2^{p\prime} + g_p \cdot q_2 + z_2^{q\prime} + g_q \cdot p_2) - \\
&\qquad \sum_{\alpha \in \{p,q\}} \sum_{i \in E_\alpha} h_{\alpha,i} \cdot (c'_{d_{\alpha,i},\alpha,i} - c'_{0,\alpha,i}) \mod \mathcal{P} \\
&N - (p_2q_2 + z_2^{p\prime} + g_p \cdot q_2 + z_2^{q\prime} + g_q \cdot p_2) - \left(\sum_{i \in E_p} h_{p,i} \cdot d_{p,i} \cdot q_2\right) \\
&\qquad - \left(\sum_{i \in E_q} h_{q,i} \cdot d_{q,i} \cdot p_2\right) + \left(\sum_{\alpha \in \{p,q\}} \sum_{i \in E_\alpha} h_{\alpha,i} \cdot (c'_{d_{\alpha,i},\alpha,i} - c'_{0,\alpha,i})\right) \\
&\quad \equiv p_1q_1 + z_1^p + z_1^q \mod \mathcal{P} \\
&N - (p_2q_2 + \bar{z}_2^q + g_p \cdot q_2 + \bar{z}_2^q + g_q \cdot p_2) - \left(\sum_{i \in E_p} h_{p,i} \cdot d_{p,i} \cdot q_2\right) \\
&\qquad - \left(\sum_{i \in E_q} h_{q,i} \cdot d_{q,i} \cdot p_2\right) \\
&\quad \equiv p_1q_1 + \bar{z}_1^p + \bar{z}_1^q \mod \mathcal{P}
\end{aligned}
$$

We here use the notation that $\bar{z}_1^\alpha = \sum_{i \in [2\ell+3s] \setminus E_\alpha} h_{\alpha,i} \cdot c'_{1,\alpha,i} \mod \mathcal{P}$ and $\bar{z}_2^\alpha = \sum_{i \in [2\ell+3s] \setminus E_\alpha} -h_{\alpha,i} \cdot c'_{0,\alpha,i} \mod \mathcal{P}$. Notice that we have completely removed the values $c'_{0,\alpha,i}$ and $c'_{1,\alpha,i}$ for $i \in E_\alpha$, thus the above hold unconditionally of what $\hat{P}_2$ sends. Next see that if we add $\sum_{\alpha \in \{p,q\}} \sum_{i \in E_\alpha} h_{\alpha,i} \cdot c'_{d_{\alpha,i},\alpha,i} \mod \mathcal{P}$ to both sides we have the following, independent of whether $\hat{P}_2$ sends incorrect values or

not:

$$N - (p_2 q_2 + \bar{z}_2^p + g_p \cdot q_2 + \bar{z}_2^q + g_q \cdot p_2) - \sum_{i \in E_p} h_{p,i} \cdot d_{p,i} \cdot q_2 - \sum_{i \in E_q} h_{q,i} \cdot d_{q,i} \cdot p_2$$
$$+ \sum_{\alpha \in \{p,q\}} \sum_{i \in E_\alpha} h_{\alpha,i} \cdot c'_{d_{\alpha,i},\alpha,i}$$
$$\equiv p_1 q_1 + z_1^p + z_1^q \equiv a_1 \mod \mathcal{P}$$

Which will then be exactly how a_1 is computed in Sim_1^2. In particular this means that the value $a_1 + a'_2 \mod \mathcal{P}$ will be the same value whether executing using Sim_1^1 or Sim_1^2.

For the remaining steps 8-12, first notice that Sim_1^1 emulates $\mathcal{F}_{\mathsf{OT}}^{\kappa,\lceil \log(e) \rceil}$ exactly like the real world and the only remaining message sent to $\hat{P}_2$ in these steps is $\nu = \{\top, \bot\}$. $\top$ *should* be returned to $\hat{P}_2$ iff the value r'_{w_2} reflects an index i s.t. $i = w_1$. Thus, after receiving r'_{w_2} sent in step 11, Sim_1^2 extracts the index i, by comparing $r'_{a=w_2}$ with the list $r_0, \ldots, r_{\lceil \log(e) \rceil - 1}$. First note that no two values r_i, $r_{i'}$ for $i \neq i'$ are equal e.w.n.p. in κ, so r'_{w_2} will match at most a single item. If there is a match then Sim_1^2 inputs this to $\mathcal{F}_{\mathsf{RSA}}$ as part of the *construct candidate* command to learn whether $i \neq w_1$. If this does *not* hold, then Sim_1^2 returns $\bot$. Otherwise it returns $\top$ exactly like what would happen in the real execution. Thus $\top$ is returned even if r'_{w_2} does not match any of the messages in $\mathcal{F}_{\mathsf{OT}}^{\kappa,\lceil \log(e) \rceil}$. However, this is exactly like it would happen in the execution with P_1. Notice that this candidate will never be used again in the simulation.

