Improved Distributed RSA Key Generation Using the Miller-Rabin Test

Jakob Burkhardt[1], Ivan Damgård[1], Tore Kasper Frederiksen[2]*, Satrajit Ghosh[3], and Claudio Orlandi[1]

[1] Department of Computer Science, Aarhus University, Denmark
[2] Zama France,
[3] Indian Institute of Technology, Kharagpur, India

jakob@cs.au.dk, ivan@cs.au.dk, tore.frederiksen@zama.ai, satrajit@cse.iitkgp.ac.in, orlandi@cs.au.dk

Abstract. Secure distributed generation of RSA moduli (e.g., generating $N = pq$ where none of the parties learns anything about p or q) is an important cryptographic task, that is needed both in threshold implementations of RSA-based cryptosystems and in other, advanced cryptographic protocols that assume that all the parties have access to a trusted RSA modulo. In this paper, we provide a novel protocol for secure distributed RSA key generation based on the Miller-Rabin test. Compared with the more commonly used Boneh-Franklin test (which requires many iterations), the Miller-Rabin test has the advantage of providing negligible error after even a single iteration of the test for large enough moduli (e.g., 4096 bits).

From a technical point of view, our main contribution is a novel divisibility test which allows to perform the primality test in an efficient way, while keeping p and q secret.

Our semi-honest RSA generation protocol uses any underlying secure multiplication protocol in a black-box way, and our protocol can therefore be instantiated in both the honest or dishonest majority setting based on the chosen multiplication protocol. Our semi-honest protocol can be upgraded to protect against active adversaries at low cost using existing compilers. Finally, we provide an experimental evaluation showing that for the honest majority case, our protocol is much faster than Boneh-Franklin.

* Majority of this work was done while at the Alexandra Institute, Aarhus, Denmark and some while at Protocol Labs.

Table of Contents

1 Introduction

The RSA cryptosystem [RSA78] is one of the earliest public-key cryptosystems and remains in use today. In classical applications of RSA, each user stores their secret key and uses it to decrypt or sign, as the case may be. However, this creates a single point of attack, which is often undesirable. The problem can be solved by using a distributed or threshold version, where a number of devices each have a share of the secret key and can therefore collaborate to decrypt or sign. This can be set up such that breaking into any minority of the servers will neither compromise the secret key, nor compromise the availability of the system.

However, to really avoid a single point of attack, one needs to perform also the key generation phase in a distributed fashion so that the secret key can never be stolen from a single location. Moreover, many modern cryptographic protocols and primitives (e.g., time-lock puzzles [RSW96], accumulators [Bd94,CL02], VDFs [BBBF18], etc.) assume that all parties have access to some RSA modulus whose factorization is known to none of the parties.

Thus, the problem we consider in this paper is: we are given n parties, of which any $t < n$ can be corrupted by an adversary, and we want a secure protocol that outputs a random and valid RSA modulus $N = pq$ where p, q are primes of a given size, while the adversary learns nothing but N from the protocol. The standard approach to RSA key generation is to select random candidates for p and q and test for primality. This is non-trivial in the distributed setting, as we can of course not afford to make p or q public. Therefore generating the modulus is usually a bottleneck in distributed RSA systems. There is a rich literature on distributed RSA key generation, see the related work section below for details. However, with two exceptions we return to below, all previous works on this make use of the Boneh-Franklin bi-primality test [BF97].

The idea in most such protocols is that the parties create shares of two random numbers p, q of the desired size. They then compute securely the product $N = pq$ and make it public. Now, one can very efficiently run the Boneh-Franklin test by exploiting that N is public and the candidate numbers p, q are secret shared among the parties. Without going into details, this test has the property that it always accepts when p and q are prime, and otherwise accepts with probability at most $1/2$. While this is a worst case bound, no better bound is known for the average case behavior, so in order to make sure the probability of outputting an invalid N is negligible, one needs to repeat the test security parameter many times.

The first paper in the literature that uses a different approach is by Algesheimer et al. [ACS02]. They use generic MPC methods to generate the prime factors by testing random candidate numbers individually for primality using the well-known Miller-Rabin test [Mil76,Rab80]. Since this test requires an exponentiation modulo the number to test, one needs an MPC protocol for exponentiation where the base, the exponent and the modulus must all remain secret. This protocol has the advantage that it can generate each of the two primes independently of the other, in contrast to the approach followed by all other protocols, where one generates pairs of candidates and must wait until both are prime. However, it is still a very inefficient protocol by today's standards. In particular, it inherently requires many rounds because each secure multiplication costs at least one round, and the multiplications done during the exponentiation cannot be parallelized, as far as we know.

A subsequent paper, by Damgård and Mikkelsen [DM10], also uses the Miller-Rabin test but obtains a much more efficient protocol. Their idea is also to generate p and q, compute $N = pq$, make N public and finally test p and q individually using Miller-Rabin. In a moment we will get back to how this is done without revealing p and q, but first we note that very good bounds on the average case behavior of the Miller-Rabin test are known [DLP93]. Hence, while the test may accept a composite with probability $1/4$ in the worst case, the error probability is much smaller for a random input. So much so, in fact, that when numbers get large enough, we only need to do the test once (see Table 1) to ensure a negligible probability of outputting a composite, assuming we choose random candidates and output the first one that passes the test [DM10,DLP93].

In [DM10], this result is used as follows: they make sure that $p = q = 3 \bmod 4$, as this simplifies the Miller-Rabin test, and it proceeds as follows (for p and similarly for q): choose random $\gamma \in \mathbb{Z}_p^*$ and accept if $\gamma^{(p-1)/2} = \pm 1 \bmod p$. The high-level idea is then to choose a random $\gamma \in \mathbb{Z}_N^*$ and compute (in secret shared form) $\gamma^{(p-1)/2} - 1 \bmod N$ (and $\gamma^{(p-1)/2} + 1 \bmod N$). This can be done efficiently and in constant-round by exploiting the fact that p is already secret shared and N is public. The remaining problem is that to complete the test, we need to check whether p divides one of these two numbers. This problem was solved in [DM10], and they obtained two protocols for 3 players, secure against 1 semi-honest, respectively 1 malicious corruption. However, there were two drawbacks: First, for malicious security, a set-up assumption is required to establish a commitment scheme for integers, and players are required to commit to every share of every secret generated in the protocol, which requires large-scale exponentiations each time. Second, the test for divisibility uses secure computation modulo a very large prime, whose bit-length must be $\Omega(nk)$ where n is the number of parties and k is the bit-length of N. While this was not so serious in the context of [DM10], as they only handled the case of $n = 3$ explicitly, it raises the following question:

Can we use the Miller-Rabin test for efficient distributed RSA key generation, with more than 3 parties and without set-up assumptions?

1.1 Our Contributions

Semi-honest security. In this paper, we address the open problem described above. We first propose a protocol for distributed generation of RSA moduli that is secure against semi-honest corruption of t of the n parties, for any $t < n$. The protocol is statistically secure, assuming access to a functionality for secure multiplication. The functionality can be implemented with unconditional security by standard Shamir-sharing based methods if $t < n/2$ (or replicated secret-sharing for small n), and otherwise requires a computationally secure approach. Like the 3-party protocol from [DM10], ours is based on the Miller-Rabin test and only requires one iteration of the test for large enough target modulus sizes. We follow the blueprint from [DM10] but propose a new divisibility test which, like the one from [DM10], requires secure computation modulo an auxiliary prime, but we only require it to have bit length $O(k + \log n)$, rather than $\Omega(nk)$. Hence our protocol, instantiated for $n = 3, t = 1$, compares favorably to the semi-honest protocol from [DM10].

We also compare favorably to semi-honest secure protocols using the Boneh-Franklin test, such as [BF97]: the computational complexity of both protocols is dominated by the required exponentiations modulo N. Boneh-Franklin needs s of these per player where s is the security parameter, while we need only 2 (one for each prime factor). As for the communication complexity of Boneh-Franklin, this is also dominated by the $\Omega(s)$ iterations of the test, as each player needs to send a k-bit number to the others per iteration. On the other hand, we need $O(n)$ secure multiplications modulo N. For $t < n/2$, a secure multiplication using standard methods requires each player to send a k-bit number to the others, so as long as $n << s$, our communication complexity is smaller. As one would typically want $s \approx 60 - 100$, n will be much smaller than s in many practical settings. For $t \geq n/2$, the picture is less clear since our protocol become dominated by the more expensive method used for secure multiplication in this setting. Finally, both Boneh-Franklin and our protocol can be done in a constant number of rounds. In particular, twice in our protocol we need to multiply together $O(n)$ secret values, and this can be done in constant round using an idea by Bar-Ilan and Beaver [BIB89][4].

Malicious security. There are several ways to upgrade from semi-honest to malicious security. A very efficient approach is to use the compiler from [DOS18], which was implemented and optimized in [EKO+19]. It takes a semi-honest protocol and produces an actively secure one tolerating a smaller number of corruptions. The compiler does not use zero-knowledge proofs, but is based on several players doing the same computation so that they can verify each others' actions. It is therefore very efficient, but is most interesting for a small number n of players, asymptotically it only tolerates $\Theta(\sqrt{n})$ active corruptions. Using this approach, we can obtain a maliciously secure protocol for $n = 3, t = 1$, the same setting as [DM10], but without any set-up assumptions and no need for a computationally expensive commitment scheme. This gives security with abort, but we can also get a protocol for $n = 4, t = 1$ with guaranteed output delivery. In both cases the overhead over the semi-honest version is a small constant factor. More details in Section 6.1.

Another approach to getting malicious security is to use zero-knowledge proofs. This has been done many times in constructions based on the Boneh-Franklin test (see the related work section for examples).

[4] for a small number of parties, it is faster to do the multiplications in a standard tree structure. See Section 7 for details about the best cut-off point between the two techniques.

Usually, the idea is to optimistically run the tests with a semi-honest protocol and then prove afterwards that everything was done correctly. This allows tolerating more corruptions but on the other hand degrades performance, and the zero-knowledge step is usually the bottleneck in earlier work: if compact proofs such as SNARKS are used, proofs are short, but a heavy computational burden is placed on the prover. Alternatively, more "classical" zero-knowledge protocols can be used that are more efficient for the prover, but where the communication complexity will increase with s, the number of iterations of the test. The same approach can be used with our protocol, but with less performance degradation, because the local computation that must be proved correct is much smaller in our case, as explained above. We leave working out the details of this approach for future work.

Experimental Evaluation. We implemented our semi-honest protocol as well as a semi-honest version of the protocol using the Boneh-Franklin test, for 2, 3, 5, 7, 9 and 11 players[5]. Details on the implementation and results are found in Section 7. The results clearly show that our protocol is much faster than Boneh-Franklin in case of honest majority, by a factor 3.2-14 in latency and 42-77 in throughput, while we are slower for dishonest majority. This confirms what one might expect from the fact that we need several secure multiplications, which are cheap for honest majority, but expensive for dishonest majority. While in contrast, Boneh-Franklin is slowed down by the computational load of the many exponentiations, but does not need secure multiplications. We also benchmark the extra steps needed for the maliciously secure version for $n = 3, t = 1$ based on the compiler from [DOS18]. We found that this was slower than the semi-honest version by a factor between 1.8x and 4x depending on the metric [6].

1.2 Technical Overview

We follow the blueprint of the protocol in [DM10], i.e., we generate a random candidate modulus $N = pq$, where we individually test p and q for primality using Miller-Rabin. As described in the introduction, the problem to solve is to do a secure and efficient test to decide if one secret-shared number divides another. More precisely, the setting is as follows: we have a public number N and an (additive) secret sharing of a divisor f in N. We use $[f]_{\mathbb{Z}}$ as shorthand for shares in f held by the parties, where the integer sum of the shares is f. We also have an additive sharing modulo N, $[\delta]_N$, where $\delta \in \mathbb{Z}_N^*$ and the subscript N means that the shares add to δ modulo N. The goal is to decide whether f divides δ.

The basic idea is to choose a public prime $P > N$ and exploit the fact that if indeed f divides δ, then $\delta \cdot f^{-1} \bmod P = \delta/f$. It is relatively easy to obtain $[\delta]_P$ and $[f]_P$ from what we are given (there are some caveats we gloss over here for simplicity). And then we can compute $[\delta \cdot f^{-1} \bmod P]_P$ using standard methods. We then do exactly the same for another large prime Q to obtain $[\delta \cdot f^{-1} \bmod Q]_Q$. The point is now that if f divides δ, then the two numbers shared modulo P and Q are the same, namely both are equal to δ/f. But if not, we can show that if Q is random and large enough, the two numbers are equal with only negligible probability. So we now need an equality test between values shared in different domains. While this is not a standard MPC primitive, we show a relatively easy solution to solve this problem. The resources needed for this is simple linear computations on shares and $O(n)$ secure multiplications where n is the number of parties.

2 Related Work

In [BF97] Boneh and Franklin presented the first efficient distributed RSA key generation protocol in a semi-honest setting. They proposed an algorithm to check whether a number is the product of two primes or

[5] We implemented only the actual bi-primality test, and not the part of a full RSA modulus generation where bad candidates are rejected. This would in practice be done the same way no matter how the final output is confirmed, namely by test division by small primes (or sieving) and by a single execution of Boneh-Franklin, which heuristically will reject almost all bad candidates, though a proof of this is not known, as mentioned.

[6] One could, of course, also compile the Boneh-Franklin protocol but this would lead to similar overhead factor, and so a comparison would still come out in favor of our protocol.

not, without knowing the factors, and used it to build a protocol for generating RSA keys securely assuming an honest majority. Most of the works in literature use the Boneh-Franklin bi-primality test along with other cryptographic primitives to design protocols for distributed RSA key generation. [ACS02] and [DM10] are notable exceptions, as discussed in Section 1.

In [FMY98] Frankel, MacKenzie and Yung modified the [BF97] protocol to propose a maliciously secure protocol in the honest majority settings. Later in [PS98] Poupard and Stern presented a maliciously secure two-party protocol using oblivious transfer. Unfortunately, their protocol leaks some information about the primes to the adversary. The first fully secure and concretely efficient protocol for the two-party settings appeared in [HMRT12]. In [Gil99] Gilboa improves the efficiency of the semi-honest secure two-party protocol by introducing efficient techniques for computing the modulus from additive shares. Specifically, [Gil99] proposed three different techniques based on homomorphic encryption, oblivious polynomial evaluation, and oblivious transfer respectively. The maliciously secure protocol by Hazay *et al.* [HMRT12] uses the homomorphic encryption-based technique from [Gil99] and zero-knowledge proof to ensure semi-honest behavior. In the two-party setting Frederiksen *et al.* [FLOP18] propose a semi-honest secure protocol and a maliciously secure protocol in OT hybrid model. Their maliciously secure protocol outperforms [HMRT12] by an order of magnitude, but allows slight leakage. In recent work, Chen *et al.* [CDK$^+$22] took a protocol-modular approach and proposed an actively secure protocol for more than two parties that uses a combined sampling-and-sieving technique, based on the Chinese Remainder Theorem (CRT), both in order to eliminate the leakage of [FLOP18], but also to increase the overall efficiency. The idea of the CRT-based sampling was originally discussed by Malkin *et al.* [MWB99] and consists of sampling secret shared non-zero numbers, modulo small distinct primes (CRT-components). Using these the CRT these are combined into a large *prime-candidate*. The approach has the advantage of ensuring that the prime candidate does not have any small divisors, and hence has a greater chance of being a prime than if it was randomly sampled. In a follow-up work, Chen *et al.* [CHI$^+$21] proposed a construction for maliciously secure distributed RSA key generation involving multiple parties. Similar to [CDK$^+$22], [CHI$^+$21] uses a Chinese Remainder Theorem (CRT)-based prime sampling technique. Though unlike [CDK$^+$22], they adopt a monolithic construction in order to facilitate different optimizations. Other than that they also assume a specific communication model with a semi-honest aggregator and many weak malicious clients. All these factors help them to enhance the concrete performance of their protocol. Consequently, they presented an efficient instantiation of distributed RSA key generation which can scale to more than 1000 parties. Unfortunately, these optimizations lead to different attacks when this protocol is used in real-world systems, as demonstrated in [Shl20a,Shl20b].Later, Guilhem *et al.* [DMRT21] introduced another optimization to the sampling of secret shared prime-candidates. Similar to Chen *et al.* [CDK$^+$22,CHI$^+$21], they use a CRT-based approach. However, unlike previous protocols, Guilhem *et al.*samples each CRT component using multiplicative secret-sharing. They observe that converting these multiplicative shares to additive shares, using a semi-honest protocol, can still result in a protocol for distributed RSA key generation which is *maliciously* secure and more efficient than doing non-zero random sampling based on additive shares.

We take a modular approach like [CDK$^+$22], which leads to a simple protocol description and helps us optimize individual modules in order to enhance the overall performance. We would like to emphasize that unlike most of the recent works [CDK$^+$22,CHI$^+$21] in the literature, we optimize distributed Miller-Rabin test and use that instead of the Boneh-Franklin bi-primality test. That along with a new divisibility test and other optimizations help us to design an efficient protocol for distributed RSA key generation. We refer the reader to Section 1.1 for a brief overview of our contributions.

