Optimizing Robustness while Generating Shared Secret Safe Primes

Emil Ong and John Kubiatowicz

University of California, Berkeley

Abstract. We develop a method for generating shared, secret, safe primes applicable to use in threshold RSA signature schemes such as the one developed by Shoup. We would like a scheme usable in practical settings, so our protocol is robust and efficient in asynchronous, hostile environments. We show that the techniques used for robustness need special care when they must be efficient. Specifically, we show optimizations that minimize the number and size of the proofs of knowledge used. We also develop optimizations based on computer arithmetic algorithms, in particular, precomputation and Montgomery modular multiplication.

Keywords: *Distributed key generation, safe primes, threshold RSA signatures.*

1 Introduction

Shoup's scheme [1] for threshold RSA signatures was a great leap forward in making threshold signature schemes practical. Its ability to avoid interaction while signing makes the scheme efficient and easy to implement. Unfortunately, Shoup's scheme required the use of a safe prime product modulus for its proof of correctness. Moreover, the scheme assumes a trusted dealer to create and distribute this modulus and the private key shares. Since the development of Shoup's scheme, several works ([2–4]) have been published to try to eliminate the single dealer, but none have shown the costs associated with a robust solution.

In this paper, we show the cost required for a robust implementation of a distributed safe prime generation scheme. We follow the basic form of the algorithm in [2], but we also show that the changes necessary for robustness are non-trivial if we want efficiency. We develop several techniques for reducing the number of proofs of knowledge while maintaining security. Our methods are based on computer arithmetic, number theory, and simple protocol analysis to reduce redundancy.

1.1 Algorithm Overview

Before diving into the details of our safe prime generation algorithm, we will give a high-level overview. Our approach to prime finding is very familiar: effectively we generate candidate numbers and test them until we find a safe prime. First we use the usual techniques for improving our search – we make sure that our

1. Find a candidate number ϕ which has no small prime factors and has the property $\phi \equiv 3 \mod 4$.
2. If the number 2 is a Miller-Rabin witness to the compositeness of ϕ or $\frac{\phi-1}{2}$, return to step 1.
3. Run the Miller-Rabin test repeatedly on ϕ with random inputs a sufficient number of times to ensure primality with a small error probability.

Fig. 1: Algorithm Overview

candidate prime is not composed of small prime factors. However instead of doing trial division, we produce our candidate in a constructive way following the lead of Malkin, Wu, and Boneh [5]. We modify their algorithm however by making it robust through contributing several zero-knowledge proofs. This method is detailed in Section 3.

After finding such a candidate, we then proceed to do more rigorous tests. Specifically, we follow the techniques outlined by Cramer and Shoup in [6]. The procedure recommended in that work involves two specialized Miller-Rabin tests followed by a generic Miller-Rabin test of compositeness. To do Miller-Rabin compositeness tests, we have to perform modular exponentiation. In our case the modulus is secret, a fact which is the main source of difficulty in our algorithm. Sections 4.1 and 4.2 are dedicated to optimizing the performance of this type of modular exponentiation. Specifically, we generalized the modular exponentiation method given in [2] and provided a new modular multiplication algorithm based on Montgomery multiplication. The high-level algorithm is summarized in Figure 1.

1.2 Application: RSA Signatures

After successfully generating two shared, safe primes with this algorithm, the players can simply multiply their shares of these primes together and reveal the result. The factorization of the composite number is not revealed because the VSS and multiplication schemes conceal. At this point, the players have all generated a public RSA modulus for which no player knows the factors. Moreover, the players can compute secret shares of the Euler totient function of the modulus. This fact allows them to use the algorithm of Catalano et al. [7] to compute secret shares of an RSA private key. These key shares are then immediately usable for Shoup's RSA signature scheme [1].

1.3 Related Work

Shoup's proof of correctness required the use of safe primes in the RSA modulus (i.e. $n = pq$ where p, q are primes of the form $p = 2p' + 1, q = 2q' + 1$ with p', q' also being primes). [3, 4] noted that this requirement is a bit strong and replaced it with assumptions relating to the computational difficulty of certain operations in RSA groups. Safe primes meet and exceed the requirements set by [3, 4] and these works both showed RSA moduli with lesser constraints are suitable for Shoup's scheme. The assumptions made are non-standard, but reasonable.