Consider the case of $\mathcal{I} = 2$:

First see that $r_{\alpha,i}$ is picked uniformly at random by Sim_2^2 exactly as in Sim_2^1. Next, notice that by the ideal functionality of $\mathcal{F}_{\mathsf{OT}}^{2\ell,2}$ we have that $\hat{P}_1$ can only learn a single value of the two messages in the OT. This means that whether $\hat{P}_1$ receives $r_{\alpha,i}$ in Sim_2^1 or $r_{\alpha,i} + \alpha_i \mod \mathcal{P}$ in Sim_2^2 (when it chooses message 1 in the OT) it cannot distinguish as they are both randomly distributed. Furthermore, we see that the only other value sent dependent on $r_{\alpha,i}$ is a_2. However, we see that a_2 is computed exactly like in Sim_2^1 since $a_1 = p_1 q_1 + z_1^p + z_1^q \mod \mathcal{P}$ and $a_2 = N - a_1 \mod \mathcal{P} \Rightarrow N = a_1 + a_2 \mod \mathcal{P}$.

Next see that $\hat{P}_1$ can choose whatever it wants for the values $h_{\alpha,1}, \ldots, h_{\alpha,2\ell+3s}$, g_α, d_α and in particular that it can pick whatever it wants as input to the OTs. Specifically, since d_α is only used once (as input to the OT) we define $\hat{P}_1$'s choice of these values from what the simulator extracts from the OTs.

In regards to step 8-12 we see that Sim_2^2 emulates $\mathcal{F}_{\mathsf{OT}}^{\kappa,\beta}$ by selecting a random value $r_{w'_1} \in \{0,1\}^\kappa$ and returning this to $\hat{P}_1$ and extracting its input message w'_1. This $r_{w'_1}$ is clearly distributed the same as in the real execution with P_1, since this is also randomly sampled. Sim_2^2 then inputs w'_1 to $\mathcal{F}_{\mathsf{RSA}}$ as part of the *construct candidate* command to learn whether $w'_1 = w_2$. If this holds, then Sim_2^2 sends $r_{w'_1}$ to $\hat{P}_1$. If $\hat{P}_1$ asks the simulator to discard the candidate (i.e. if it behaves honestly) then the candidate is discarded. Otherwise the candidate is passed on to the next phase. Notice that even if $\hat{P}_1$ is sending incorrect messages he

can at most make simulator discard a proper candidate, *not* not make it accept an incorrect candidate. This follows since the ideal functionality ensures that $\gcd(e, \phi(N)) = 1$ since $\phi(N) = (p-1)(q_1)$ and $\gcd(e, p-1) = \gcd(e, q-1) = 1$ and the primes are larger than e.

Trial Division. We see that $\mathsf{Sim}^3_{\mathcal{I}}$ and $\mathsf{Sim}^2_{\mathcal{I}}$ execute the trial division in the same way. Specifically Sim^3_1 waits for ν from $\hat{P}_2$ and discards the candidate iff $\nu = \perp$, otherwise it continues, like Sim^2_1. Sim^3_2 does trial division on N, in the same way in both the simulations.

Biprimality Test. Consider step 1:

First consider the case where $\mathcal{I} = 1$: Notice that $\hat{P}_2$ does not send any messages. Thus we only need to argue that the messages sent by Sim^3_1 are indistinguishable from the ones sent by Sim^2_1. This is quite easy since we notice that Sim^3_1 simulates in the same way as in the proof of Boneh and Franklin [BF01], with the exception of what they call "step 4", thus we refer the reader to their proof for arguments of indistinguishability of the messages sent. There is only one slight difference which is that $\hat{P}_2$ returns a bit ν. However, in this case we see that both Sim^2_1 and Sim^3_1 discard N iff $\nu = \perp$ and accepts its iff $\nu = \top$.