Protocols for secure distributed generation of an RSA modulus have numerous applications, both in theory and practice, and they are used as building blocks in many complex protocols. For instance, RSA-based accumulators and verifiable delay functions (VDF) have already been implemented and deployed in Blockchain settings [BBF19,Wes19] and require an RSA group of unknown order (which can be generated using a secure distributed setup). Furthermore, secure distributed generation of RSA moduli finds usage by companies offering distributed key management, that offer the replacement of Hardware Security Modules (HSM) with interactive protocols. Apart from that, RSA is still widely employed "in the wild" by legacy applications, for instance in the banking world.

3 Preliminaries

3.1 Notation

We work with n-parties denoted by $\mathcal{P} = \{P_1, \ldots, P_n\}$, and $[n]$ denotes the set of integers $\{1, \ldots n\}$. Like most previous work in this area we consider the case of static corruptions and synchronous networks.

For secret sharings, we use the following notation and conventions: $[\alpha]_\beta$ will denote the secure additive sharing of value α in the integer domain $\mathbb{Z}_\beta$. It thus means that for all $i \in [n]$, party P_i is in possession of $\alpha^{(i)} \in \mathbb{Z}_\beta$, and $\sum_{i\in[n]} \alpha^{(i)} \mod \beta = \alpha$. Writing $[\alpha]_\mathbb{Z}$, defines an additive secret sharing in the integers. Similarly, $\langle\alpha\rangle_\beta$ denotes the multiplicative sharing of α in $\mathbb{Z}_\beta$. So party P_i with $i \in [n]$, has $\alpha_i \in \mathbb{Z}_\beta$ and $\prod_{i\in[n]} \alpha_i \mod \beta = \alpha$.[7]

For $\alpha, \gamma, c \in \mathbb{Z}_\beta$ whe write $c \cdot [\alpha]_\beta + [\gamma]_\beta$ to mean that each party $P_i \in \mathcal{P}$ locally computes $c\alpha^{(i)} + \gamma^{(i)} \mod \beta$. This then implies an additive sharing $[c\alpha + \gamma]_\beta$. Writing $[\alpha]_\beta + c$ means that party P_1 computes $\alpha^{(1)} + c \mod \beta$, which implies $[\alpha + c]_\beta$. Finally, for multiplicative sharings $\langle\alpha\rangle_\beta \cdot \langle\gamma\rangle_\beta$ makes all parties $P_i \in \mathcal{P}$ compute $\alpha_i \cdot \gamma_i \mod \beta$, which implies $\langle\alpha \cdot \gamma\rangle_\beta$.

We let s be a statistical security parameter, p and q denote the two prime candidates and they consist of k bits. We assume, as it is the case in practice, that the length of the primes k satisfies $k > 2s$. Capital P and Q will denote two big primes, such that $n^2 2^{2k} < nP < Q$. It will later be specified, how exactly those are picked.

3.2 Probabilistic Primality Test

We use the Damgård and Mikkelsen [DM10] variant of the original Miller-Rabin test, where the main modification is that the exponentiations are done modulo N. The test takes as input integers f and N such that $f \equiv 3 \mod 4$ and f divides N. Also, $f \in [2^{k-1}, 2^k]$ and $N \in [2^{2k-2}, 2^{2k}]$, for some integer k. To test whether or not f is a prime, the test now proceeds as follows.

1. Choose an element v from $\mathbb{Z}_N$ uniformly at random.
2. Compute $\gamma = v^{\frac{f-1}{2}} \mod N$.
3. If $\gamma \equiv \pm 1 \mod f$, then output `probably-prime`, else output `composite`.

[DM10] gives an upper bound on the error probability, showing that the probability of outputting a composite already gets negligible after just a single iteration, when k is chosen big enough (See Table 1). Note that the reported values are statistical errors which are independent of the computing power of the adversary. Furthermore, since 4096 bit moduli is already the minimum recommended modulus size, 1 iteration of the test should be acceptable in practice.

$\log_2(N) =$	2048	3072	4096
$t = 1$	37	50	61
$t = 2$	67	81	103

Table 1: The table shows $-\log_2(\alpha)$, where α is the probability that, given $N = pq$, either p or q are not prime after t iterations of the test.

[7] This notation does not capture instantiations based on Shamir secret sharing or replicated secret sharing. However, these forms of secret sharing are only used in some cases for implementing the functionality for secure multiplication that we assume access to, and do not appear in the main protocol. Therefore, for simplicity, we restrict the notation to additive sharing.

3.3 From Distributed Biprimality Testing to Distributed Key RSA Generation

Starting from the seminal work by Boneh and Franklin [BF97] all existing distributed RSA key generation protocols we are aware of, with the notable exception of Algesheimer *et al.* [ACS02], follow some standard steps.

In the first step, all the involved parties sample shares of two random numbers which become the prime candidates p, q. Thanks to the prime number theorem, we know that a random k-bit number is prime with probability about $1/k$. Now, the parties run divisibility tests to ensure that each of the sampled numbers is not divisible by small primes. After this sieving step, they run a secure multiplication protocol with input the two prime candidates to get a candidate bi-prime $N = pq$. Once N is known to all the parties, the parties run a bi-primality test to ensure that N is a product of two primes[8]. Finally (in applications that require it) the parties run a distributed computation to securely generate the public key and shared secret key for the RSA cryptosystem [RSA78] or other factoring-based schemes such as Pailler [Pai99,HMRT12,HMR+19]. Note that, in order to avoid having to run for multiple rounds of "rejection sampling" due to the significant probability that N turns out not to be a bi-prime, in many applications it is more desirable to run several instances of the protocol sketched above in parallel.

Crucially, we observe that the bi-primality test is typically the most expensive single step of the entire procedure. For this reason the rest of this paper will focus solely on optimizing this step.

4 Sub-protocols

Our main protocol is built modularly from several subprotocols aka macros. See an overview of all protocols and macros in this paper in Table 2. In this section we describe the subprotocols (or macros) that our protocol is built from. We choose to present the protocol in a modular way, since the subprotocols are used multiple times within the final protocol and they have separate semantic meanings thus simplifying the exposition of the main protocol.

Since the macros take place within a larger protocol, we define them to take place in a certain parametrized context, sometimes with an assumption on how input parameters are distributed. We do so via the **Pre** specification. The actions to be carried out by the participating parties when calling the macro are specified in the **Execute** specification. Finally, for completeness, we specify what is achieved after the execution of the macro in the **Post** specification. All macros are implicitly parameterized by the number of parties n, which is omitted for better readability.

For all macros, we will argue for their correctness and construct a simulator that constructs a view for the corrupt parties. In the simulations, we always assume (without loss of generality) that party P_1 is honest and parties $\mathcal{P} \setminus \{P_1\}$ are corrupt. More concretely, all macro-simulators take the same input as the macro they simulate, together with a state. The idea when simulating is then, that the bigger simulator at some point calls the sub simulator with the already simulated values for its input. A state is passed through all sub simulators, and updated with all simulated values on the way, to enable us to reconstruct correlations when simulating the final view in the protocol calling sub simulators. Thus every macro-simulator outputs an updated state and a sub view that will be used in the main simulation. On top of that, it outputs the simulated values of what the macro would output. In theory, all values that are input to the macro-simulators could also be extracted from the sate by the simulator, and the simulated output could instead be entered into the state. Though we chose this convention to make calling a sub simulator more readable.

4.1 Multiplication

The multiplication macro Mult in Figure 4.1 is different from the others, as it embodies a generic multiplication functionality, instead of an actual implementation. For this generic macro, we simply assume that there exists a simulator that simulates the macro implementing it. As discussed in the introduction, this allows to provide a generic exposition of our protocol that can later be instantiated with different thresholds for

[8] Note that, if the test fails, both p, q need to be discarded since N has been made public.

Ref.	Usage	Notes
	Share and Open	
4.2	$\mathsf{Share}(\mathbb{Z}_A, P_i, x) \to [x]_A$	Party P_i creates a random additive sharing of x in $\mathbb{Z}_A$. Similar for $\langle x \rangle_A$.
4.2	$\mathsf{OpenTo}(\mathbb{Z}_A, P_i, [x]_A) \to x$	Secret value x is revelaed to P_i. Similar for $\langle x \rangle_A$.
4.2	$\mathsf{OpenAll}(\mathbb{Z}_A, [x]_A) \to x$	Secret value x is revealed to everyone. Similar for $\langle x \rangle_A$.
	Local Commands	
3.1	$[x]_A + [y]_A,\ c \cdot [x]_A,\ [x]_A + c$	Local linear computation on shares.
3.1	$\langle x \rangle_A \cdot \langle y \rangle_A$	Local multiplicative computation on shares.
4.3	$\mathsf{Random\text{-}sample}(\mathbb{Z}_A) \to [r]_A$	Distributed generation of random value.
4.4	$\mathsf{Larger\text{-}domain}(\mathbb{Z}_A, \mathbb{Z}_B, [x]_A) \to [x + c_A A]_B$	Local share domain conversion from $\mathbb{Z}_A$ to $\mathbb{Z}_B$ with B sufficiently larger than A. Does not produce random shares.
4.5	$\mathsf{Int\text{-}to\text{-}mod}(\mathbb{Z}_A, [x]_\mathbb{Z}) \to [x \mod A]_A$	Local share domain conversion from integer to $\mathbb{Z}_A$. Does not produce random shares.
	Interactive Subprotocols	
4.1	$\mathsf{Mult}(\mathbb{Z}_A, [x]_A, [y]_A) \to [x \cdot y]_A$	Secure multiplication, which outputs random shares. Can be instantiated in different ways.
4.6	$\mathsf{Mult\text{-}to\text{-}add}(\mathbb{Z}_A, \langle x \rangle_A) \to [x]_A$	Secure conversion of multiplicative into additive sharing.
4.7	$\mathsf{Invert}(\mathbb{Z}_P, [x]_P) \to [x^{-1}]_P$	Secure inverse computation, assumes $x \neq 0$. Only in prime domains $\mathbb{Z}_P$.
4.8	$\mathsf{Membership}(\mathbb{Z}_P, \mathrm{X}, [x]_P) \to [z]_P$	Secure membership testing that outputs $z = 0$ if $x \in \mathrm{X}$, or some non-zero (non-random) value z otherwise. Assumes $x \neq 0$ and $0 \notin \mathrm{X}$. Only in prime domains $\mathbb{Z}_P$.
	Novel Protocols	
5	$\mathsf{Divisible}(k, s, \mathbb{Z}_N, \mathbb{Z}_P, \mathbb{Z}_Q, [\delta]_N, [f]_\mathbb{Z}) \to [y]_Q$	Secure membership testing that outputs $y = 0$ if f divides δ or some non-zero (non-random) value y otherwise. Assumes that f divides N, and that N, P, Q are of appropriate size.
6	$\mathsf{Biprime}(k, s, \mathbb{Z}_P, \mathbb{Z}_Q, N, [p]_\mathbb{Z}, [q]_\mathbb{Z}) \to b$	Secure biprimality testing that outputs whether N is a biprime or not. Assumes $N = pq$ and other conditions on the input shares.

Table 2: Overview of macros, subprotocols, and protocols in this paper.

the number of corrupted parties, using different underlying secure multiplication protocols. On top of this, we will sometimes call the Mult macro with more than two sharing arguments, which should be interpreted as running it multiple times in a row, obtaining the product of all its arguments. Note that this requires a number of rounds logarithmic in the number of inputs. We can also multiply multiple (non-zero) elements in constant round using a randomization trick due to Bar-Ilan and Beaver [BIB89]. The trick is discussed further when implementing the sub-protocol for membership testing later on.

FIGURE 4.1 ($\mathsf{Mult}(\mathbb{Z}_A, [x]_A, [y]_A) \to [x \cdot y]_A$)

This functionality is parametrized by a domain $\mathbb{Z}_A$, and takes as input sharings $[x]_A$ and $[y]_A$.

Pre: All parties hold shares of $[x]_A$ and $[y]_A$.
Post: All parties hold shares of $[z]_A = [x \cdot y \mod A]_A$ with uniformly random $z^{(i)} \in \mathbb{Z}_A$ for all parties i.

Generic interface for secure multiplication

Multiplication of two additively shared values $[x]_A$ and $[y]_A$ will be treated as a black box in the form of a generic macro that once called outputs random shares of $x \cdot y \mod A$ to all parties. Figure 4.1 presents this multiplication functionality. When using it in later macros and the main protocol, one should think of using a securely implemented version of it instead. Having specified multiplication in this way, enables us to use several different secure multiplication protocols in our implementation.

In general, we sometimes withdraw the sharing arguments when referring to a macro. Using the multiplication macro as an example, we sometimes write $\mathsf{Mult}(\mathbb{Z}_A)$, when the sharing inputs are not yet specified.

Simulation. When simulating, we will assume that the macro implementing the multiplication functionality in Figure 4.1 has a simulator that produces a view and a state in the usual manner. We will write

$$([x]_A, \texttt{view}, \texttt{state}_2) \leftarrow \mathcal{S}_{\mathsf{Mult}}(\mathbb{Z}_A, [x_1]_A, \ldots, [x_\ell]_A, \texttt{state}_1),$$

when calling the multiplication simulator with the shared values $x_1, \ldots, x_\ell$. When $\ell > 2$ this is an abbreviation for saying that we call the multiplication simulator $\ell - 1$ times providing each call with the state from the last call. The produced view `view` is then a concatenation of all the views produced by the individual simulator calls.

The simulator for the multiplication protocol, by assumption of the multiplication protocol being secure, produces a view indstingusihable from the multiplication protocols we will be using.

4.2 Basic sub-protocols

We have already (Section 3.1) defined the basic linear operations on additive and multiplicative sharings which are executed locally.

When writing $\mathsf{OpenAll}(\mathbb{Z}_\beta, [\alpha]_\beta)$ (similar for multiplicative sharings), it means that each party $P_i \in \mathcal{P}$ sends $\alpha^{(i)}$ to every other party. Each party then computes $\alpha = \sum_{i=1}^{n} \alpha^{(i)} \mod \beta$. Only opening to party $P_j \in \mathcal{P}$, is done by writing $\mathsf{OpenTo}(\mathbb{Z}_\beta, P_j, [\alpha]_\beta)$ (similar for multiplicative sharings). Finally, $\mathsf{Share}(\mathbb{Z}_\beta, P_j, [\alpha]_\beta)$ makes party P_j share value α in $\mathbb{Z}_\beta$. So party P_j samples values $\alpha^{(1)}, \ldots, \alpha^{(n)}$ uniformly at random such that $\alpha = \sum_{i=1}^{n} \alpha^{(i)} \mod \beta$ and then sends $\alpha^{(i)}$ to party P_i.

Simulation. All commands involving only local share manipulation are trivially simulated by following the protocol instructions. As no information is transferred simulation is perfect. Additive sharing of a value is simulated by selecting random shares for the corrupt party and a dummy share for the honest party, which leads to perfect simulation. When opening a secret value (in the interesting case where at least one of the corrupted parties learns the result) is as usual simulated by first simulating the value to be opened (all values that will be opened will either be random or a function of the desired output of the function), and then setting the share of the honest party to the only value that matches the value to be opened and the shares of the corrupt parties (that the simulator keeps track of within its state). This leads to an indistinguishable view as long as we can argue that the adversary had no information about the share of the honest party right before the opening process.

4.3 Sample random shared value

Macro $\mathsf{Random\text{-}sample}$ (Figure 4.2), samples some uniformly random value $r \in \mathbb{Z}_A$ in a distributed manner by making all parties sample their share $r^{(i)}$ uniformly at random. This results in a uniformly random shared value r in $\mathbb{Z}_A$.

Simulation. Figure 4.3 provides a simulator for the $\mathsf{Random\text{-}sample}$ macro. As no communication is happening, the only thing it does is to pick every share at random. The state is updated with the entire sharing $[r]_A$, while the view is constructed only with the corrupt shares.

Since no communication is involved, and the shares of the corrupt parties are sampled at random, it is straightforward to argue that the view of the corrupt parties produced by the simulator is identical to their view in the protocol.

FIGURE 4.2 (Random-sample($\mathbb{Z}_A$) $\to [r]_A$)

This macro is parametrized by a domain $\mathbb{Z}_A$.

Post: All parties hold shares of $[r]_A$ with $r \in \mathbb{Z}_A$ being sampled uniformly at random.
Execute: Each party $P_i \in \mathcal{P}$ proceeds as follows:

1. Sample $r^{(i)} \in \mathbb{Z}_A$ uniformly at random.

Secure sampling of a random shared value

FIGURE 4.3 ($\mathcal{S}_{\text{Random-sample}}(\mathbb{Z}_A, \texttt{state}_1) \to ([r]_A, \texttt{view}, \texttt{state}_2)$)

This simulator takes as input a state $\texttt{state}_1$, and produces a new state $\texttt{state}_2$ and a view $\texttt{view}$ for the macro.

Simulate:

1. Sample sharing $[r]_A = (r^{(1)}, \dots, r^{(n)})$ uniformly at random
2. Set $\texttt{state}_2 = \texttt{state}_1 \cup \{[r]_A\}$
3. Set $\texttt{view} = \{r^{(2)}, \dots, r^{(n)}\}$
4. Return $([r]_A, \texttt{view}, \texttt{state}_2)$

Simulator for the Random-sample macro

4.4 From smaller to large domain

Macro Larger-domain (Figure 4.4) lifts shares from one integer-domain to a larger one, by having all parties interpret their shares in the new domain. This introduces an error, which can be bound as a function of the number of parties and the size of the shares. Note that the produced shares are not uniformly distributed, but this is enough in the context where the protocol is used later on.