Works by Boneh et al. [8, 5] developed ways to generate and verify RSA moduli and inverses, but do not necessarily produce primes suitable to Shoup's threshold scheme. Moreover, these schemes are secure only in the honest-but-curious setting. An optimization in [5], called *distributed sieving*, however is very useful and we will develop a robust version in section 3.

Frankel, MacKenzie, and Yung [9] developed a robust method for RSA key generation, but also do not produce safe prime product moduli. Many of their techniques will be very useful in our protocol, however.

Algesheimer, Camenish, and Shoup [2] were the first to suggest an algorithm for distributively generating safe primes and we follow their exposition closely. Our work expands on their algorithm by making it robust and optimizing within this robust framework.

1.4 Contributions

Our contributions to this field are three-fold:

– We provide a robust version of the Malkin, Wu, and Boneh [5] distributed sieving algorithm,
– We improve the Miller-Rabin algorithm of Algesheimer, Camenish, and Shoup [2] by (1) generalizing the modular exponentiation method and (2) introducing Montgomery multiplication into a distributed computational framework for faster modular arithmetic.

2 Preliminaries: Model and Commitments

We deal with two preliminaries before proceeding to our algorithm, the computational and network models and the commitment and verified secret sharing schemes we use.

2.1 Model

We assume an asynchronous network only offering point-to-point messages. We view the network as an adversary that can choose to drop or delay the messages sent between two parties. The protocols we use require authenticated messages however, so we will assume that there exists some way of ensuring the integrity of messages which are delivered. We rely on the work of Goldwasser and Lindell [10] which provides a broadcast protocol which is simpler than full, authenticated Byzantine agreement, but is sufficient for both serial and parallel composition of secure computation.

For the secrecy and binding of our commitment and secret sharing protocols, we rely on the assumption that computing discrete logarithms is difficult. We will build our protocols to be secure in the random oracle model since we intend them to be used for Shoup's RSA signature scheme [1], which uses a random oracle for non-interactivity. This assumption can be removed by reintroducing additional interactivity, though at significant cost, as usual.

In describing these multiparty protocols, we will borrow the notation of [2] for secret sharing. We assume familiarity with both additive and polynomial secret sharing (also known as Shamir secret sharing [11]). Our algorithms will only involve polynomial secret sharing and we shall denote player j's polynomial share of a as $[a]_j^p \in \mathbb{Z}_p$. In general, we will use these notations to show the format of the input and output values of our multiparty protocols, but refer to the shared value directly in the body of the protocol for clarity.

2.2 Commitments

We will need a commitment scheme where the properties of the *integers* hold because we may need to deal with negative numbers to prove relative primality. We will also need a verifiable secret sharing scheme that works using integer commitments. This property will give us the ability to prove statements about the numbers we share. These two primitives are the basis for robustness in our algorithm.

We will use a scheme discussed in [7] which uses a prime finite field of very large order and relies on the discrete logarithm problem. In truth, these are not commitments over the integers, but the finite field on which they are defined is large enough that relations we are trying to prove also hold in these fields. We prefer this technique over that of [12] in our case. A more complete explanation of the differences is available in the extended version of this paper.[1]

Setup The prime number that we use needs to be larger than any term we will use in our commitments and computations. Since our goal is to create an RSA modulus by multiplying shared primes, the players can agree to a specific RSA key size *a priori* and this size will determine the maximum size of our committed numbers. For example, if we are trying to generate a key of size 1024, we can set the bound for candidate safe prime numbers at $B = 2^{512}$.