Next consider the case where $\mathcal{I} = 2$: Again we piggyback on the proof of Boneh and Franklin [BF01]. However, since $\hat{P}_1$ can supply possibly malicious input, here we must also argue that the simulators proceed similarly if $\hat{P}_1$ sends incorrect messages. In particular $\hat{P}_1$ could send incorrectly constructed values γ and γ_1 to the simulator. Now, we know that p and q are primes, meaning that if $\hat{P}_1$ sends correct values then the test must always pass. Now see that Sim^3_2 checks whether $\left(\frac{\gamma}{N}\right) = 1$ and aborts if not (exactly like the real Sim^2_2). Thus we can assume $\left(\frac{\gamma}{p}\right) = \left(\frac{\gamma}{q}\right)$. From which it follows that $\gamma^{\phi(N)/4} \equiv \pm 1 \mod N$ [BF01]. This means that

$$\begin{aligned}
\gamma^{\frac{(p-1)\cdot(q-1)}{4}} &\equiv \pm\gamma^{\frac{N+1-p-q}{4}} \mod N \\
&\equiv \pm\gamma^{\frac{N+1-p_1-p_2-q_1-q_2}{4}} \mod N \\
&\equiv \pm\gamma^{\frac{N+1-p_1-q_1}{4}} \mod N \\
&\equiv \pm\gamma^{\frac{-p_2-q_2}{4}} \mod N
\end{aligned}$$

Using this we make the following observation based on the value γ_1 that Sim^3_2 receives from $\hat{P}_1$:

$$\begin{aligned}
\gamma_1 \cdot \gamma^{\frac{N+1-p_1-q_1}{4}} &\equiv \pm\gamma_1 \cdot \gamma^{\frac{-p_2-q_2}{4}} \mod N \\
\gamma_1 \cdot \gamma^{\frac{N+1-p_1-q_1}{4}} &\equiv \pm 1 \mod N
\end{aligned}$$

This means that Sim^2_2 will return $\top$ to $\hat{P}_1$ iff $\pm\gamma_1 \cdot \gamma^{\frac{-p_2-q_2}{4}} \equiv \pm 1 \mod N$. Thus the simulation is sound.

Consider step 2:

First see that the AES encryptions sent are indistinguishable between Sim_2^2 and Sim_2^3 because of the IND-CPA assumption. Next notice that the simulator emulates the OT functionality when doing multiplication as in "Construct Modulus". See each 0-message contains a $3\ell + s + 2$ bit random value which is the same as in the real execution and that the 1-message contains $p'_{\mathcal{I}} + q'_{\mathcal{I}}$ (or $p'_{\mathcal{I}} + q'_{\mathcal{I}} - 1$ if $\mathcal{I} = 2$) added to the random value, modulo $3\ell + s + 2$. However, $p'_{\mathcal{I}} + q'_{\mathcal{I}}$ and $p_{\mathcal{I}} + q_{\mathcal{I}}$ are both picked at random (under the constraints given by leakage) and thus both messages input to the OT are indistinguishable between Sim_2^2 and Sim_2^3. We see that the since the leakage is constant, the simulator can in constant time find random values $p'_{\mathcal{I}}$, $q'_{\mathcal{I}}$ consistent with the leakage.

Because the OT messages are picked in the same way in the simulation this also means that the value $s_{\mathcal{I}}$ simulated is indistinguishable from the real execution. This is seen first from the fact that $3\ell + 4s + 2$ OT executions are done on messages of at most $3\ell + s + 2$ bits, following the same argument as in "Construct Modulus", i.e. Lemma C.6. Second, because $s_{\mathcal{I}}$ is computed under the same constraints and that the values dependent on $\tilde{p}_{\mathcal{I}}$ and $\tilde{q}_{\mathcal{I}}$ (which are simulated by $\tilde{p}'_{\mathcal{I}}$ and $\tilde{q}'_{\mathcal{I}}$ in Sim_2^3) will be statistically indistinguishable from the true values used in Sim_2^2 as $\bar{r}_{\mathcal{I}}$ will statistically hide them (argument below). Furthermore since $\bar{r}_{\mathcal{I}}$ is only used for computing $s_{\mathcal{I}}$ and picked uniformly at random in both Sim_2^2 and Sim_2^3 this will be indistinguishable.