FIGURE 4.4 (Larger-domain($\mathbb{Z}_A, \mathbb{Z}_B, [x]_A$) $\to [x + c_A A]_B$)

This macro is parametrized by domains $\mathbb{Z}_A, \mathbb{Z}_B$ such that $B > nA$, and takes as input a sharing $[x]_A$.

Pre: All parties hold shares of $[x]_A$.
Post: All parties hold shares of $[y]_B = [x + c_A A]_B$ with $c_A < n$.
Execute: Each party $P_i \in \mathcal{P}$, proceeds as follows:

1. Interpret $[x]_A$ as shares in $\mathbb{Z}_B$ i.e., set$y^{(i)} = x^{(i)}$.

Secure conversion to larger domain

Correctness. Note that the protocol simply interprets shares from the source domain as shares in the target domain. This works, since

$$\begin{aligned}\sum_{i\in[n]} y^{(i)} \mod B &= \sum_{i\in[n]} x^{(i)} \mod B \\ &= x + c_A A \mod B = x + c_A A,\end{aligned}$$

where $c_A < n$, due to all the shares $x^{(i)}$ being less than A. The last equality holds over the integers since we assumed $B > nA$.

Simulation. The simulator for the Larger-domain macro is provided in Figure 4.5 and the only thing it does is to update the state with the output sharing. As for other macros which do not involve communication, the simulation of the view of the corrupt parties is trivially perfect.

FIGURE 4.5 ($\mathcal{S}_{\text{Larger-domain}}(\mathbb{Z}_A, \mathbb{Z}_B, [x]_A, \texttt{state}_1) \to ([y]_B, \texttt{view}, \texttt{state}_2)$)

This simulator takes as input domains $\mathbb{Z}_A$ and $\mathbb{Z}_B$, sharing $[x]_A$, and a state $\texttt{state}_1$. It produces a new state $\texttt{state}_2$ and a view $\texttt{view}$ for the macro.

Simulate:
1. Set $[y]_B = (x^{(1)}, \ldots, x^{(n)})$
2. Set $\texttt{state}_2 = \texttt{state}_1 \cup \{[y]_B\}$
3. Return $([y]_B, \{\}, \texttt{state}_2)$

Simulator for the Larger-domain macro

4.5 From integer shares to constrained domain

Figure 4.6 shows macro Int-to-mod, which converts the integer sharing of a value, to a sharing in some constrained integer domain. The only thing it makes the parties do, is to perform a modulo operation, which restricts their shares to the constrained domain. Note that also this macro produces shares that are not uniformly random in the new domain.

Correctness. Summing over the newly computed shares in the target domain, gives us the following

$$\begin{aligned} \sum_{i\in[n]} y^{(i)} \mod A &= \sum_{i\in[n]} (x^{(i)} \mod A) \mod A \\ &= \sum_{i\in[n]} x^{(i)} \mod A \\ &= x \mod A, \end{aligned}$$

which is exactly what the macro promises after being invoked.

FIGURE 4.6 (**Int-to-mod**$(\mathbb{Z}_A, [x]_\mathbb{Z}) \to [x \mod A]_A$)

This macro is parametrized by a domain $\mathbb{Z}_A$, and takes as input a sharing $[x]_\mathbb{Z}$.

Pre: All parties hold shares of $[x]_\mathbb{Z}$.
Post: All parties hold shares of $[y]_A = [x \mod A]_A$.
Execute: Each party $P_i \in \mathcal{P}$, proceeds as follows:
1. Interpret $[x]_\mathbb{Z}$ as shares in $\mathbb{Z}_A$ i.e., set $y^{(i)} = x^{(i)} \mod A$.

Secure conversion from integer to modular shares

Simulation. The simulator for the Int-to-mod macro is provided in Figure 4.7 and the only thing it does is to constrain the shares to $\mathbb{Z}_A$ and then update the state with them. As for other macros which do not involve communication, the simulation of the view of the corrupt parties is trivially perfect.

4.6 From multiplicative to additive shares

Figure 4.8 shows a protocol that converts the multiplicative sharing of a value to an additive sharing of the same value using $O(n)$ multiplications. In Section 6.2, we describe a variant of this protocol that only requires $O(1)$ online multiplications, assuming preprocessing. The basic version of the macro, simply makes all parties additively share their shares of the input. It then makes the parties compute the product of the n sharings, producing an additive sharing of the input (as already discussed, this can be done in either $O(\log n)$ or $O(1)$ rounds)

FIGURE 4.7 ($\mathcal{S}_{\text{Int-to-mod}}(\mathbb{Z}_A, [x]_\mathbb{Z}, \texttt{state}_1) \rightarrow ([y]_A, \texttt{view}, \texttt{state}_2)$)

This simulator takes as input a domains $\mathbb{Z}_A$, sharing $[x]_\mathbb{Z}$, and a state $\texttt{state}_1$. It produces a new state $\texttt{state}_2$ and a view $\texttt{view}$ for the macro.

Simulate:

1. Set $[y]_A = (x^{(1)} \mod A, \ldots, x^{(n)} \mod A)$
2. Set $\texttt{state}_2 = \texttt{state}_1 \cup \{[y]_A\}$
3. Return $([y]_A, \{\}, \texttt{state}_2)$

Simulator for the Int-to-mod macro

FIGURE 4.8 (Mult-to-add($\mathbb{Z}_A, \langle x \rangle_A) \rightarrow [x]_A$)

This macro is parametrized by a domain $\mathbb{Z}_A$, and takes as input a sharing $\langle x \rangle_A$. The macro assumes access to Mult($\mathbb{Z}_A$).

Pre: All parties hold multiplicative shares of $\langle x \rangle_A = (x_1, \ldots, x_n)$.
Post: All parties hold additive shares of $[x]_A$.
Execute: Each party $P_i \in \mathcal{P}$, proceeds as follows: let x_i be P_i's share of $\langle x \rangle_A$, then

1. $[x_i]_A \leftarrow$ Share($\mathbb{Z}_A, P_i, x_i$)
2. Calculate $[x]_A \leftarrow$ Mult($\mathbb{Z}_A, [x_1]_A, \ldots, [x_n]_A$)

Secure conversion from multiplicative to additive shares

Correctness. For correctness, we only need that the produced sharing is actually a sharing of x, which it is since

$$\sum_{i=1}^{n} x^{(i)} \mod A = \prod_{i=1}^{n} x_i \mod A = x.$$

Simulation. The simulator first samples the additive sharings of the honest party multiplicative share uniformly at random, while it generates additive shares for the multiplicative shares of the corrupt parties following the protocol. It then calls the multiplication simulator and constructs the view by combining the multiplication view with all but one of the uniformly random values. The only share that is not included in the view, is the first party's own share of her share x_1, as everything else is either on the corrupt parties' random tape or in their transcript. The view of the corrupted parties produced by the simulator is distributed identically as in the protocol, thanks to the assumption on the underlying multiplication protocol, and the fact that the honest party (additive) share of their own (multiplicative) share is never revealed. The simulator is given in Figure 4.9.

4.7 Computing Inverses

Macro Invert (Figure 4.10), given an additively shared x in some prime field $\mathbb{Z}_P$, computes shares of its inverse x^{-1}. The macro is a special case of the one used by Algesheimer *et al.* [ACS02], which is originally due to Bar-Ilan and Beaver [BIB89]. The main underlying idea is that performing the inversion in the cleartext space is much more efficient than doing so in the secret-shared domain and, thanks to the assumption that $x \neq 0$, we can invert a randomized version of the input instead, and then correct the result in the secret-shared domain.

Correctness. First note that in macro Invert, x is invertible, since P is a prime and it is required that $x \neq 0$. To see that the macro actually computes the inverse of $x \in \mathbb{Z}_P$, let us sum over the shares that are output

FIGURE 4.9 ($\mathcal{S}_{\text{Mult-to-add}}(\mathbb{Z}_A, \langle x\rangle_A, \texttt{state}_1) \to ([x]_A, \texttt{view}, \texttt{state}_2)$)

This simulator takes as input a domains $\mathbb{Z}_A$, sharing $\langle x\rangle_A$, and a state $\texttt{state}_1$. It produces a new state $\texttt{state}_2$ and a view $\texttt{view}$ for the macro.

Simulate:

```
1. Sample sharing $[x_1]_A = (x_1^{(1)}, \ldots, x_1^{(n)})$ uniformly at random
2. For $i = 2, \ldots, n$
   (a) Sample sharing $[x_i]_A = (x_i^{(1)}, \ldots, x_i^{(n)})$ uniformly at random under the constraint that their
       sum in $\mathbb{Z}_A$ is equal to $x_i$.
3. Call $([x]_A, \texttt{view}_1, \texttt{state}_2) \leftarrow \mathcal{S}_{\text{Mult}}(\mathbb{Z}_A, [x_1]_A, \ldots, [x_n]_A, \texttt{state}_1)$
4. Construct $\texttt{state}_3 = \texttt{state}_2 \cup \{[x_1]_A, \ldots, [x_n]_A\}$
5. Construct $\texttt{view} = \texttt{view}_1 \cup \{(x_1^{(2)}, \ldots, x_1^{(n)}), (x_2^{(1)}, \ldots, x_2^{(n)}), \ldots, (x_n^{(1)}, \ldots, x_n^{(n)})\}$
6. Return $([x]_A, \texttt{view}, \texttt{state}_3)$
```

Simulator for the Mult-to-add macro

FIGURE 4.10 (**Invert**$(\mathbb{Z}_P, [x]_P) \to [x^{-1}]_P$)

This protocol is parametrized by domain $\mathbb{Z}_P$ with $P \in \mathbb{N}$ being a prime, and takes as input a sharing $[x]_P$. The macro assumes access to Mult$(\mathbb{Z}_P)$, and Random-sample$(\mathbb{Z}_P)$.

Pre: All parties hold additive shares of $[x]_P$ such that $x \neq 0$.
Post: All parties hold shares of $[x^{-1}]_P$.
Execute: Each party $P_i \in \mathcal{P}$, proceed as follows:

```
1. Obtain $[r]_P \leftarrow$ Random-sample$(\mathbb{Z}_P)$.
2. Compute $v \leftarrow$ OpenAll$(\mathbb{Z}_P,$ Mult$(\mathbb{Z}_P, [r]_P, [x]_P))$
3. Compute locally $[x^{-1}]_P = v^{-1} \cdot [r]_P$.
```

Secure computation of inverse

at the end;

$$\begin{aligned}\sum_{i=1}^{n} z^{(i)} \mod P &= \sum_{i=1}^{n} (r \cdot x)^{-1} r^{(i)} \mod P \\ &= x^{-1} \cdot r^{-1} \sum_{i=1}^{n} r^{(i)} \mod P \\ &= x^{-1} \mod P,\end{aligned}$$

where $(z^{(1)}, \ldots, z^{(n)})$ denotes the sharing $[x^{-1}]_P$ which is output by the macro.

As a final remark, we note that the only way rx can be non-invertible, is if $r = 0$ This happens only with negligible probability in s though, due to our assumptions on P, k and s; $P > n2^{2k} > n2^{4s}$.

Simulation. Here the simulator first calls simulators $\mathcal{S}_{\text{Random-sample}}$ and $\mathcal{S}_{\text{Mult}}$, and then executes the same calculation as the simulated macro, except that it reveals a uniformly random value v' instead of the output of the calculations (when opening, it sets the share of the honest party to be consistent with v' and the shares of the corrupt parties that the simulator keeps track of). In order to argue that the view produced by the simulator is indistinguishable to the one of the real protocol, it is crucial to observe that the opened value v is the direct output of a (secure) multiplication protocol, and therefore the honest party's share of v is uniformly random in the view of the adversary right before the opening. Thus, the simulator can freely "lie" about it. Then, notice that the opened value v' is distributed uniformly to the actual value v, since r is random and not used anywhere else and $x \neq 0$. The simulator is provided in Figure 4.11.

FIGURE 4.11 ($\mathcal{S}_{\textsf{Invert}}(\mathbb{Z}_P, [x]_P, \texttt{state}_1) \rightarrow ([x^{-1}]_P, \texttt{view}, \texttt{state}_2)$)

This simulator takes as input a domains $\mathbb{Z}_P$ with $P \in \mathbb{N}$ being prime, sharing $[x]_P$ and a state $\texttt{state}_1$. It produces a new state $\texttt{state}_2$ and a view $\texttt{view}$ for the macro.

Simulate:

```
1. Call $([r]_P, \texttt{view}_1, \texttt{state}_2) \leftarrow \mathcal{S}_{\textsf{Random-sample}}(\mathbb{Z}_P, \texttt{state}_1)$
2. Call $([v]_P, \texttt{view}_2, \texttt{state}_3) \leftarrow \mathcal{S}_{\textsf{Mult}}(\mathbb{Z}_P, [r]_P, [x]_P, \texttt{state}_2)$
3. Pick $v' \in \mathbb{Z}_P$ uniformly at random
4. Set $v^{(1)} = v' - \sum_{i=2}^{n} v^{(i)}$
5. Compute $y = v'^{-1} \mod P$
6. Set $[x^{-1}]_P = (y \cdot r^{(1)}, \dots, y \cdot r^{(n)})$
7. Construct $\texttt{state}_4 = \texttt{state}_3 \cup \{[x^{-1}]_P\}$
8. Construct $\texttt{view} = \texttt{view}_1 \cup \texttt{view}_2 \cup \{v^{(1)}\}$
9. Return $([x^{-1}]_P, \texttt{view}, \texttt{state}_4)$
```

Simulator for the Invert macro

4.8 Computing Set-Membership

Figure 4.12 provides macro Membership which checks if a secret shared value $[x]_P$ is a member of some set Δ. It does so by computing $[z]_P = \prod_{\delta \in \Delta}([x]_P - \delta)$. This computation will result in $z = 0$ if and only if $x \in \Delta$ and some non-zero value otherwise. Note that this non-zero value might leak some information about the input. A (standard) easy way of dealing with this is to multiply the result with a random value before opening it, but this is not necessary in the context in which the protocol will be used (looking ahead, we will multiply the result of two membership tests together and we are only interesting in protecting the inputs in the case at least one of the two values is 0, which leads to the result being 0 regardless of which of the two items was in the set). [9]

Note that computing this product in the naive way would result in $\log_2(|\Delta|)$ rounds, when computing the product in a binary tree structure (where the product of two numbers is multiplied with the product of two other numbers). To improve efficiency and get a constant round protocol, we exploit an idea by Bar-Ilan and Beaver [BIB89]. What Bar-Ilan and Beaver shows is that when wanting to compute $x_1 \cdot x_2 \cdot x_3$ it is possible to compute and open $y_1 = x_1 \cdot r_1^{-1} \cdot r_2$, $y_2 = x_2 \cdot r_2^{-1} \cdot r_3$ and $y_3 = x_3 \cdot r_3^{-1} \cdot r_1$ in parallel (and therefore constant rounds). It is then possible to compute $y_1 \cdot y_2 \cdot y_3 = x_1 \cdot x_2 \cdot x_3$ in the open. This of course scales for arbitrary-sized products. Unfortuantely this does not directly work for set-membership, since $x_i = x - \delta = 0$ when $x = \delta \in \Delta$. Thus there will be a value $y_i = 0$ for some i, which leaks exactly which element in Δ that is equal to x. To prevent this leakage we instead suggest considering the polynomial $F(X) = \prod_{\delta \in \Delta}(X - \delta) = X^m + c_{m-1} \cdot X^{m-1} + \cdots + c_1 \cdot X + c_0$ where $m = |\Delta|$ and $c_i \in \mathbb{Z}_P$. When testing for membership, we can then evaluate this polynomial, by computing the powers x^i of the input value for $i \in [2; m]$. Though, we now need to assume that the input satisfies $x \neq 0$ to avoid leaking auxiliary information. Note also, that the computation of x^{i-1} can be used in x^i, by opening and reusing partial products. We avoid leakage by picking $m-1$ auxiliary random values, α_i for $i \in [2, m]$ and opening values equal to $x^i \cdot \alpha_i$ to avoid leaking information, while enabling the evaluation of the polynomial. At the same time the values $x^{i-1} \cdot r_{i-1}^{-1}$ are used with $x \cdot r_{i-1}$ to allow reusing partial computations. [10]

Correctness. First see that, since $x \neq 0$ we know that $v_i, w_i \neq 0$ for all $i \in [2; m]$ except with negligible probability (less than $2/P$), since α_i and r_i are uniformly random sampled from $\mathbb{Z}_P$. This implies that in

[9] Note that in case where one needs the answer as a bit in secret shared format, one can use a technique by Tomas Toft [Tof07, Sec. 9.2], to get this with $O(s)$ extra multiplications, except with negligible probability.

[10] We note that if Δ is a contiguous range of integers, e.g., $\Delta = \{5, 6, 7, 8, 9\}$ then the membership testing problem could also be solved using two comparisons. This can be achieved in constant rounds and linear complexity in $\log_2(|\Delta|)$ [Cd10].