To argue that $s_{\mathcal{I}}$ computed in Sim_2^2 is indistinguishable from the value of Sim_2^3 we first observe that $\bar{r}_{\mathcal{I}}$ is random in $\mathbb{Z}_N$ and so multiplying this with any value modulo N will yield something random. Thus $s_{\mathcal{I}}$ will also be random.

Finally we notice that the simulated check pass exactly in the cases as the real check since we already know (as we are running this simulation) that N is a biprime.

Proof of Honesty. First note that Boneh and Franklin [BF01] showed that whether a random value γ_i with Jacobi symbol 1 over N is used (like in $\mathsf{Sim}_{\mathcal{I}}^3$, or if it is simulated by squaring a random value and picking a random sign (like in $\mathsf{Sim}_{\mathcal{I}}^4$) is indistinguishable.

Proof of Honesty, simulating P. Next see that the values $\gamma_{i,P}$, whether computed like in the real protocol by $\mathsf{Sim}_{\mathcal{I}}^3$ or simulated in $\mathsf{Sim}_{\mathcal{I}}^4$ using the malicious party's shares, are indistinguishable since N is always a biprime when $\mathsf{Sim}_{\mathcal{I}}^4$ simulates following the same argument as above for Sim_2^3. Thus whether $\hat{P}_{3-\mathcal{I}}$ returns $\perp$ is independent of whether it executes with $\mathsf{Sim}_{\mathcal{I}}^3$ or $\mathsf{Sim}_{\mathcal{I}}^4$. We next see that when $b_j = 0$ the simulated values t_j, H_{t_j}, $\bar{\gamma}_{i,j}$ and v_j are computed in exactly the same way in both $\mathsf{Sim}_{\mathcal{I}}^3$ and $\mathsf{Sim}_{\mathcal{I}}^4$ since t_j is randomly sampled from $\{0,1\}^{\ell-2+s}$ and H_{t_j}, $\bar{\gamma}_{i,j}$, and v_j are computed based on this in the same way in both simulations. For the cases where $b_j = 1$ first see that the statistical difference between $-\tilde{p}_P - \tilde{q}_P + t_j$ and a uniformly random string of $\ell - 2 + s$ bits is at most 2^{-s}, since t_j is uniformly random and of $\ell - 2 + s$ bits. Let $t_{j,3}$ be the value picked in $\mathsf{Sim}_{\mathcal{I}}^3$ and consider $t_{j,4} = t_{j,3} - \tilde{p}_P - \tilde{q}_P$ as the value in $\mathsf{Sim}_{\mathcal{I}}^4$ instead of a uniformly random sampled value t_j. These are clearly 2^{-s}

statistically indistinguishable. Next consider all the values where $t_{j,4}$ is used:

$$\begin{aligned}\bar{\gamma}_{i,j} &\equiv \gamma_i^{t_{j,4}} \cdot (-1)^{b_i} \cdot \gamma_i^{\frac{N-5}{4}-\tilde{p}_V-\tilde{q}_V} \mod N \\ &\equiv \gamma_i^{t_{j,3}-\tilde{p}_P-\tilde{q}_P} \cdot (-1)^{b_i} \cdot \gamma_i^{\frac{N-5}{4}-\tilde{p}_V+\tilde{q}_V} \mod N \\ &\equiv \gamma_i^{t_{j,3}} \cdot (-1)^{2b_i} \mod N \equiv \gamma_i^{t_{j,3}} \mod N\end{aligned}$$

Thus we see that the values sent in $\mathsf{Sim}_{\mathcal{I}}^4$ are indistinguishable from the ones sent in $\mathsf{Sim}_{\mathcal{I}}^3$.

Regarding H_{t_j} see that if the adversary can distinguish between the simulations this means that it knows $\tilde{p}_P + \tilde{q}_P$, since this is used to verify whether the digest is computed correctly from the value v_j. However, by Lemma C.1 we have that $(\tilde{p}_P, \tilde{q}_P)$ still has at least s bits of entropy. This in turn means that $\tilde{p}_P + \tilde{q}_P$ also has at least s bits of entropy and thus that the adversary cannot find this value in polytime with non-negligible probability.

Next see that if V acts honestly it will accept the simulated values v_j of $\mathsf{Sim}_{\mathcal{I}}^4$ by the following equation based on the γ_i checks:

$$\begin{aligned}\gamma_i^{v_j} &\equiv \bar{\gamma}_{i,j} \cdot \gamma_{i,P}^{b_j} \cdot \gamma_i^{-b_j \cdot \frac{N-5}{4}} \mod N \\ \gamma_i^{t_j} &\equiv (-1)^{b_i} \cdot \gamma_i^{t_j + b_j \cdot \left(\frac{N-5}{4} - \tilde{p}_V - \tilde{q}_V\right)} \cdot (-1)^{b_i} \cdot \gamma_i^{\tilde{p}_V + \tilde{q}_V} \cdot \gamma_i^{-b_j \cdot \frac{N-5}{4}} \mod N \\ \gamma_i^{t_j} &\equiv \gamma_i^{t_j + b_j \cdot \left(\frac{N-5}{4} - \tilde{p}_V - \tilde{q}_V - \frac{N-5}{4} + \tilde{p}_V + \tilde{q}_V\right)} \cdot (-1)^{2b_i} \mod N \\ \gamma_i^{t_j} &\equiv \gamma_i^{t_j} \mod N\end{aligned}$$

Finally $\mathsf{Sim}_{\mathcal{I}}^4$ emulates $\mathcal{F}_{\mathsf{2PC}}$ and aborts in exactly the same case as in $\mathsf{Sim}_{\mathcal{I}}^3$ since it uses the true shares given as input of both the malicious party and the true honest party.

Following the same argument as in the biprimality testing, we see that $\mathsf{Sim}_{\mathcal{I}}^4$ accept the values sent by $\hat{P}_{3-i}$ in step 2 iff $\mathsf{Sim}_{\mathcal{I}}^3$ would accept these.

Next we see that $\mathsf{Sim}_{\mathcal{I}}^4$ does not sample anything and performs the same checks in the same way as in $\mathsf{Sim}_{\mathcal{I}}^3$, the only exception being the value δ. However, since $\hat{r}_P \in \{0,1\}^{3\ell+2s+2}$ and the other term of δ, $\hat{r}_V \cdot ((\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_1 + \tilde{p}_2 + \tilde{q}_1 + \tilde{q}_2) + 5) - \sigma) \in \{0,1\}^{3\ell+s+2}$ we see that the simulated δ is 2^{-s} statistically indistinguishable. Finally $\mathsf{Sim}_{\mathcal{I}}^4$ emulates $\mathcal{F}_{\mathsf{2PC}}$ and aborts in exactly the same case as in $\mathsf{Sim}_{\mathcal{I}}^3$ since it uses the true shares input of both the true malicious party and the true honest party.

Proof of Honesty, simulating V, $\mathsf{Sim}_{\mathcal{I}}^5$. We need to argue that whether the simulation of $\mathsf{Sim}_{\mathcal{I}}^5$ is used to compute the output of $\mathcal{F}_{\mathsf{2PC}}$ or whether it is computed using the true input $\tilde{p}_{\mathcal{I}}, \tilde{q}_{\mathcal{I}}, \rho_{\mathcal{I}}, K_{\mathcal{I}}$, like in $\mathsf{Sim}_{\mathcal{I}}^4$, the output will be the same. We know, from the sheer fact that we are executing $\mathsf{Sim}_{\mathcal{I}}^5$ that the true N is biprime and not marked as bad. However it might be the case that $N \neq (p_1+p_2)(q_1+q_2)$ if the adversary cheated in *Candidate Generation*. However, in that case the simulator sets $\chi = 0$ since it knows if this is the case from the simulation transcript

and the value N it got from $\mathcal{F}_{\mathsf{RSA}}$. This is also what would happen in the real execution since the $\mathcal{F}_{\mathsf{2PC}}$ functionality verifies this directly. Thus we can assume that the N in use is correctly computed.