FIGURE 4.12 (Membership$(\mathbb{Z}_P, \mathit{\Delta}, [x]_P) \rightarrow [z]_P$)

This macro is parametrized by domain $\mathbb{Z}_P$ with prime $P \in \mathbb{N}$, and takes as input a set $\mathit{\Delta} \subseteq \mathbb{Z}_P^*$ and a sharing $[x]_P$. Define $m = |\mathit{\Delta}|$. Assumes access to Random-sample$(\mathbb{Z}_P)$, Invert$(\mathbb{Z}_P)$ and Mult$(\mathbb{Z}_P)$.

Pre: All parties hold additive shares of $[x]_P$, such that $x \neq 0$ and $\delta \neq 0\ \forall \delta \in \mathit{\Delta}$.
Post: All parties hold random shares of $[z]_P$ with $z = 0$ if $x \in \mathit{\Delta}$ and a (non-random) zero-value in $\mathbb{Z}_P^*$ otherwise.

```
Membership: Each party $P_j \in \mathcal{P}$, proceed as follows:
  1. Call Random-sample$(\mathbb{Z}_P)$ $2m-2$ times to sample:
     – $[r_i]_P$ for $i \in [2;m]$;
     – $[\alpha_i]_P$ for $i \in [2;m]$
  2. Use Invert$(\mathbb{Z}_P)$ $2m-2$ times to compute
     – $[r_i^{-1}]_P \leftarrow$ Invert$(\mathbb{Z}_P, [r_i]_P)$ for $i \in [2;m]$.
     – $[\alpha_i^{-1}]_P \leftarrow$ Invert$(\mathbb{Z}_P, [\alpha_i]_P)$ for $i \in [2;m]$.
  3. Use Mult$(\mathbb{Z}_P)$ $3m-5$ times to compute the values:
     – $[t_i]_P \leftarrow$ Mult$(\mathbb{Z}_P, [x]_P, [r_i^{-1}]_P)$ for $i \in [2;m]$ and let $[t_1]_P = [x]_P$.
     – $[v_i]_P \leftarrow$ Mult$(\mathbb{Z}_P, [\alpha_i]_P, [t_i]_P)$ for $i \in [2;m]$
     – $[w_i]_P \leftarrow$ Mult$(\mathbb{Z}_P, [t_{i-1}]_P, [r_i]_P)$ for $i \in [2;m]$
  4. For $i \in [2;m]$ reveal $v_i \leftarrow$ OpenAll$(\mathbb{Z}_P, [v_i]_P)$ and $w_i \leftarrow$ OpenAll$(\mathbb{Z}_P, [w_i]_P)$
  5. Locally compute $y_i = v_i \prod_{j=2}^{i} w_j \mod P$ for $i \in [2;m]$.        // $y_i = \alpha_i \cdot x^i$
  6. Locally compute the coefficients $c_i \in \mathbb{Z}_P, i \in [0;m-1]$ of the polynomial over $X$ as
        $$F(X) = \prod_{\delta \in \mathit{\Delta}} (X - \delta) = X^m + c_{m-1} \cdot X^{m-1} + \cdots + c_1 \cdot X + c_0$$
  7. Compute and return        // $[z]_P = F([x]_P)$
        $$[z]_P = [\alpha_m^{-1}]_P \cdot y_m + c_{m-1} \cdot [\alpha_{m-1}^{-1}]_P \cdot y_{m-1} + \cdots + c_2 \cdot [\alpha_2^{-1}]_P \cdot y_2 + c_1 \cdot [x] + c_0$$
```

Secure membership testing of a secret value in a public set.

step 5, $y_i \neq 0$ and thus for $i \in [2;m]$:

$$\begin{aligned} y_i &= v_i \prod_{j=2}^{i} w_j \mod P \\ &= \alpha_i \cdot x \cdot r_i^{-1} \prod_{j=2}^{i} x \cdot r_{j-1}^{-1} \cdot r_j \mod P \\ &= \alpha_i \cdot x^i \ . \end{aligned}$$

The last equality follows due to all the r_j values canceling out from r_2 up to r_i, as there is exactly one r_j and one r_j^{-1} for $j \in [i]$ in the product (in the above product r_1 and therefore r_1^{-1} are both equal to 1). Next we will argue that value z computed in step 7, is actually the polynomial evaluated on $[x]_P$;

$$\begin{aligned} z &= \alpha_m^{-1} \cdot y_m + c_{m-1} \cdot \alpha_{m-1}^{-1} \cdot y_{m-1} + \cdots + c_2 \alpha_2^{-1} \cdot y_2 + c_1 \cdot x + c_0 \\ &= x^m + c_{m-1} \cdot x^{m-1} + \cdots + c_1 \cdot x + c_0 \\ &= F(x), \end{aligned}$$

where the second equality follows from the fact that $\alpha_i^{-1} \cdot y_i = \alpha_i^{-1} \cdot (\alpha_i \cdot x^i) = x^i$. Since each $\delta \in \mathit{\Delta}$ is a root of the polynomial $F(X)$, it is clear that if $x \in \mathit{\Delta}$ then $F(x) = 0$. If on the other hand $x \notin \mathit{\Delta}$ then, per the argument above, $F(x) \neq 0$ except with negligible probability in P.

Simulation. As before, the simulation calls the already defined subsimulators, does the exact same calculations as the macro and updates the view and the state along the way. When opening the values in step 4, the simulator instead picks some random values v_i', w_i' and opens these instead, faking the share of the honest party to be consistent with these values and the shares of the corrupted parties that the simulator keeps track of. Note that both the values being opened are the fresh output of a multiplication protocol, where the values are multiplied by a random mask that has not been used anywhere else (α_i for v_i and r_i for w_i). Note finally that the output of the protocol is not guaranteed to be random. This will not matter in the context where the protocol is used. The membership macro is simulated by Figure 4.13.

FIGURE 4.13 ($\mathcal{S}_{\text{Membership}}(\mathbb{Z}_P, \Delta, [x]_P, \texttt{state}_1) \rightarrow ([z]_P, \texttt{view}, \texttt{state}_2)$)

This simulator takes as input a domains $\mathbb{Z}_P$ with $P \in \mathbb{N}$ being a prime, sharing $[x]_P$, a set Δ of size m, and a state $\texttt{state}_1$. It produces a new state $\texttt{state}_2$ and a view $\texttt{view}$ for the macro.

Simulate:

```
1. Let $[r_1]_P$ be canonical sharing of 1
2. Set $\texttt{view} = \{\}$
3. Set $\texttt{state}_0 = \{\}$
4. For $i = 2, \dots, m$
   (a) Call $([r_i]_P, \texttt{view}_1, \texttt{state}_1) \leftarrow \mathcal{S}_{\text{Random-sample}}(\mathbb{Z}_P, \texttt{state}_0)$
   (b) Call $([r_i^{-1}]_P, \texttt{view}_2, \texttt{state}_2) \leftarrow \mathcal{S}_{\text{Invert}}(\mathbb{Z}_P, [r_i]_P, \texttt{state}_1)$
   (c) Call $([\alpha_i]_P, \texttt{view}_3, \texttt{state}_3) \leftarrow \mathcal{S}_{\text{Random-sample}}(\mathbb{Z}_P, \texttt{state}_2)$
   (d) Call $([\alpha_i^{-1}]_P, \texttt{view}_4, \texttt{state}_4) \leftarrow \mathcal{S}_{\text{Invert}}(\mathbb{Z}_P, [\alpha_i]_P, \texttt{state}_3)$
   (e) Set $\texttt{view} = \texttt{view} \cup \texttt{view}_1 \cup \texttt{view}_2 \cup \texttt{view}_3 \cup \texttt{view}_4$
   (f) Set $\texttt{state}_0 = \texttt{state}_4 \cup \{[r_i]_P, [r_i^{-1}]_P, [\alpha_i]_P, [\alpha_i^{-1}]_P\}$
5. Call $([t_1]_P, \texttt{view}_5, \texttt{state}_1) \leftarrow \mathcal{S}_{\text{Mult}}(\mathbb{Z}_P, [x]_P, [r_1^{-1}]_P, \texttt{state}_0)$
6. Set $\texttt{view} = \texttt{view} \cup \texttt{view}_5$
7. For $i = 2, \dots, m$
   (a) Call $([t_i]_P, \texttt{view}_6, \texttt{state}_2) \leftarrow \mathcal{S}_{\text{Mult}}(\mathbb{Z}_P, [x]_P, [r_i^{-1}]_P, \texttt{state}_1)$
   (b) Call $([v_i]_P, \texttt{view}_7, \texttt{state}_3) \leftarrow \mathcal{S}_{\text{Mult}}(\mathbb{Z}_P, [\alpha_i]_P, [t_i]_P, \texttt{state}_2)$
   (c) Call $([w_i]_P, \texttt{view}_8, \texttt{state}_4) \leftarrow \mathcal{S}_{\text{Mult}}(\mathbb{Z}_P, [t_{i-1}]_P, [r_i]_P, \texttt{state}_3, )$
   (d) Pick $v_i' \in \mathbb{Z}_P$ uniformly at random
   (e) Pick $w_i' \in \mathbb{Z}_P$ uniformly at random
   (f) Set $v_i^{(1)} = v_i' - \sum_{j=2}^{n} v_i^{(j)}$
   (g) Set $w_i^{(1)} = w_i' - \sum_{j=2}^{n} w_i^{(j)}$
   (h) Compute $y_i = v_i' \prod_{j=2}^{i} w_j' \mod P$
   (i) Set $\texttt{view} = \texttt{view} \cup \texttt{view}_6 \cup \texttt{view}_7 \cup \texttt{view}_8 \cup \{v_i^{(1)}, w_i^{(1)}\}$
   (j) Set $\texttt{state}_1 = \texttt{state}_4 \cup \{y_i\}$
8. For $i \in [0; m-1]$, compute the coefficient $c_i$ of polynomial $\prod_{\delta \in \Delta}(X - \delta)$
9. Calculate $[z]_P$ such that
```

$$z^{(1)} = (\alpha_m^{-1})^{(1)} \cdot y_m + c_{m-1} \cdot (\alpha_{m-1}^{-1})^{(1)} \cdot y_{m-1} + \dots + c_2 \cdot (\alpha_2^{-1})^{(1)} \cdot y_2 + c_1 \cdot x^{(1)} + c_0$$
$$z^{(i)} = (\alpha_m^{-1})^{(i)} \cdot y_m + c_{m-1} \cdot (\alpha_{m-1}^{-1})^{(i)} \cdot y_{m-1} + \dots + c_2 \cdot (\alpha_2^{-1})^{(i)} \cdot y_2 + c_1 \cdot x^{(i)} \forall i = 2, \dots, n$$

```
10. Construct $\texttt{state} = \texttt{state}_1 \cup \{c_0, \dots, c_{m-1}, [z]_P\}$
11. Return $([z]_P, \texttt{view}, \texttt{state})$
```

Simulator for the Membership macro

5 New divisibility test

We present now our novel distributed divisibility test Divisible (Figure 5.1), which uses uses several macros from Section 4: Assuming that the parties have shares of $[\delta]_N$ over $\mathbb{Z}_N$, and $[f]_\mathbb{Z}$ over the integers[11], Divisible outputs whether or not f divides δ. On top of having distributed shares of $[\delta]_N$ and $[f]_\mathbb{Z}$, we as well assume that f divides N. As proven below, the procedure works except with a negligible probability of false positives.

Let k denote the length of f, i.e. $f \in [2^{k-1}, 2^k]$ and $N \in [2^{2k-2}, 2^{2k}]$. As part of the setup, we need two big primes P and Q, such that $n^2 2^{2k} < nP < Q < T$ (i.e. $n^2 N < nP < Q < T$) and Q is chosen in $]P, T[$ at random. We also assume some statistical security parameter s, such that $2s < k$, which implies that $f, Q > 2^s$. Here, the upper bound T on Q is chosen such that there are 2^s primes between P and T. A concrete expression for T can easily be derived from the prime number theorem, $T = 2P$ would be sufficient. In the argument for the divisibility test, it will be important that Q is independent of the inputs to the divisibility test. However, as we work with passive security, P and Q can be picked and fixed before the protocol is run.

FIGURE 5.1 (Divisible$(k, s, \mathbb{Z}_N, \mathbb{Z}_P, \mathbb{Z}_Q, [\delta]_N, [f]_\mathbb{Z}) \to [y]_Q$)

This protocol is parameterized by the length of the tested integer k and a security parameter s s.t $k > 2s$. It is also parameterized by domains $\mathbb{Z}_P$ and $\mathbb{Z}_Q$ such that $n^2 2^{2k} < nP < Q$, P and Q are primes and Q is chosen at random independent of P. Finally $\mathbb{Z}_N$ is a domain such that $N \in [2^{2k-2}, 2^{2k}]$. The protocol uses Mult$(\mathbb{Z}_A)$, Int-to-mod$(\mathbb{Z}_A)$, Larger-domain$(\mathbb{Z}_A, \mathbb{Z}_B)$, Invert$(\mathbb{Z}_A)$ and Membership$(\mathbb{Z}_A)$ for various combinations of $A, B \in \{N, P, Q\}$.

Pre: All parties hold additive shares of $[\delta]_N$ and $[f]_\mathbb{Z}$ such that f divides N and $f \in [2^{k-1}, 2^k]$.
Post: All parties hold additive shares of $[y]_Q$, with $y = 0$ if f divides δ and a (non-random) value in $\mathbb{Z}_Q^*$ otherwise.
Execute: Each party $P_i \in \mathcal{P}$ proceeds as follows:

```
 1. Convert $[\delta + c_1 N]_P \leftarrow$ Larger-domain$(\mathbb{Z}_N, \mathbb{Z}_P, [\delta]_N)$
 2. Convert $[\delta + c_1 N]_Q \leftarrow$ Larger-domain$(\mathbb{Z}_N, \mathbb{Z}_Q, [\delta]_N)$
 3. Convert $[f]_P \leftarrow$ Int-to-mod$(\mathbb{Z}_P, [f]_\mathbb{Z})$
 4. Convert $[f]_Q \leftarrow$ Int-to-mod$(\mathbb{Z}_Q, [f]_\mathbb{Z})$
 5. Compute $[f^{-1}]_P \leftarrow$ Invert$(\mathbb{Z}_P, [f]_P)$
 6. Compute $[f^{-1}]_Q \leftarrow$ Invert$(\mathbb{Z}_Q, [f]_Q)$
 7. Compute $[a]_P \leftarrow$ Mult$(\mathbb{Z}_P, [\delta + c_1 N]_P, [f^{-1}]_P)$
 8. Compute $[b]_Q \leftarrow$ Mult$(\mathbb{Z}_Q, [\delta + c_1 N]_Q, [f^{-1}]_Q)$
 9. Convert $[a + c_3 P]_Q \leftarrow$ Larger-domain$(\mathbb{Z}_P, \mathbb{Z}_Q, [a]_P)$
10. Compute $[z]_Q = [a + c_3 P]_Q - [b]_Q$
11. Test $[y]_Q \leftarrow$ Membership$(\mathbb{Z}_Q, \{1, \ldots, n\}, (P^{-1} \cdot [z]_Q) + 1)$
```

Protocol for passively secure divisibility testing

Correctness. The correctness of our divisibility test is summarized in the following Lemma, which is proven below.

Lemma 5.2. *Consider the protocol from Figure 5.1 and assume of the input that $f \in [2^{k-1}, 2^k]$ and $N \in [2^{2k-2}, 2^{2k}]$. If f divides δ, all parties will end up with an additive sharing of 0. If on the other hand f does not divide δ, all parties obtain an additive share of a non-zero value, except with negligible probability in s.*

Proof. First note, that both pairs (N, P) and (N, Q) satisfy the conditions of Larger-domain since both $P > nN$ and $Q > nN$. Therefore, lifting δ to domain $\mathbb{Z}_P$ and $\mathbb{Z}_Q$ respectively in steps 1 and 2, results in the

[11] When using this divisibility test later, f will correspond to some candidate RSA prime and δ will correspond to $\gamma \pm 1$ in the Miller-Rabin primality test.

same constant c_1, since there is no overflow in either case. Now remember that we assumed that f divides N, which implies that in $\mathbb{Z}_Q$

$$\begin{aligned}[b]_Q &= [\delta + c_1 N]_Q \cdot [f^{-1}]_Q \\ &= [\delta f_Q^{-1} + c_1 g \mod Q]_Q \\ &= [(\delta f_Q^{-1} \mod Q) + c_1 g - b_1 Q]_Q,\end{aligned} \tag{1}$$

where $N = fg$ for some $g < Q/n^2$ and $f_Q^{-1} = (f^{-1} \mod Q)$ denotes the inverse of f in $\mathbb{Z}_Q$. We can upper-bound g, since N and thus its factors f and g are smaller than Q/n^2 by construction. Then in the above, $b_1 \in \{0, 1\}$, since $c_1 \leq n$ and thus $c_1 g$ can only make the sum overflow at most once.

Similarly, we have that lifting a to domain $\mathbb{Z}_Q$ in step 9 gives

$$[a + c_2 P]_Q = [(\delta f_P^{-1} \mod P) + c_1 g - b_2 P + c_2 P]_Q, \tag{2}$$

where g is the same as before and $f_P^{-1} = (f^{-1} \mod P)$ denotes the inverse of f in $\mathbb{Z}_P$. By the same reasoning as before, $b_2 \in \{0, 1\}$, as $c_1 g$ can only trigger a single overflow.