First see that the simulation $\mathsf{Sim}_{\mathcal{I}}^5$ verifies that $\bar{r}'_P \cdot (4\tilde{p}'_P + 4\tilde{q}'_P + 6) + \alpha_P + \beta_P \mod N = \sigma - s_V \mod N$, where $\bar{r}'_P = \bar{r}_P$, $\tilde{p}'_P = \tilde{p}_P$ and $\tilde{q}'_P = \tilde{q}_P$ as these are based on the extracted values; as in $\mathsf{Sim}_{\mathcal{I}}^4$. Thus what gets checked in $\mathsf{Sim}_{\mathcal{I}}^4$ is also checked in $\mathsf{Sim}_{\mathcal{I}}^5$ with the exception that s_V and thus $\bar{r}_V$ and $\tilde{p}'_V, \tilde{q}'_V$ are simulated in $\mathsf{Sim}_{\mathcal{I}}^5$. Next we see that $\mathsf{Sim}_{\mathcal{I}}^5$ accepts iff $\delta' = \delta \mod N$ whereas $\mathsf{Sim}_{\mathcal{I}}^4$ might accept a $\delta' \neq \delta \mod N$. However for the adversary to find such a δ' it must guess (as a minimum) guess the 2^s bit value $\hat{r}_V$. To see this notice that to construct a $\delta' \neq \delta \mod N$ that still gets accepted it must be the case that $\delta' = \hat{r}_P - \epsilon$ where $0 \neq \epsilon = \hat{r}_V \left((\bar{r}_P + \bar{r}_V) \cdot (4(\tilde{p}_P + \tilde{p}_V + \tilde{q}_P + \tilde{q}_V) + 5) - \sigma\right) \mod N$. However, since $\bar{r}_V \in \{0,1\}^s$ there are 2^s possible values and since $\hat{r}_V$ is random, and completely hidden to the adversary by the $\mathcal{F}_{\mathsf{2PC}}$ in both simulations it cannot guess such a value with probability greater than 2^{-s}.

Next see that $N - 5 - 4(\tilde{p}_P + \tilde{q}_P + \tilde{p}_V + \tilde{q}_V) \mod e \neq 0$ (i.e. that $\gcd(\phi(N), e) = 1$) when $H_{\tilde{p}_P} = \mathsf{AES}_{K_P}(\tilde{p}_P)$, $H_{\tilde{q}_P} = \mathsf{AES}_{K_P}(\tilde{q}_P)$, $H_{\bar{r}_P} = \mathsf{AES}_{K_P}(\bar{r}_P)$ and $c_p = \mathsf{AES}_{\mathsf{AES}_{r_P}(K_P)}(0)$. This means that if $\hat{P}_P$ inputs the same values to $\mathcal{F}_{\mathsf{2PC}}$ as were extracted in the beginning of the protocol then $\mathcal{F}_{\mathsf{2PC}}$ should output $\chi = 1$, otherwise $\chi = 0$. When using $\mathsf{Sim}_{\mathcal{I}}^4$, and thus executing like in the real protocol (*but* with the same assumptions as above), we see then $\mathcal{F}_{\mathsf{2PC}}$ will output $\chi = 1$ by correctness of the protocol. However, if they are not then it will output $\chi = 0$. This follows since we know that $H_{\tilde{p}_P} \neq \mathsf{AES}_{K'_P}(\tilde{p}'_P)$, $H_{\tilde{q}_P} \neq \mathsf{AES}_{K'_P}(\tilde{q}'_P)$,$H_{\bar{r}_P} \neq \mathsf{AES}_{K'_P}(\bar{r}_P)$ or $c_p \neq \mathsf{AES}_{\mathsf{AES}_{r_P}(K'_P)}(0)$ if $K'_P \neq K_P$, $\tilde{p}'_P \neq \tilde{p}_P$, or $\tilde{q}'_P \neq \tilde{q}_P$ following the argument that AES is a PRP. In regards to v_j and H_{t_j} we notice that $\mathsf{Sim}_{\mathcal{I}}^5$ does exactly the same check as in $\mathsf{Sim}_{\mathcal{I}}^4$.

Proof of Honesty, simulating P, $\mathsf{Sim}_{\mathcal{I}}^5$. In the case where the simulator simulates P we first see, using the same argument above, that $\mathsf{Sim}_{\mathcal{I}}^5$ will output $\chi = 0$ if the malicious verifier gives input $\tilde{p}'_V \neq \tilde{p}_V$, $\tilde{q}'_V \neq \tilde{q}_V$, $K'_V \neq K_V$, $\bar{r}'_V \neq \bar{r}_V$ or N is not the one received from $\mathcal{F}_{\mathsf{RSA}}$.

Regarding the output $\bar{d}_V$ when $\chi = 1$ see that the simulator gets the true output share d_V from $\mathcal{F}_{\mathsf{RSA}}$ as part of the *construct* command. Since the malicious party adds ρ_V to the $\bar{d}_V$ to get d_V the simulator must subtract ρ_V to ensure that it returns a value consistent with what will be the output of the computation.

Furthermore we must argue that the shares d_1, d_2 output by $\mathsf{Sim}_{\mathcal{I}}^4$ are distributed the same as what is output by $\mathsf{Sim}_{\mathcal{I}}^5$. To see this first notice that $d_V \in [-2^{2\ell}, 2^{\ell}[$ since $N \in \mathbb{Z}_{2^{2\ell}}$, thus when $V = P_1$ $d_V \in [-2^{2\ell}, 0]$ and when $V = P_2$ $d_V \in [2^{\ell}]$. Since $\rho_P \in [2^{2\ell+s}$ we see that adding d_V to r_P results in a number in $[2^{2\ell+s}]$ e.w.p. 2^{-s}. Furthermore, this number will be 2^{-s} from a random number in $[2^{2\ell+s}]$. Thus, the value $d_{3-\mathcal{I}}$ output by the adversary (sent by $\mathsf{Sim}_{\mathcal{I}}^5$ from $\mathcal{F}_{\mathsf{2PC}}$) will be 2^{-s} indistinguishable from the value output by the adversary (sent by $\mathsf{Sim}_{\mathcal{I}}^4$ from $\mathcal{F}_{\mathsf{2PC}}$) since $\mathcal{F}_{\mathsf{RSA}}$ gives $\mathsf{Sim}_{\mathcal{I}}^5$ a value $d_{3-\mathcal{I}}$ randomly sampled from the same range as what is returned in the real execution.

However, when $V = \hat{P}_2$ there is a slight problem. Specifically, depending on the private shares of the parties, the output of $\mathcal{F}_{\mathsf{2PC}}$ might be off-by-one and need to be adjusted in *Generate shared key*. The simulated output of $\mathcal{F}_{\mathsf{2PC}}$ is adjusted with an off-by-one error iff $e| - 4\tilde{p}_2 - 4\tilde{q}_2$. Still, this is also what is done in the real execution. Thus we see that the value from $\mathcal{F}_{\mathsf{RSA}}$ will be subtracted by 1 exactly if there would be an off-by-one in the real execution.

Generate Shared Key. Both simulators input *Select* to $\mathcal{F}_{\mathsf{RSA}}$ to accept the candidate. It is clear that the candidate exists and is acceptable, since both simulators only execute on N, constructed from two primes. However, we must still argue that too much information on the honest party's shares $\tilde{p}_{\mathcal{I}}, \tilde{q}_{\mathcal{I}}$ have been leaked. However, this follows directly from C.1 since no more calls have been made to *Leak* after the *Construct Modulus* phase.