Summarizing the constants, we have two bits b_1 and b_2 that arose from moving $c_1 g$ out of the modulus, and two constants $c_1, c_2 \leq n$ stemming from domain changes. The two divisibility cases will now be handled separately in the following.

$\underline{f|\delta}$: In this case, there exists some non-negative constant $\alpha < \delta \leq N < Q$, such that $\delta = \alpha f$. Since $\delta < Q$, this translates directly to the $\mathbb{Z}_Q$ domain, as $\delta \equiv \alpha f \mod Q$. But then multiplying by the inverse of f in $\mathbb{Z}_Q$, which exists as Q is prime, we get that

$$\delta f_Q^{-1} \equiv \alpha \mod Q.$$

Since $N < P$ as well, the same reasoning gives us that

$$\delta f_P^{-1} \equiv \alpha \mod P.$$

Combining those two observations with Eq. (1) and Eq. (2), we have that in step 10, z will be given in $\mathbb{Z}_Q$ by the following

$$\begin{aligned}[z]_Q &= [a + c_2 P]_Q - [b]_Q \\ &= [\alpha + c_1 g - b_2 P + c_2 P]_Q - [\alpha + c_1 g - b_1 Q]_Q \\ &= [((c_2 - b_2)P + b_1 Q) \mod Q]_Q \\ &= [c_2 P \mod Q]_Q \\ &= [c_2 P]_Q\end{aligned}$$

where $c_2 \leq n$, since $nP < Q$ (see Figure 4.4). In the above calculation, we first insert a and b as described before and then use the fact that $b_1 = b_2 = 0$ if f divides δ. To see this, remember that b_1 in Eq. (1) came from moving $c_1 g$ out of the modulus, but when $f|\delta$ both δf_Q^{-1} and $c_1 g$ are smaller than $P/2^s$ (as $\delta, c_1 N < P$ and $f > 2^s$), so no overflow will occur. The same holds for b_2.

But then in step 11, y will be zero, which follows from the correctness of macro Membership.

$\underline{f \nmid \delta}$: If on the other hand f does not divide δ, z will be given by the following, where we again make use of Eq. (1) and Eq. (2)

$$\begin{aligned}[z]_Q = [(\delta f_P^{-1} &\mod P) + c_1 g - b_2 P + c_2 P]_Q \\ &- [(\delta f_Q^{-1} \mod Q) + c_1 g - b_1 Q]_Q \\ = [(\delta f_P^{-1} &\mod P) - b_2 P + c_2 P \\ &- (\delta f_Q^{-1} \mod Q) + b_1 Q + b_3 Q]_Q \\ = [(\delta f_P^{-1} &\mod P) + c_3 P - (\delta f_Q^{-1} \mod Q) + d_1 Q]_Q,\end{aligned}$$

where $c_2 - b_2 = c_3 \leq n$ and $b_1 + b_3 = d_1 \in \{0, 1, 2\}$. In the above, $b_3 \in \{0, 1\}$ is again a bit, since the entire minus operation can only trigger one underflow.
What we now need to argue, is that

$$(\delta f_P^{-1} \mod P) - (\delta f_Q^{-1} \mod Q) + d_1 Q, \tag{3}$$

is non-zero with overwhelming probability over the randomness of Q. This would namely imply that z is not a small multiple of P ($z \neq cP$ with $c \leq n$), and thus in step 11, y is non zero so `non-dividing` is output except with negligible probability.
In order to prove that the above is non-zero except with negligible probability, note first that the statement is equivalent to

$$\delta f_P^{-1} - uP = \delta f_Q^{-1} - (v + d_1)Q,$$

for some integers $u < P$ and $v < Q$. Multiplying by f on both sides

$$\delta f_P^{-1} f - ufP = \delta f_Q^{-1} f - (v + d_1)fQ.$$

Now note that by definition of f_P^{-1} and f_Q^{-1}, there exist some integers α and β such that $f_P^{-1} f = 1 + \alpha P$ and $f_Q^{-1} f = 1 + \beta Q$. Furthermore we know that $\alpha < P$ and $\beta < Q$, as multiplying the two elements can not trigger more than P or Q overflows respectively. Thus the above is equivalent to

$$\delta + (\delta\alpha - uf)P = \delta + (\delta\beta - (v + d_1)f)Q.$$

In the above, $(\delta\beta - (v+d_1)f)$ is different from 0. To see this, assume for contradiction that it is actually 0. But then δ would be a multiple of f, leading to a contradiction, since we assumed that $f \nmid \delta$. Subtracting δ on both sides of the above then implies that if the above is satisfied, Q divides $(\delta\alpha - uf)P$. This in turn means that we can bound the probability of Eq. (3) being equal to 0 by the probability of Q dividing $(\delta\alpha - uf)P$;

$$\begin{aligned}\Pr((\delta f_P^{-1} \mod P) - (\delta f_Q^{-1} \mod Q) + d_1 Q = 0)& \\ \leq \Pr(Q | (\delta\alpha - uf)P)& \\ = \Pr(Q | (\delta\alpha - uf)),&\end{aligned}$$

where we use that P is a prime. Note that $|\delta\alpha - uf| \leq P^2$ and $P < Q$. So $(\delta\alpha - uf)$ has at most 2 prime factors in the interval $]P, T[$ where Q is chosen. Hence the probability of hitting one of those, is bounded by

$$\Pr(Q | (\delta\alpha - uf)) \leq \frac{2}{2^s}.$$

Here we are reliant on the fact that Q is independent of P and of the choice of inputs to the test, and that there are at least 2^s choices for Q. As the above probability is negligible in the statistical security parameter s, we conclude that Eq. (3) is non-zero with overwhelming probability. But then, assuming Membership is correct, Divisible outputs an additive sharing of a non-zero value, concluding the proof[12].

Simulation. The simulator for this protocol is provided in Figure 5.3. In a nutshell, the simulator simply calls all the simulators of the underlying protocols to keep track of the shares of the corrupt parties. Note that no values are being opened in the protocol.

6 Efficient Biprimality Test

In Figure 6.1, we present protocol Biprime, which receives shares of $[p]_{\mathbb{Z}}$ and $[q]_{\mathbb{Z}}$ together with biprime candidate N from each party P_i. After being executed all parties learn whether or not $N = pq$ is a biprime.

[12] If we instead choose Q to be a fixed prime such that $Q > P^2$, we could conclude in the proof that Q can never divide $(\delta\alpha - uf)$, and so the test would have error probability 0. But this would require a larger Q and lead to a somewhat less efficient protocol.

FIGURE 5.3 ($\mathcal{S}_{\mathsf{Divisible}}(k, s, \mathbb{Z}_N, \mathbb{Z}_P, \mathbb{Z}_Q, [\delta]_N, [f]_{\mathbb{Z}}, \mathtt{state}_1) \to ([y]_Q, \mathtt{view}, \mathtt{state}_2)$)

This simulator takes as input domains $\mathbb{Z}_N$, $\mathbb{Z}_P$ and $\mathbb{Z}_Q$, sharings $[\delta]_N$ and $[f]_{\mathbb{Z}}$ and integers k and s, such that the the pre-conditions of Divisible are met. It also takes as input a state $\mathtt{state}_1$ and produces a new state $\mathtt{state}_2$ and a view $\mathtt{view}$ for the macro.

Simulate:

```
 1. Set $\mathtt{state}_0 = \{[\delta]_N, [f]_{\mathbb{Z}}\}$
 2. Call $([\delta_1]_P, \mathtt{view}_1, \mathtt{state}_1) \leftarrow \mathcal{S}_{\mathsf{Larger\text{-}domain}}(\mathbb{Z}_N, \mathbb{Z}_P, [\delta]_N, \mathtt{state}_0)$
 3. Call $([\delta_2]_Q, \mathtt{view}_2, \mathtt{state}_2) \leftarrow \mathcal{S}_{\mathsf{Larger\text{-}domain}}(\mathbb{Z}_N, \mathbb{Z}_Q, [\delta]_N, \mathtt{state}_1)$
 4. Call $([f]_P, \mathtt{view}_3, \mathtt{state}_3) \leftarrow \mathcal{S}_{\mathsf{Int\text{-}to\text{-}mod}}(\mathbb{Z}_P, [f]_{\mathbb{Z}}, \mathtt{state}_2)$
 5. Call $([f]_Q, \mathtt{view}_4, \mathtt{state}_4) \leftarrow \mathcal{S}_{\mathsf{Int\text{-}to\text{-}mod}}(\mathbb{Z}_Q, [f]_{\mathbb{Z}}, \mathtt{state}_3)$
 6. Call $([f^{-1}]_P, \mathtt{view}_5, \mathtt{state}_5) \leftarrow \mathcal{S}_{\mathsf{Invert}}(\mathbb{Z}_P, [f]_P, \mathtt{state}_4)$
 7. Call $([f^{-1}]_Q, \mathtt{view}_6, \mathtt{state}_6) \leftarrow \mathcal{S}_{\mathsf{Invert}}(\mathbb{Z}_Q, [f]_Q, \mathtt{state}_5)$
 8. Call $([a]_P, \mathtt{view}_7, \mathtt{state}_7) \leftarrow \mathcal{S}_{\mathsf{Mult}}(\mathbb{Z}_P, [\delta_1]_P, [f^{-1}]_P, \mathtt{state}_6)$
 9. Call $([b]_Q, \mathtt{view}_8, \mathtt{state}_8) \leftarrow \mathcal{S}_{\mathsf{Mult}}(\mathbb{Z}_Q, [\delta_2]_Q, [f^{-1}]_Q, \mathtt{state}_7)$
10. Call $([a_1]_Q, \mathtt{view}_9, \mathtt{state}_9) \leftarrow \mathcal{S}_{\mathsf{Larger\text{-}domain}}(\mathbb{Z}_P, \mathbb{Z}_Q, [a]_P, \mathtt{state}_8)$
11. Compute $[z]_Q = (a_1^{(1)} - b^{(1)}, \ldots, a_1^{(n)} - b^{(n)})$
12. Compute $[z_1]_Q = ([z]_Q \cdot P^{-1}) + 1$
13. Set $\mathtt{state}_9 = \mathtt{state}_9 \cup \{[z]_Q, [z_1]_Q\}$
14. Call $([y]_Q, \mathtt{view}_{10}, \mathtt{state}_{10}) \leftarrow \mathcal{S}_{\mathsf{Membership}}(\mathbb{Z}_Q, \{1, \ldots, n\}, [z_1]_Q, \mathtt{state}_9)$
15. Set $\mathtt{view} = \bigcup_{i=1}^{10} \mathtt{view}_i$
16. Output $([y]_Q, \mathtt{view}, \mathtt{state}_{10})$
```

Simulator for the Divisible protocol

On the way, the protocol makes use of all the macros described in Section 4 and the proposed divisibility test from Section 5. As described in the introduction, the main idea of the protocol is to run the [DM10] version of the Miller-Rabin test on secret-shared inputs. The first step is to compute $\gamma = v^{\frac{f-1}{2}} \mod N$ with $f \in \{p, q\}$ and random v. (In the semi-honest model we can ask an arbitrary party to sample v). Since we have secret sharings of p, q over the integers, we can ask the parties to locally raise v to their share of p, q to perform the step above (as the order of the group $\mathbb{Z}_N^*$ is unknown this would not have been possible with modular secret sharings). Thus, the parties now hold *multiplicative* shares of the result and we therefore use one of our macros to convert the result back to additive sharing. We then test whether $f \in \{p, q\}$ divides $\gamma \pm 1$ using the divisibility test from the previous section. Note that we are only interested in security if N is a biprime (if it is not we can leak and throw away p, q anyway), thus we can assume that the result of one of the two divisibility tests will be 0. Still, we cannot leak which of the two divisibility tests succeeded, as this would leak a bit of the candidate prime. This is solved by computing the products of the two results and then revealing the outcome, which will always be 0 when N is biprime.

Input assumptions. In order for protocol Biprime to work, we need to make some assumptions regarding its parameters and input. Given $[p]_{\mathbb{Z}}$, $[q]_{\mathbb{Z}}$, N, n, k, security parameter s and primes P and Q, we assume the following;

1. $N = pq$
2. $p, q \equiv 3 \mod 4$
3. $p, q \in [2^{k-1}, 2^k]$ and $N \in [2^{2k-2}, 2^{2k}]$
4. $p^{(1)}$ and $q^{(1)}$ are odd while all other shares are even
5. $n^2 2^{2k} < nP < Q$, i.e. $n^2 N < nP < Q$
6. Q was sampled at random, independent of P

Of the above, the first three assumptions are necessary to enable the usage of the primality test from Section 3.2. Assuming that $p^{(1)}$ and $q^{(1)}$ are odd and all other shares are even, makes us able to compute a multiplicative sharing of γ in step 1b of the protocol. Note that these assumptions are without loss of

FIGURE 6.1 (Biprime$(k, s, \mathbb{Z}_P, \mathbb{Z}_Q, N, [p]_\mathbb{Z}, [q]_\mathbb{Z}) \to b$)

This protocol is parameterized by the length of the primes, k and a security parameter s such that $k > 2s$. It is also parameterized by domains $\mathbb{Z}_P$ and $\mathbb{Z}_Q$ such that $n^2 2^{2k} < nP < Q$, P and Q are primes and Q is chosen at random independent of P. It takes as input shares $[p]_\mathbb{Z}, [q]_\mathbb{Z}$ and integer N satisfying the conditions specified in **Pre** below. The protocol assumes access to Mult$(\mathbb{Z}_Q)$, Mult-to-add$(\mathbb{Z}_N)$ and Divisible$(k, s, \mathbb{Z}_N, \mathbb{Z}_P, \mathbb{Z}_Q)$.

Pre: All parties hold additive shares of $[p]_\mathbb{Z}$ and $[q]_\mathbb{Z}$ and an integer N, such that
- $N = pq$
- $p, q \equiv 3 \mod 4$
- $p, q \in [2^{k-1}, 2^k]$ and $N \in [2^{2k-2}, 2^{2k}]$
- $p^{(1)}$ and $q^{(1)}$ are odd while all other shares are even

Post: All parties output $b \in \{\texttt{biprime}, \texttt{not-biprime}\}$ indicating whether or not $N = pq$ is biprime.
Execute: Each party $P_i \in \mathcal{P}$, proceeds as follows:

1. Let $G = \emptyset$, and do the following for each $f \in \{p, q\}$:
   (a) Party P_n samples a uniformly random value $v \in \mathbb{Z}_N$ and sends it to all other parties.
   (b) Compute $\langle \gamma \rangle_N = v^{[\frac{f-1}{2}]_\mathbb{Z}} \mod N$ by doing:
   $$\gamma_1 = v^{\frac{f^{(1)}-1}{2}} \mod N, \text{ and for } i \neq 1 \ \gamma_i = v^{\frac{f^{(i)}}{2}} \mod N$$
   (c) Compute $[\gamma]_N \leftarrow$ Mult-to-add$(\mathbb{Z}_N, \langle \gamma \rangle_N)$.
   (d) Compute $[\gamma - 1]_N$ and $[\gamma + 1]_N$.
   (e) For $\delta \in \{\gamma - 1, \gamma + 1\}$ compute
   $$[y_\delta]_Q \leftarrow \text{Divisible}(k, s, \mathbb{Z}_N, \mathbb{Z}_P, \mathbb{Z}_Q, [\delta]_N, [f]_\mathbb{Z})$$
   (f) Compute $[y]_Q \leftarrow$ Mult$(\mathbb{Z}_Q, [y_{\gamma-1}]_Q, [y_{\gamma+1}]_Q)$
   (g) Reveal $y \leftarrow$ OpenAll$(\mathbb{Z}_Q, [y]_Q)$
   (h) If $y = 0$, set $G = G \cup \{f\}$
2. If $G = \{p, q\}$ output `biprime`, otherwise output `not-biprime`.

Protocol for biprimality testing secure against a passive adversary

generality: in the semi-honest model, we can just ask the parties, when sampling the random sharings of $[p]_\mathbb{Z}$ and $[q]_\mathbb{Z}$, to do so in a way to satisfy the conditions. The last two assumptions on the auxiliary primes P, Q are needed to make the divisibility test go through (see Section 5).

Correctness. The correctness of protocol Biprime follows directly from the modified version of the Miller-Rabin test as described in Section 3.2, assuming the conditions specified in **Pre** are satisfied.

In particular, we let party P_1 compute $\gamma_1 = v^{(f^{(1)}-1)/2} \mod N$ and every other party compute $\gamma_i = v^{f^{(i)}/2} \mod N$. This leads to the correct result thanks to the assumption, that party P_1 has an odd share of f while all other shares are even, thus

$$\begin{aligned}\prod_{i \in [n]} \gamma_i \mod N &= v^{(f^{(1)}-1)/2} \prod_{i=2}^{n} v^{f^{(i)}/2} \mod N \\ &= v^{(f^{(1)}-1)/2 + \sum_{i=2}^{n} f^{(i)}/2} \mod N \\ &= v^{(f-1)/2} \mod N,\end{aligned}$$

so the outcome is a multiplicative sharing $\langle \gamma \rangle_N$ of the value $\gamma = v^{(f-1)/2} \mod N$, which is then turned into an additive sharing $[\gamma]_N$ using macro Mult-to-add$(\mathbb{Z}_N)$.