$\mathsf{Sim}^5_{\mathcal{I}}$ *to* $\mathsf{Sim}_{\mathcal{I}}$. Finally we argue that we can go from $\mathsf{Sim}^5_{\mathcal{I}}$ to $\mathsf{Sim}_{\mathcal{I}}$. We first argue that the candidates used to construct N will be generated by $\mathcal{F}_{\mathsf{RSA}}$ that is, if N is marked as *bad* or not a biprime *and* $\mathcal{F}_{\mathsf{2PC}}$ has output $\chi = 1$. In particular the latter can only occur by malicious behavior. We now see exactly what can make this occur. First assume the adversary has the role of P in *Proof of Honesty*. We can assume that the malicious P inputs $\tilde{p}_P$ and $\tilde{q}_P$ that were extracted, otherwise $\mathcal{F}_{\mathsf{2PC}}$ will output 0. With that in mind we see that if N is marked as bad then we know that $N \neq (p_1 + p_2)(q_1 + q_2)$, or $N + 1 - p_1 - p_2 - q_1 - q_2 \mod e = 0$, even if $p_1 + p_2$ and $q_1 + q_2$ were primes and thus $\mathcal{F}_{\mathsf{2PC}}$ will output 0. From this we see that $\hat{P}_{3-\mathcal{I}}$ must use values $\tilde{p}'_{3-\mathcal{I}}$ and $\tilde{q}_{3-\mathcal{I}}$ in the *Proof of Honesty* phase, different from the values it encrypted in *Generate Candidate*, otherwise the protocol would have aborted (by correctness of the biprimality test). Next see, that we can also assume that $v_j = b_j(-\tilde{p}_P - \tilde{q}_P) + t_j$, otherwise $\mathcal{F}_{\mathsf{2PC}}$ will always output 0. This follows since P is committed to its choices t_j by the AES encryption in step 5. In particular this means that each t_j cannot depend on any values he learned after step 5. Thus P must send at least one value $\gamma_{i,P} \neq \gamma_i^{\frac{N-5}{4} - \tilde{p}_P - \tilde{q}_P}$ for $i \in [s]$ s.t. $\gamma_i^{-\tilde{p}_V - \tilde{q}_V} \cdot \gamma_{i,P} \equiv \pm 1 \mod N$. But since we have not already aborted this means that

$$\begin{aligned}
\bar{\gamma}_{i,j} \cdot \gamma_i^{b_j} \cdot \gamma_i^{-b_j \cdot \frac{N-5}{4}} &\equiv \gamma_i^{v_j} \mod N \\
\bar{\gamma}_{i,j} \cdot \gamma_i^{b_j} \cdot \gamma_i^{-b_j \cdot \frac{N-5}{4}} &\equiv \gamma_i^{b_j \cdot (-\tilde{p}_P - \tilde{q}_P) + t_j} \mod N
\end{aligned}$$

If $b_j = 0$ it means that

$$\bar{\gamma}_{i,j} \equiv \gamma_i^{t_j} \mod N$$

Similarly, if $b_j = 1$ it means that

$$\begin{aligned}
\bar{\gamma}_{i,j} \cdot \gamma_{i,P} \cdot \gamma_i^{-\frac{N-5}{4}} &\equiv \gamma_i^{-\tilde{p}_P - \tilde{q}_P + t_j} \mod N \\
\bar{\gamma}_{i,j} \equiv \gamma_{i,P}^{-1} \cdot \gamma_i^{-\tilde{p}_P - \tilde{q}_P} \cdot \gamma_i^{t_j} \cdot \gamma_i^{\frac{N-5}{4}} &\equiv \gamma_{i,P}^{-1} \cdot \gamma_i^{-\tilde{p}_P - \tilde{q}_P + t_j + \frac{N-5}{4}} \mod N
\end{aligned}$$

This means that for the adversary to be able to answer both queries of b_j it must hold that

$$\begin{aligned}\gamma_i^{t_j} &\equiv \gamma_{i,P}^{-1} \cdot \gamma_i^{-\tilde{p}_P - \tilde{q}_P + t_j + \frac{N-5}{4}} \mod N \\ 1 &\equiv \gamma_{i,P}^{-1} \cdot \gamma_i^{-\tilde{p}_P - \tilde{q}_P + \frac{N-5}{4}} \mod N \\ \gamma_{i,P} &\equiv \gamma_i^{\frac{N-5}{4} - \tilde{p}_P - \tilde{q}_P} \mod N\end{aligned}$$

However, this is the value the honest party should have sent. Thus he must send the correct value $\gamma_{i,P}$ for each $i \in [s]$ to be able to pass both choices of b_j for each $j \in [s]$. That is, for each $\gamma_{i,P} \neq \gamma_i^{\frac{N-5}{4} - \tilde{p}_P - \tilde{q}_P} \mod N$ the adversary can make the check pass for at most one value of b_j for each $j \in [s]$. That is, with probability at most 2^{-s}.

The last thing to argue is that whether the values $\tilde{p}_{\mathcal{I}}$ for the candidate values that are discarded are sampled at random by $\mathsf{Sim}_{\mathcal{I}}$ or as in the true protocol like in $\mathsf{Sim}_{\mathcal{I}}^5$ is indistinguishable. This follows since these values are independent from the N given as result and are sampled from the same, random, distribution in both the real and simulated execution.