In step 1e, the parties then execute the divisibility test introduced in Section 5, twice per prime-candidate. To hide which one of the two divisibility tests was successful if f is a prime, the protocol then multiplies the two outputs of Divisible and opens this in step 1g. If f is a prime, this product will result in a sharing of 0. If on the other hand f is not a prime, Divisible will output non-zero values, so the product is non-zero with overwhelming probability. As we have already argued for correctness of the divisibility test in Lemma 5.2, the parties end up with the knowledge about whether or not

$$(p|\gamma_p - 1 \vee p|\gamma_p + 1) \wedge (q|\gamma_q - 1 \vee q|\gamma_q + 1),$$

where γ_p and γ_q are the γ's stemming from the divisibility test execution for p and q respectively, meaning that the protocol outputs the right value to all parties at the end in step 2.

Simulator. The simulator for the protocol Biprime is provided in $\mathcal{S}_{\mathsf{Biprime}}$ (Figure 6.3). This simulator uses all the sub simulators for the subprotocols in a modular fashion, and produces a view indistinguishable from the view in the actual protocol. Remember that by convention party P_1 is the only one that is honest, so the simulator gets N and parties' $\mathcal{P}\backslash\{P_1\}$ shares of $[p]_{\mathbb{Z}}$ and $[q]_{\mathbb{Z}}$ as input, and is then to construct the view of those parties. The simulator as well gets a bit $b \in \{\texttt{biprime}, \texttt{not-biprime}\}$ as input, indicating whether or not N is bi-prime.

If $b = \texttt{not-biprime}$, the ideal functionality leaks p and q, meaning that in this case, we can just perfectly simulate by running the protocol as the honest parties would do.

For the case where $b = \texttt{biprime}$, note that, along the way, we have already argued for the correctness of all the sub-protocols. Together with the fact that the test we implement gives correct results with overwhelming probability, this implies that the output of the protocol matches that of the functionality $\mathcal{F}_{\mathsf{Biprime}}$, except with negligible probability. Therefore, in this case, we run $\mathcal{S}_{\mathsf{Biprime}}$ which will simulate a view of the protocol where the output is `biprime`.

It is enough to show that $\mathcal{S}_{\mathsf{Biprime}}$ produces a view with the same distribution as a real view where the output is `biprime`. For this, we need to focus on all the values which are opened during the protocol execution. In doing so, we argue separately for two facts:

1. All the values which are opened during the protocol are either random, or can efficiently be computed from the output of the protocol.
2. All the shares of the honest parties which are revealed during the OpenTo and OpenAll sub-protocols are uniformly random, with the only constraint that they determine the value that was opened, given the shares of corrupt players.

These two facts together imply that the simulated view of all the openings is distributed exactly like the real openings. Namely, $\mathcal{S}_{\mathsf{Biprime}}$ uses fact 1) to choose correctly the value to open in the simulated view, and then use fact 2) to choose the honest shares to reveal with the correct distribution.

In order to argue for 2), note that all the secret-shared values which are opened during the protocol are the direct output of some multiplication protocol Mult, and that the shares are not used by the protocol after opening. This is easy to verify by inspection of the protocol. Being the output of Mult guarantees (by assumption on the security of the Mult subprotocol), that the shares are distributed uniformly at random (under the constraint of adding up to the correct result). The fact that the opened shares are not used in any other computation in turn guarantees that no dependency is created between different opened values.

Then, we move onto arguing 1), namely that all the secret-shared values which are being declassified within the protocol are either random or can be computed from the output of the functionality. Recall that we only need to handle the case where the output is `biprime`: here the value y, which is the only value opened in the body of Biprime, is always 0 in the real protocol. This is because the output is obtained as the product of the results of the two divisibility tests, so the result is 0 as long as either of the divisibility results is 0. The simulator therefore knows it should set $y = 0$.

For all other values which are being declassified within the sub-protocol, it holds that they are simulated as uniformly random values within their domains. We argue that this is also the case in the real execution of the protocol: recall that values are only being opened within the Invert and Membership sub-protocols. In both cases it can be verified by inspection of the protocols that the opened values are non-zero values (as argued in the correctness proof) which are multiplied by a uniformly random value before being declassified, and crucially all of these multiplicative masks are used each to mask a single declassified value.

This leads to the following:

Theorem 6.2. *Assuming a secure implementation of the Mult sub-protocol, protocol Biprime securely implements the $\mathcal{F}_{\mathsf{Biprime}}$ functionality against semi-honest adversaries.*

FIGURE 6.3 ($\mathcal{S}_{\mathsf{Biprime}}(\mathbb{Z}_P, \mathbb{Z}_Q, \{p^{(2)}, \ldots, p^{(n)}\}, \{q^{(2)}, \ldots, q^{(n)}\}, N) \to (\mathtt{view}_{\mathsf{Biprime}}, \mathtt{state})$)

This simulator takes as input the corrupt parties input shared of $[p]_{\mathbb{Z}}$ and $[q]_{\mathbb{Z}}$, N. It produces a view $\mathtt{view}_{\mathsf{Biprime}}$ for Biprime, and a state $\mathtt{state}$. The simulator assumes that $N = pq$ is a biprime, as simulation becomes trivial otherwise, since p and q are leaked.

Simulate:

```
1. Set $[p]_{\mathbb{Z}} = (1, p^{(2)}, \ldots, p^{(n)})$ and $[q]_{\mathbb{Z}} = (1, q^{(2)}, \ldots, q^{(n)})$
2. Set $\mathtt{state}_0 = \{[p]_{\mathbb{Z}}, [q]_{\mathbb{Z}}, N\}$
3. For $f \in \{p, q\}$
   (a) Pick $v \in \mathbb{Z}_N$ at random
   (b) Set $\mathtt{view}_{\mathsf{Biprime}} = \mathtt{view}_{\mathsf{Biprime}} \cup \{v\}$
   (c) Calculate $\langle\gamma\rangle_N = v^{[(f-1)/2]_{\mathbb{Z}}} \mod N$ as in the protocol
   (d) Call $([\gamma]_N, \mathtt{view}_1, \mathtt{state}_1) \leftarrow \mathcal{S}_{\mathsf{Mult\text{-}to\text{-}add}}(\mathbb{Z}_N, \langle\gamma\rangle, \mathtt{state}_0)$
   (e) Set $\mathtt{view}_{\mathsf{Biprime}} = \mathtt{view}_{\mathsf{Biprime}} \cup \mathtt{view}_1$
   (f) Calculate $[\gamma \pm 1] = (\gamma^{(1)} \pm 1, \gamma^{(2)}, \ldots, \gamma^{(n)})$
   (g) For $\delta \in \{\gamma - 1, \gamma + 1\}$
       i. Call
            $([y_\delta]_Q, \mathtt{view}_2, \mathtt{state}_2) \leftarrow \mathcal{S}_{\mathsf{Divisible}}(k, s, \mathbb{Z}_N, \mathbb{Z}_P, \mathbb{Z}_Q, [\delta]_N, [f]_{\mathbb{Z}}, \mathtt{state}_1)$
       ii. Set $\mathtt{view}_{\mathsf{Biprime}} = \mathtt{view}_{\mathsf{Biprime}} \cup \mathtt{view}_2$
       iii. Set $\mathtt{state}_1 = \mathtt{state}_2$
   (h) Call
            $([y]_Q, \mathtt{view}_3, \mathtt{state}_0) \leftarrow \mathcal{S}_{\mathsf{Mult}}(\mathbb{Z}_Q, [y_{\gamma-1}]_Q, [y_{\gamma+1}]_Q, \mathtt{state}_1)$
   (i) Set $y^{(1)} = 0 - \sum_{i=1}^{n} y^{(i)}$
   (j) Set $\mathtt{view}_{\mathsf{Biprime}} = \mathtt{view}_{\mathsf{Biprime}} \cup \mathtt{view}_3 \cup \{y^{(1)}\}$
Output $(\mathtt{view}_{\mathsf{Biprime}}, \mathtt{state}_0)$
```

Simulator for protocol Biprime

FIGURE 6.4 ($\mathcal{F}_{\mathsf{Biprime}}(n, k, s)$)

This functionality is parametrized by the number of parties participating, n, the length of the input primes k and the statistical security parameter s.

Verify: Upon receiving $(\mathtt{verify}, \mathtt{ssid}, N, p_i, q_i)$ from each party i, proceed as follows:

1. Compute $p = \sum_{i \in [n]} p_i$ and $q = \sum_{i \in [n]} q_i$.
2. Compute
$$\delta = p \text{ is prime} \wedge q \text{ is prime}$$
3. If $\delta = \bot$, send $(\mathtt{not\text{-}biprime}, \mathtt{ssid})$ as delayed output to all parties. On top of that, leak p and q, by sending $(\mathtt{leak}, \mathtt{ssid}, p, q)$ to all parties.
4. If instead $\delta = \top$, send $(\mathtt{biprime}, \mathtt{ssid})$ as delayed output to all parties.

The ideal functionality for biprimality testing

6.1 How to achieve active security.

In this section we give some details on how we can achieve malicious security using the compiler from [DOS18], which was implemented and optimized in [EKO+19].

This compiler takes a protocol secure against $t^2 + t$ semi-honest corruptions and produces a protocol for the same number of players and the same purpose, tolerating t malicious corruptions. It works also for protocols in the preprocessing model, where the protocol assumes correlated randomness is given "for free". The compiler is clearly more interesting for small values of t. In particular, for $t = 1$ we get the following: Let $\pi_{3,2}$ be our protocol instantiated for $n = 3$ and security against 2 semi-honest corruptions. It can be seen as a protocol that works in the preprocessing model, given multiplication/Beaver triples for secure multiplication modulo N (and modulo an auxiliary prime). Let $C(\pi_{3,2})$ be the compiled version of that protocol. Thus, $C(\pi_{3,2})$ is a protocol for 3 players, secure against 1 malicious corruption that does the Miller-Rabin test on N, given appropriate correlated randomness. Importantly, the compiler is not based on zero-knowledge proofs, but on several players doing the same computation so they can verify each others' actions. Therefore, the overhead in going from $\pi_{3,2}$ to $C(\pi_{3,2})$ is small, roughly a factor 3.

Luckily, [DOS18,EKO+19] show how to efficiently construct a protocol $\pi_{preproc}$ that generates multiplications triples for 3 players and 1 malicious corruption, where the multiplications can be modulo any number N. Hence, running $\pi_{preproc}$ followed by $C(\pi_{3,2})$ gives a protocol for exactly the same setting as in [DM10], but without any set-up assumptions and no need for a computationally expensive commitment scheme[13].

One can also instantiate our semi-honest protocol for $n = 5$ and 2 semi-honest corruptions. In this case, the multiplications can be done using standard Shamir-sharing based methods. Running the compiler on this protocol produces a 5-player protocol secure against 1 malicious corruption. This approach has the advantage that we do not need multiplication triples generated by another protocol, which would lead to additional overhead. Instead, the overhead over the semi-honest version is only a small constant factor incurred by the compilation.

Finally, we note that the protocols mentioned so far only achieve malicious security with abort. At the cost of assuming a larger ratio of honest to corrupt players, we can get guaranteed output delivery. In particular, we can instantiate our semi-honest protocol for $n = 4$ tolerating 3 semi-honest corruptions assuming correlated randomness and generate this randomness using a generalized version of $\pi_{preproc}$ for 4 players and 1 malicious corruption (that always terminates). By the result in [DOS18], the compiled version of this protocol will have guaranteed output delivery.

6.2 Achieving $O(1)$ online multiplications

As already mentioned, the most computationally expensive operations in Biprime are the $O(n)$ multiplications required within the subprotocols Mult-to-add and Divisible (or rather the protocol Membership used within Divisible). While this is of course not an issue for a small number of parties, it represents a bottleneck for large n.

While we do not know how to get rid of this linear dependency (while keeping our overall good performances in practice), we present in this section a variant of the protocol that pushes the $O(n)$ multiplications to a preprocessing phase (before $N, [p]_{\mathbb{Z}}, [q]_{\mathbb{Z}}$ are available) and only requires performing $O(1)$ multiplication in the online phase (after the inputs are known).

First, consider the Mult-to-add$(\mathbb{Z}_A, \langle x \rangle_A)$ protocol: assuming the parties have access to a pre-processed pair $([r]_A, \langle r \rangle_A)$, the protocol could be modified to let the parties compute and open $\langle y \rangle_A = \langle r \rangle_A \cdot \langle x \rangle_A$ and then locally compute $[x]_A = [r]_A \cdot y^{-1}$. This requires in fact no multiplications, but only a single opening and local operations. Unfortunately this does not quite work for us: the issue is that within the protocol Biprime, the protocol Mult-to-add is called for the domain $\mathbb{Z}_N$, but since N is not known in the preprocessing phase, the pair $([r]_N, \langle r \rangle_N)$ cannot be pre-computed. Instead, we will preprocess a pair $([r]_{\mathbb{Z}}, \langle r \rangle_{\mathbb{Z}})$, aka a pair of a

[13] Note that we are not going to run $\pi_{preproc}$ in an off-line setting before the online protocol is executed. This would not make sense as we do not know N until it is generated by the compiled protocol. Instead, we would run $\pi_{preproc}$ on the fly as needed.

multiplicative and additive sharing of a large enough random value r. In the online phase then the parties can convert $([r]_{\mathbb{Z}}, \langle r \rangle_{\mathbb{Z}})$ into $([r]_N, \langle r \rangle_N)$ by taking their shares modulo N. As long as r was chosen in such a way that the share of each party is large enough so that $r_i \mod N$ is uniformly random modulo N, then this will allow to run the secure Mult-to-add described above. Note that the correlation $([r]_{\mathbb{Z}}, \langle r \rangle_{\mathbb{Z}})$ can be efficiently generated offline by having each party choose a random r_i (their multiplicative share) uniformly at random in a large enough domain. Then they can secret share this (modulo some larger B which allows to simulate integer computation) and compute, using n multiplications in $\mathbb{Z}_B$, an additive sharing $[r]_{\mathbb{Z}}$.

Now, we turn our attention to reducing the number of online multiplications in the Divisible protocol. We show how to do this at the cost of rejecting 3/4 valid candidates of N (in order to avoid leakage). Recall that in Step 10 of the Divisible protocol the parties end up with a value $[z]_Q$ such that $z = cP$ (for some constant $c < n$) if the divisibility test is successful, and something bigger but unspecified otherwise. Since the value c might leak sensitive information, we check whether z is of the form cP essentially by "brute force" thanks to the Membership protocol, exploiting the fact that $c < n$. The idea towards reducing the number of online multiplications is to avoid using the membership testing altogether, and instead compute and open the value

$$[u]_Q = [z]_Q + P \cdot [t]_Q,$$

for a large enough random t, which can be used to statistically hide c. Note at the same time that t cannot be too large, as we do not want to have an overflow modulo Q. Therefore, we preprocess a value $[t]_Q$ uniformly random constrained to the range $0 < t < Q/P - n$. .

Note now that the parties can easily check (in cleartext) whether u is a multiple of P and reject otherwise.

If on the other hand f does not divide δ, opening $[u]_Q$ introduces some unspecified leakage. In our regular protocol this is prevented by multiplying the result of the other divisibility test (which is guaranteed to be 0 when N is biprime) thus fully hiding u. This is not possible now, and therefore we must instead simply discard N if the first divisibility test with $\gamma + 1$ fails, thus preventing us from checking if the test would have been succesfull with $\gamma - 1$. This implies that we are only successful if by chance the divisibility tests for both primes are succesfull in their first attempt which, over the randomness of picking v in the Miller-Rabin test, happens with probability 1/4.

Finally, note that in our regular protocol we hide whether the divisibility test is succesfull for $\gamma + 1$ or $\gamma - 1$ by multiplying the two results at the end. Clearly this is not possible anymore, as the result is learned by the party in plaintext. We solve this by pre-processing an integer secret sharing of a bit over the integers $[b]_{\mathbb{Z}}$ which, during the online phase, is reduced to $[b]_N$ and is used to compute $[\gamma + 2b - 1]_N$. Thus, we run the divisibility test without knowing whether we are testing the divisibility of $\gamma + 1$ or $\gamma - 1$.

7 Experimental Evaluation

We compare our protocol against the one by Boneh-Franklin [BF97], and to a lesser extent the one by Damgård-Mikkelsen [DM10]. We observe that the Boneh-Franklin protocol only requires a *single* multiplication over a domain $2^{3k+s+2\log(n)}$ using the approach of Frederiksen *et al.* [FLOP18], generalized to more than two parties. For completeness we include this protocol in Figure 7.1. Similarly we observe that the Damgård-Mikkelsen protocol (which the authors present only for 3 parties) requires 42 multiplications, of which one is over a domain of size 2^{6k+6} and the rest over a domain of size at most 2^{2k+3}.

Code design. We benchmarked an implementation of our (semi-honest) protocol against the semi-honest Boneh-Franklin protocol for several different choices of modulus size, security parameters and amount of parties with the goal of comparing "apples-to-apples". We did not implement the Damgård-Mikkelsen protocol, since it closely follows the structure and has the same amount of rounds as ours, and the same amount of exponentiations, but requires 48% more multiplications than our protocol! Thus we believe it is self-evident that we will outperform this protocol. Furthermore, this protocol is only specified in the honest majority, 3-party setting, and thus is less general than both ours and the Boneh-Franklin protocol. Although it can be generalized to more parties, at a significant increase in communication complexity and domain size in one of its multiplications.

FIGURE 7.1 (**Biprime**$(k, s, N, [p]_{\mathbb{Z}}, [q]_{\mathbb{Z}}) \to b$)

This protocol is parameterized by the length of the primes, k and a security parameter s such that $k > 2s$. It takes as input shares $[p]_{\mathbb{Z}}, [q]_{\mathbb{Z}}$ and integer N satisfying the conditions specified in **Pre** below. The protocol assumes access to Mult$(\cdot)$, Random-sample$(\mathbb{Z}_N)$ and Int-to-mod$(\mathbb{Z}_N, [x]_{\mathbb{Z}})$.

Pre: All parties hold additive shares of $[p]_{\mathbb{Z}}$ and $[q]_{\mathbb{Z}}$ and an integer N, such that
- $N = pq$
- $p, q \equiv 3 \mod 4$
- $p, q \in [2^{k-1}, 2^k]$ and $N \in [2^{2k-2}, 2^{2k}]$
- $p^{(1)}$ and $q^{(1)}$ are odd while all other shares are even

Post: All parties output $b \in \{\texttt{biprime}, \texttt{not-biprime}\}$ indicating whether or not $N = pq$ is biprime.
Execute:
1. The parties in $\mathcal{P}$ execute the following test s times.
   (a) P_1 samples a random value $\gamma \in \mathbb{Z}_N^{\times}$ with Jacobi symbol 1 over N and sends this to all other parties.
   (b) P_1 computes $\gamma_1 = \gamma^{\frac{N+1-p^{(1)}-q^{(1)}}{4}} \mod N$ and sends this to all other parties.
   (c) P_i for $i \neq 1$ computes $\gamma_i = \gamma^{\frac{-p^{(i)}-q^{(i)}}{4}} \mod N$ and sends this to all other parties.
   (d) All parties validate if $\prod_{i \in [n]} \gamma_i \mod N = \pm 1$. If this is not the case, then output `not-biprime`.
2. The parties verify that $\gcd(N, p+q-1) = 1$ as follows:
   (a) Obtain $[r]_N \leftarrow$ Random-sample$(\mathbb{Z}_N)$ // The parties will verify that $r \cdot (p+q-1) \mod N = 1$
   (b) Convert $[p]_N \leftarrow$ Int-to-mod$(\mathbb{Z}_N, [p]_{\mathbb{Z}})$ and $[q]_N \leftarrow$ Int-to-mod$(\mathbb{Z}_N, [q]_{\mathbb{Z}})$
   (c) Compute $[\alpha]_N \leftarrow$ Mult$(\mathbb{Z}_N, [r]_N, [p]_N + [q]_N - 1)$
   (d) Reveal $\alpha \leftarrow$ OpenAll$(\mathbb{Z}_N, [\alpha]_N)$.
   (e) If $\alpha = 1$, the parties output `biprime`, otherwise they output `not-biprime`.

Boneh and Franklin's [BF97] protocol for biprimality testing secure against a passive adversary based on the optimization of Frederiksen *et al.* [FLOP18], generalized to multiple parties.

We stress that our goal has *not* been to make as efficient an implementation as possible, but rather to make a proof-of-concept implementation.[14] For this reason we have prioritized architecture and readability over optimizations. We did the implementation in Java and used the standard library `BigInteger` for big integer arithmetic and (deterministic, yet seeded) `SecureRandom` for randomness generation, but optimize computation of exponentiations in large domains by *optional* usage of the C-based GNU Multiple Precision arithmetic library (GMP) through the Java Native Interface (JNI) by using a wrapper [Des]. All aspects of our implementation use a *single* thread, and hence should scale directly in throughput linearly in the amount of cores applied. This approach has the advantage of cross-platform portability and ease of usage and readability when used without the JNI, but still allows more efficient runtimes when taking advantage of the native implementation for the computationally heavy aspects of ours and Boneh-Franklin's protocols, i.e. modular exponentiations. We discuss the advantage of the JNI in more detail below, but here highlight that all benchmarks unless otherwise stated, are using the JNI optimization.

We took a micro-benchmark approach to our implementation, letting it consist of several interchangeable components. We did so, to both allow us to find the optimal setup of our protocol (and the one by Boneh-Franklin), but also to allow us to give more detail, and to isolate bottlenecks, in our benchmarks. Concretely we isolate exactly the amount of bytes to be sent over the network interface, rounds and the amount of multiplications required. We then microbenchmark the core of the protocols, *excluding any* communication and multiplications. We achieve this by using dummy modules for multiplication and the network interface. This allowed us to run our benchmark on a single machine and find the formula for the expected latency under *any* concrete network setup (in latency and bandwidth) and multiplication protocol. However, because of this, our latency benchmarks are computed based on raw execution time, communication and round complexity.

[14] The implementation is openly available at `https://github.com/jot2re/rsa`

Hence we note that they *exclude* any TLS computational TCP overhead that might occur in practice (other than a 40 kb header needed for each TCP package).

All our tests were executed on an AWS t3.micro instance with 2 virtual cores on an Intel Xeon 8000 series given a baseline of 10% performance and 1 GB of RAM, running the latest version of Ubuntu and openJDK 19. We picked this system due to its pervasive usage and very competitive price point (something we believe is high important for a protocol like ours), although it is not the fastest nor most performance consistent due to the CPU cores being shared. Throughout the benchmarks we amortize communication s.t. each party will send and receive an equal amount. All communication complexity numbers are *only* for sending *or* receiving. I.e. the total network I/O is the double of the included numbers, but all modern networks are full duplex and hence it will not affect throughput in practice. All benchmarks with networking estimates are based on a 1 gigabit connection with 10 ms latency, which reflects two servers connected in the same region, but *not* the same LAN or building. We believe this reflects the desired use-case where multiple companies host servers in the same region to collaborate in key generation. All our tests (except the microbenchmark of the Gilboa multiplication and the malicious protocol) were run using the Java Microbenchmark Harness (JMH) with parameters to expect at least 10 warm-up iterations and at least 10 iterations (each consisting of multiple samples) for the actual benchmarks. The tests were furthermore executed with the parameters `-Xms512m -Xmx512m` to ensure sufficient dedicated memory for the test from the beginning. I.e. it is sufficient to avoid hitting a memory cap. Concretely we found the maximum memory used by both our protocol, and the Boneh-Franklin protocol during the JMH benchmarks being 325 MB. Due to the microbenchmark nature of our tests, some presented results are a combination of different microbenchmarks and hence not include an error margin. However we note that in general the largest standard deviation was found on the micorbenchmarks of the Shamir multiplication protocol as seen in Tab. 3. We found significantly lower standard deviation on the actual protocol executions, with our protocol showing the largest instability as seen in Fig. 2.

Multiplication We implemented 3 different multiplication protocols, each of which works for different types of secret-sharing schemes:

Shamir secret-sharing using the simplified multiplication protocol of Gennaro *et al.* [GRR98]. Constants depending on the amount of parties are precomputed for 2, 3, 5, 7, 9, and 11 parties. Furthermore, we optimize the inner workings of Shamir-based multiplications using `long` instead of `BigInteger` to avoid unnecessary overhead during polynomial manipulation. Note that while Shamir secret-sharing requires a field to work, we also need to perform sharing modulo the RSA candidate N. This fails (except with negligible probability) only if N is not a biprime, in which case we can just reject it anyway.

Replicated secret-sharing for 1-out-of-3 parties based on additive shares [ISN89].

Gilboa/OT-based multiplication was carried out over additive secret-shares using the protocol of Gilboa [Gil99]. Our benchmarks *only* include the *online* time and thus assume that a sufficient amount of random OTs have been constructed previously, which can be done highly efficiently using OT-extensions e.g. [KOS15,BCG$^+$19,Roy22]. In addition, we also implemented the approach of Ishai *et al.* [IPS09]. In relation to communication complexity, we unsurprisingly found it inferior to the approach of Gilboa. Furthermore we also found it inferior in computation time in *all* situations required by both protocol and the Boneh-Franklin protocol.

Shamir secret-sharing and replicated secret-sharing necessarily only give security in the honest majority setting, whereas the OT-based approach works in the dishonest majority setting. However, Shamir and replicated secret-sharing have the disadvantage of requiring conversions to and from additive secret-sharings before and after any series of multiplications. Turning additive shares into Shamir or replicated shares require a round of communication. Although turning Shamir or replicated shares into additive shares can be done non-interactively. For Shamir this is done by having parties with id lower than $t+1$ take their share and multiply it with the relevant Lagrange coefficient. Parties with larger id, then set their share to be 0. For replicated shares, simply taking the least significant share of the set of shares yield an additive share. We also note that communication complexity for Shamir and replicated multiplication could be improved further by having each party share a seed with each t other parties, which can be used to derive random points when needed during sharing and multiplication.

bits	Replicated		Shamir		Gilboa	
	ms.	KiB	ms.	KiB	ms.	KiB
2052	9.60 ± 0.70	101	1.48 ± 0.55	51.2	11,800	26,800
3096	16.1 ± 1.7	152	2.33 ± 0.98	76.9	19,800	59,900
5000	25.0 ± 1.6	202	3.10 ± 1.7	102	30,400	106,000

Table 3: Communication and computation complexity for **100** multiplications operations for 3 parties with standard deviation.

bits	Replicated		Shamir	
	ms.	KiB	ms.	KiB
2052	1.60 ± 0.24	101	1.34 ± 0.60	51.2
3096	2.40 ± 0.34	152	2.15 ± 1.3	76.9
5000	3.60 ± 0.40	202	2.43 ± 1.2	102

Table 4: Communication and computation complexity for **100** additive sharing operations for 3 parties.

We performed microbenchmarks of these protocols and outline the findings in Tables 3 and 4. Concretely we did benchmarks for domains slightly larger than the bits of the modulus N we wish to validate biprimality of. This is because our protocol requires us to do multiplication both modulo N, P and Q with $N < P < Q$. Thus the microbenchmark can be considered upper bounds on *any* of the multiplications needed in our, or the Boneh-Franklin, protocol.

Unsurprisingly we observe that the computation scales almost perfectly in $n-1$ for the OT-based protocol and Shamir protocols, as does the communication. While we only implemented replicated secret sharing for 3 parties, we note that the communication complexity, and share-size scales in $\binom{n}{t}$. Concretely we find that despite the disadvantages of Shamir and replicated sharing, they perform orders of magnitude better than the OT-based approach, with Shamir always performing best in our situations, both in relation to computation and communication.

Number of Rounds. Multiple places in our protocol we need to multiply a series of $O(n)$ secret shared values. As described earlier one way of doing so is using a tree-structure (leading to $\log n$ rounds), and one is based on the Bar-Ilan Beaver [BIB89] trick (leading to constant rounds but at the price of a linear increase in multiplication gates).

We implemented both approaches and express the impact on multiplication gates and rounds in Tables 5 and 6. These tables also take into account how the choice of multiplication protocol might impact on our and the Boneh-Franklin protocol. More specifically some multiplication protocols work over *additive* shares, similar to both our protocol and the Boneh-Franklin protocol. This typically includes OT based [Gil99,IPS09] or additive homomorphic protocols [CDN01,DN03] [15]. However other multiplication protocols rely on having the inputs being shared using *other* types of linear sharing. This include Shamir based protocols [GMW87,GRR98] or replicated secret sharing [ISN89]. This incurs some extra computation and an extra round of communication for each sequence of multiplications when coming from additive shares. The total amount of times a single value must be converted is captured in the column *From Add.* in Table 6 and the round overhead (assuming batched conversion) is captured in the rows *Our (other)* and *[BF97] (other)* in table 5. On the other hand Shamir and replicated secret sharing based protocols allow a multiplication followed by an opening to be done in a single round[16]. Furthermore, both additive and replicated sharing can afford sampling of random values non-interactively through a common seed.[17]. Finally we observe that due to the modular presentation of our protocol, a few operations are redundant and in fact

[15] These are typically based on Paillier encryption, however for us to use such schemes we would require it to work over an arbitrary domain that is not a prime (i.e. the candidate public key N). Hence such approaches will likely be inefficient as the only additive homomorphic schemes we are aware of that work in this setting are class groups [CL15] which are known to be quite inefficient in practice in setup for each new modulus [BCIL22, Tab. 3] which we would require to happen for each biprimality test due to the changing modulus N.

[16] For simplicity, and to follow typical implementation approaches we do however count this as 2 rounds.

[17] For simplicity, we don't reduce the rounds on this account, but note that a round can be removed from the membership test if this optimization is applied.

only need to be done once. For instance, the Divisible protocol in Fig. 5.1 gets called twice on both $[p]_{\mathbb{Z}}$ and $[q]_{\mathbb{Z}}$, but the inversions in Steps 5 and 6 only need to be performed once.

We found that for all our test settings the logarithmic approach would always be the most efficient. Hence this is the approach we used for our benchmarks.

Table 5: Round complexity of our protocol, the Damgård-Mikkelsen [DM10] and the Boneh-Franklin [BF97] protocol depending on whether *additive* or an *other* linear secret sharing scheme is used, i.e. Shamir or replicated secret sharing.

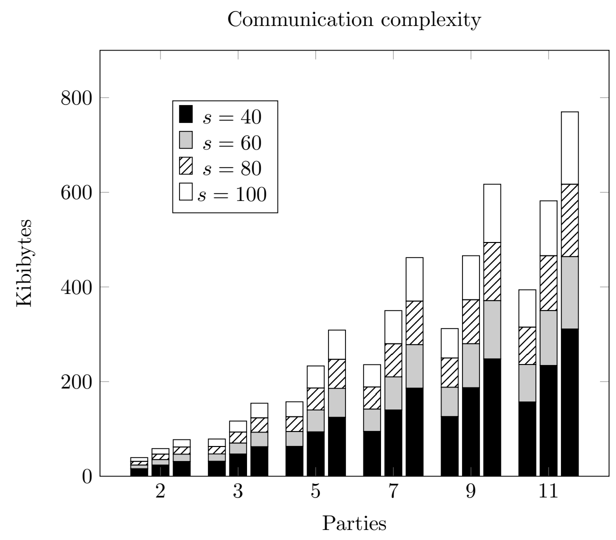
Parties	2	3	5&7	> 8
[BF97] (additive)	2	2	2	2
[BF97] (other)	3	3	3	3
[DM10]	-	10	-	-
Our (additive)	5	7	9	11
Our (other)	8	10	12	14

Table 6: Multiplicative complexity of our protocol, depending on the approach taken to handle multiplications.

Rounds	Parties	Mults	From Add.
Log	2	21	18
	3	27	18
	5	39	18
	7	51	18
	9	64	18
	11	75	18
Const.	2	41	38
	3	67	58
	5	119	98
	7	147	114
	9	191	146
	11	235	178

Protocol benchmark We perform our benchmark on the actual, online executable protocol code in the *worst case*, meaning that a protocol will continue until the end regardless of whether the candidate N is a biprime or not. In practice, both our protocol and the Boneh-Franklin protocol may output `not-prime` without executing all steps. We exclude the time it takes to load classes and setup parameters. Hence for our protocol we do *not* include the time it takes to sample P (which just needs to be a large enough prime) and Q (which needs to be independent of N) and setting up other parameters. We find this the most fair and accurate in the use-case where many biprimality tests have to be executed. We note, that once parameters have been setup, an indefinite amount of biprimality tests can be executed, as long as the shares of p and q are picked independently of the prime Q.

We present the micro benchmarks of the Boneh-Franklin protocol in Fig. 1 and our protocol in Fig. 2. These benchmarks are *raw* in the sense that they *exclude* the time needed for performing necessary multiplications, along with network overhead and hence present the base confrontational and communication requirements of the protocols. We see that our protocol performs orders of magnitude better than the one by Boneh-Franklin at comparable choices of statistical security parameters. We also observe that neither we, nor Boneh-Franklin, observe any significant increase in computation time with an increase in parties, but a significant increase in communication. That is, more parties do not significantly hurt throughput, but only latency.



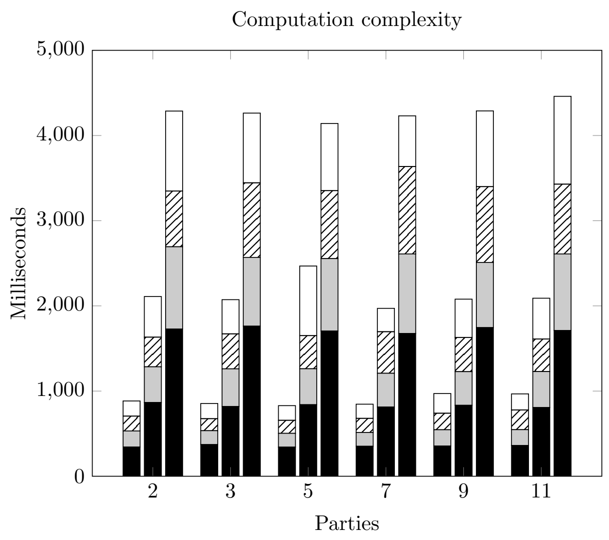


Fig. 1: Raw communication and computation of the Boneh-Franklin protocol *excluding* the single multiplication it requires. Each segment of 3 stacked bars represents, left to right, an RSA modulus of 2048, 3072, 4096 bits respectively.

Fig. 3 shows the comparison between the protocols in the honest-majority case when implemented with the optimal choice of multiplication protocol (which turned out to always be Shamir), along with realistic network overhead. It can be observed that the latency of our protocol is between 3.5x and 20x lower that of the Boneh-Franklin protocol for 3 parties and *generally* decreases the more parties involved to between 3.2x and 14x for 11 parties. Note that in our protocol the statistical security parameter follows the size of the biprime N we are generating, while in Boneh-Franklin it is an independent parameter. To make a fair comparison we vary the statistical security parameter s in Boneh-Franklin to match ours for the different sizes of N. We find that the main bottleneck in latency for the Boneh-Franklin protocol is computation time, whereas for our protocol it is network latency, due to the rounds of communication needed.

We further compare the protocols in the 3-party case both for honest majority (using replicated secret-sharing and Shamir secret-sharing) and dishonest majority (using the OT-based Gilboa protocol [Gil99]) in Table 7. Here we compare according to price, where we consider price in USD cents for 1,000,000 biprimality tests when using a t3.micro AWS instance priced at 1.04 cents per hour of usage and 2 cents per gigabyte egress [aws]. From this we see that in the honest majority our protocol greatly outperforms Boneh-Franklin both in throughput (improvement of 42x-77x) and price (improvement of 3.0x-7.5x). Alas, in the dishonest majority case our protocol gets severely punished by the amount of multiplications we require and hence Boneh-Franklin achieves a higher throughput and lower price than our protocol. In both the honest majority and dishonest majority we found *communication*, and not *computation*, being the main price-bottleneck for our protocol. For the Boneh-Franklin protocol this was only true in the dishonest majority setting.

Malicious version. While we did not make a full, working implementation of the malicious version of our protocol, we did implement the operations (without JNI) and communication needed for a malicious version, based the compiler of Damgård *et al.* [DOS18], with some of the optimizations suggested by Eerikson *et al.* [EKO$^+$20]. In particular we used their "post-processing" approach to validate semi-honest multiplications. We summarize the findings in Table 8. Unlike our semi-honest protocol we benchmark this on a t3.xlarge

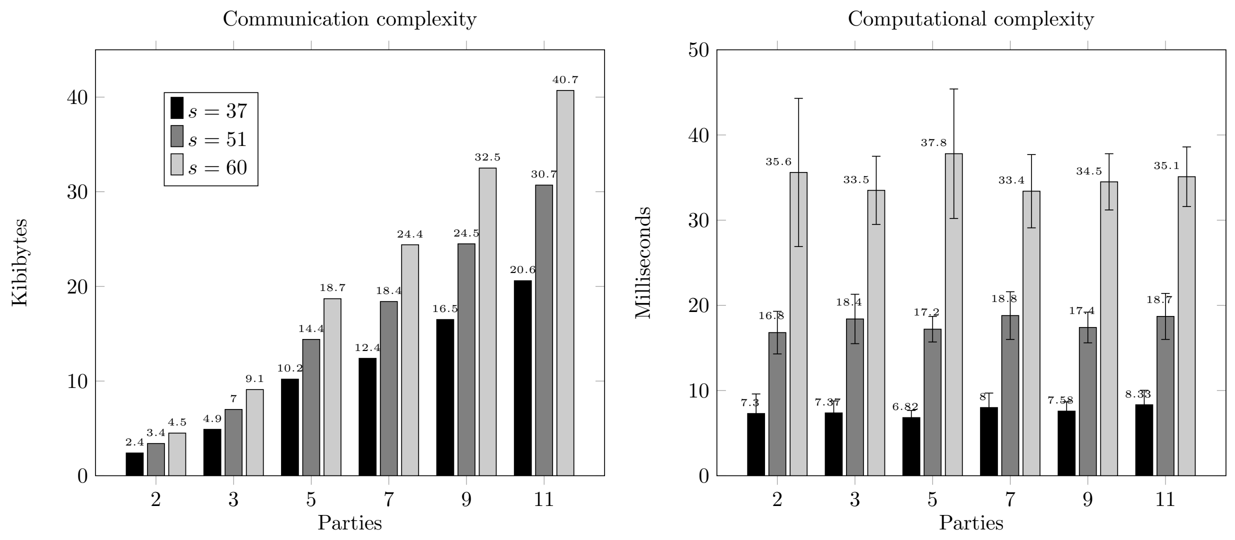


Fig. 2: Raw communication and computation of our protocol *excluding* the multiplications it requires and any network latency. Each segment of 3 stacked bars represent, left to right, an RSA modulus of 2048, 3072, 4096 bits respectively. Error-bars represent standard deviation.

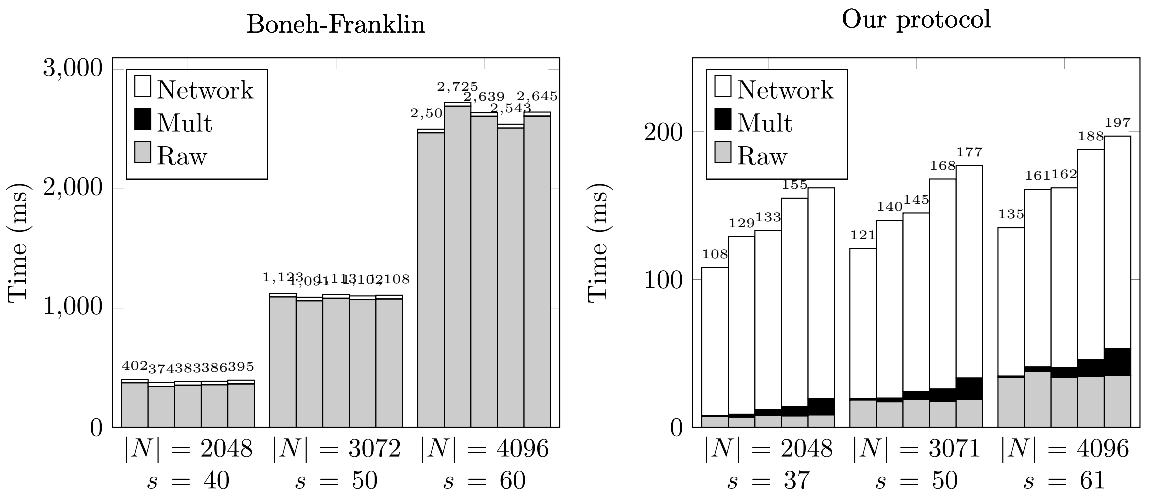


Fig. 3: Total computation time of our protocol and the Boneh-Franklin protocol for 3, 5, 7, 9 and 11 parties in the honest majority model assuming 1 Gigabit network with 10 ms latency. The left bar in a pair represents 3 parties then 5, 7, and 9 parties (using Shamir secret sharing).

$\lvert N\rvert$	Latency Ours	Latency [BF97]	Throughput Ours	Throughput [BF97]	Price Ours	Price [BF97]
			Shamir			
2048	**108**	402	**124**	2.69	**58.2**	174
3072	**121**	1,124	**52.0**	0.915	**89.3**	439
4096	**135**	2,501	**28.7**	0.405	**122**	917
			Replicated secret sharing			
2048	**111**	402	**97.6**	2.69	**104**	180
3072	**123**	1,124	**43.2**	0.915	**158**	444
4096	**142**	2,501	**24.4**	0.405	**213**	911
			Gilboa			
2048	3,440	**523**	0.312	**2.04**	15,100	**727**
3072	5,550	**1,326**	0.187	**0.775**	33,200	**1,661**
4096	8,550	**2,814**	0.122	**0.361**	58,400	**3,062**

Table 7: Computation time for 3 parties on a 1 gigabit network with 10 ms latency for different multiplication protocols. s is $37, 50, 61$ for $|N| = 2048, 3072, 4096$ respective for our protocol and $40, 50, 60$ for Boneh-Franklin.

instance which has the same CPU as t3.micro, but with access to 4 cores.[18] We find the malicious version to be around 4x slower in raw computation. Similar to the semi-honest protocol, network latency becomes the bottleneck. Concretely, the compilation introduces 4 more rounds to the overall execution time, and the post processing process introduces 2 more rounds (assuming the needed coin-tossing gets carried out in parallel with the compiled protocol).

Table 8: Benchmark of our maliciously secure protocol for 3 parties on a 1 gigabit network with 10 ms latency using the compiler of Damgård *et al.* [DOS18]. "Compil." is the raw computation time executing the compiled protocol and "Post-Pro." the time needed validate multiplications. s is $37, 50, 61$ for $|N| = 2048, 3072, 4096$. Times are in milliseconds and overhead compared with our semi-honest protocol.

	Compil.	Post-Pro.	Total comp.		Latency	
			Time	Overhead	Time	Overhead
2048	21.7	14.8	36.5	3.9x	199	1.8x
3072	70.4	22.5	92.9	4.1x	257	2.1x
4096	160	31.4	191	4.0x	356	2.4x

Pure Java vs. JNI Comparing our protocol against the Boneh-Franklin protocol it quickly becomes clear that the computational bottlenecks are in the large-integer arithmetic. We experimented with replacing these operations with JNI wrapped calls to a natively compiled GMP library. However, each call to the JNI is not free and comes with a certain amount of computational and memory overhead. We found that the only operation that did not result in worse performance when computed with JNI, was large modular exponentiations. That is, we found out that simpler operations such as addition or computing of inverses, were actually slower when executed using GMP through the JNI, due to overhead. Through experimentation we found large exponentiations to account for around 70% of the raw protocol runtime in *both* ours and the Boneh-Franklin protocols. Hence in our JNI implementation we only replaced all large exponentiations with wrapped calls to GMP.

[18] Due to an artifact of the protocol compilation, resulting in the benchmark only being possible when simulating all 3 parties. Each party has their own thread, and to get a fair comparison in per-party runtime we require an instance with at least 3 virtual CPUs of which the t3.xlarge offers.

We show the impact of the JNI vs. the pure Java implementation when running on at t3.micro instance running an x64 architecture on AWS in Table 9. From this we see that the JNI always gives a significant boost in performance for larger moduli, both for ours and the Boneh-Franklin protocols, whereas for a modulo of 2048 bits there is barely any performance difference.

As a side-note we found the impact JNI usage to be platform inconsistent; in the sense that tests run on an ARM-based M1 Macbook Pro showed significantly higher performance improvements through the JNI than the t3.micro x64-based architecture. In particular on this architecture we found a significant improvement already at biprimality testing of the smaller 2048 bits candidates.

Table 9: Raw computation time of our protocol and the Boneh-Franklin protocol with and without using the JNI to do exponentiations through GMP.

	Ours			[BF97]		
$\|N\|$	No JNI	JNI	Diff (%)	No JNI	JNI	Diff (%)
3 parties						
2048	7.03	7.37	-	362	372	-
3072	21.4	**18.4**	14.0	1,278	**1,093**	14.5
4096	39.6	**33.6**	15.1	3,260	**2,470**	24.2
7 parties						
2048	7.94	8.00	-	367	352	-
3072	21.4	**18.8**	12.1	1,265	**1,082**	14.5
4096	40.4	**33.7**	16.6	3,089	**2,610**	15.5
11 parties						
2048	7.78	8.33	-	362	384	-
3072	22.3	**18.7**	16.1	1,246	**1,075**	13.7
4096	46.5	**35.1**	24.5	3,033	**2,610**	13.9

Acknowledgment. The authors would like to thank Jesper Buus Nielsen, Benny Pinkas and Yehuda Lindell for useful research discussions and Diego Aranha for helping with tips and tricks to get the implementation running as fast as possible. We would also like to thank the anonymous CCS reviewers for useful suggestions that helped us improve the experimental evaluation. The research described in this manuscript has received funding from: the Concordium Blockhain Research Center, Aarhus University, Denmark; the Carlsberg Foundation under the Semper Ardens Research Project CF18-112 (BCM); the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC); the EU Horizon 2020 research and innovation programme under grant agreement No 786725 OLYMPUS; the Danish Independent Research Council under Grant-ID 1051-001068 (ACBD).

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 2002*, volume 2442 of *LNCS*, pages 417–432. Springer, Heidelberg, August 2002.

aws. Amazon EC2 on-demand pricing. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 2023-05-02.

BBBF18. Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 757–788. Springer, Heidelberg, August 2018.

BBF19. Dan Boneh, Benedikt Bünz, and Ben Fisch. Batching techniques for accumulators with applications to IOPs and stateless blockchains. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part I*, volume 11692 of *LNCS*, pages 561–586. Springer, Heidelberg, August 2019.

BCG$^+$19. Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

BCIL22. Cyril Bouvier, Guilhem Castagnos, Laurent Imbert, and Fabien Laguillaumie. I want to ride my BICYCL: BICYCL implements CryptographY in CLass groups. Cryptology ePrint Archive, Report 2022/1466, 2022. https://eprint.iacr.org/2022/1466.

Bd94. Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

BF97. Dan Boneh and Matthew K. Franklin. Efficient generation of shared RSA keys (extended abstract). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 425–439. Springer, Heidelberg, August 1997.

BIB89. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

Cd10. Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010.

CDK$^+$22. Megan Chen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, abhi shelat, and Ran Cohen. Multiparty generation of an RSA modulus. *Journal of Cryptology*, 35(2):12, April 2022.

CDN01. Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 280–299. Springer, Heidelberg, May 2001.

CHI$^+$21. Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. In *2021 IEEE Symposium on Security and Privacy*, pages 590–607. IEEE Computer Society Press, May 2021.

CL02. Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

CL15. Guilhem Castagnos and Fabien Laguillaumie. Linearly homomorphic encryption from DDH. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 487–505. Springer, Heidelberg, April 2015.

Des. Didier Deshommes. Gmp-java. https://github.com/dfdeshom/GMP-java. Accessed: 2023-07-27.

DLP93. Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of computation*, 61(203):177–194, 1993.

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

DMRT21. Cyprien Delpech de Saint Guilhem, Eleftheria Makri, Dragos Rotaru, and Titouan Tanguy. The return of eratosthenes: Secure generation of RSA moduli using distributed sieving. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 594–609. ACM Press, November 2021.

DN03. Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 247–264. Springer, Heidelberg, August 2003.

DOS18. Ivan Damgård, Claudio Orlandi, and Mark Simkin. Yet another compiler for active security or: Efficient MPC over arbitrary rings. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 799–829. Springer, Heidelberg, August 2018.

EKO$^+$19. Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! arithmetic 3pc for any modulus with active security. *Cryptology ePrint Archive*, 2019.

EKO$^+$20. Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 5:1–5:24. Schloss Dagstuhl, June 2020.

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 2018, Part II*, volume 10992 of *LNCS*, pages 331–361. Springer, Heidelberg, August 2018.

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

Gil99. Niv Gilboa. Two party RSA key generation. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 116–129. Springer, Heidelberg, August 1999.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

GRR98. Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, pages 101–111. ACM, June / July 1998.

HMR$^+$19. Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, Tomas Toft, and Angelo Agatino Nicolosi. Efficient RSA key generation and threshold paillier in the two-party setting. *Journal of Cryptology*, 32(2):265–323, April 2019.

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.

IPS09. Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Secure arithmetic computation with no honest majority. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 294–314. Springer, Heidelberg, March 2009.

ISN89. Mitsuru Ito, Akira Saito, and Takao Nishizeki. Secret sharing scheme realizing general access structure. *Electronics and Communications in Japan (Part III: Fundamental Electronic Science)*, 72(9):56–64, 1989.

KOS15. Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

Mil76. Gary L Miller. Riemann's hypothesis and tests for primality. *Journal of computer and system sciences*, 13(3):300–317, 1976.

MWB99. Michael Malkin, Thomas D. Wu, and Dan Boneh. Experimenting with shared generation of RSA keys. In *NDSS'99*. The Internet Society, February 1999.

Pai99. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 223–238. Springer, Heidelberg, May 1999.

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.

Rab80. Michael O Rabin. Probabilistic algorithm for testing primality. *Journal of number theory*, 12(1):128–138, 1980.

Roy22. Lawrence Roy. SoftSpokenOT: Quieter OT extension from small-field silent VOLE in the minicrypt model. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 657–687. Springer, Heidelberg, August 2022.

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

RSW96. Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

Shl20a. Omer Shlomovits. Diogenes Octopus*: Playing Red Team for Eth2.0 VDF. Medium blog post, 2020. `https://medium.com/zengo/diogenes-octopus-playing-red-team-for-eth2-0-vdf-part-1-dac3f2e3cc7b`.

Shl20b. Omer Shlomovits. DogByte Attack: Playing Red Team for Eth2.0 VDF. Medium blog post, 2020. `https://medium.com/zengo/dogbyte-attack-playing-red-team-for-eth2-0-vdf-ea2b9b2152af`.

Tof07. Tomas Toft. *Primitives and Applications for Multi-party Computation.* PhD thesis, Aarhus University, 3 2007.

Wes19. Benjamin Wesolowski. Efficient verifiable delay functions. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 379–407. Springer, Heidelberg, May 2019.