There are two computations for which we need to be careful: those involved in verifiable secret sharing (VSS) and the proofs of k-roughness. For VSS, if we create a t-out-of-n sharing using a method similar to Pedersen's VSS [13], then we are creating a random polynomial of degree t which will be evaluated at the integers $\{1, 2, ..., n\}$. In order to make Lagrange interpolation calculations remain in the integers, we will need to multiply our secret by $n!$ and choose the coefficients of this polynomial to be bound by $\pm Bn!$. As a consequence, no shared point on the polynomial should exceed $2Btn^t n!$. We choose prime $p > 2Btn^t n!$ and work over the field $GF(p)$. [7] contains a VSS protocol in which obeying this bound is a requirement for secret share verification. This protocol is the one we will use to share secrets.

[1] Visit `http://oceanstore.cs.berkeley.edu` for the extended version.

Setup

INPUT: A bound B on the size of prime candidates, a number b such that $M_b = \prod_{i \le b} p_i < B$, and a bound β for the prime p (usually either $\beta = 2Btn^t n!$ or $\beta = 2BM_b$).

1. Perform three β-bit joint coin-flipping protocols in parallel. Call the results x, y, and z.
2. Let p be the smallest prime greater than x.
3. Let $g = y \mod p$ and $h = z \mod p$.

Making commitments

To make a commitment to $x \in \mathbb{Z}_p$, a party chooses $r_x \in_R \mathbb{Z}_p$ and the commitment is $g^x h^{r_x} \mod p$.

Fig. 2: The integer commitment scheme

We will give a brief discussion here of k-roughness proofs in order to develop our commitment scheme with full details to follow in Section 3.1. We say that a number a is k-rough if it has no small prime factors less than k. Let $M_b = \sum_{i \le b} p_i$, where p_i is the i^{th} prime. The proof that a number is p_{b+1}-rough involves showing that a is relatively prime to M_b. Specifically, we will compute and commit to x and y such that $ax + M_b y = 1$. We need to make sure that our commitment scheme is sufficient to contain ax and $M_b y$, both of which are bounded by BM_b. Since we will never use ax or $M_b y$ in conjunction with VSS however, we can set the bounds for the prime field in our commitment scheme to be $p > 2B \max(M_b, tn^t n!)$.

Usually, the M_b term will dominate the $tn^t n!$ term, unless there are a large number of players. Because of the way that we create the prime candidates, we must choose b such that $M_b < B$ to ensure that the candidates are not too large. Choosing a large b means that we will be more likely to find a safe prime quickly, but increases our commitment size. However, if the number of players in our generation scheme is large, we can use a large b without this difficulty because of the $tn^t n!$ term. As an example, suppose we are trying to generate a 1024-bit RSA key with factors of size 512-bits. Thus $B = 2^{512}$. We can choose $b = 71$, which makes the bit size of M_b be $|M_b|_2 = 475$. Suppose $t = 5$ and $n = 11$. Then $|tn^t n!)|_2 = 45$. In this case p must be greater than 2^{988} since $475 > 45$.

As an optimization, after we prove the k-roughness of a number a, we can use the secret share conversion methods of [2] to reshare a over a prime field of size p with $2Btn^t n! < p < 2BM_b$. This step will reduce the size of messages for the more communication intensive modular multiplication protocol.

A summary of the commitment scheme is given in Figure 2.

3 Distributed Sieving

In this section, we show how to generate safe prime candidates in a robust way. We begin with a technique by Malkin, Wu, and Boneh [5] called distributed siev-

INPUT: A bound, B, for generated prime candidates. Let $M_b = \prod_{i \leq b} p_i < B$ be a product of the first b primes.
OUTPUT: A number $a + rM_b$ relatively prime to M_b.

1. Each server sieves to find a random integer a_i relatively prime to M_b. In other words, each server finds a random integer with no prime factors smaller than p_b. The a_i are multiplicative shares of a (i.e. $a = \prod_{i=1..n} a_i$). Note that a also has no prime factors less than p_b.
2. The servers produce an additive sharing of a such that each server has a share b_i with $a = \sum_{i=1..n} b_i$.
3. The servers choose a random number $r_i \in_R [0..\frac{B}{M_b}]$, then locally compute $b_i + r_i M_b$. At this point, each server has an additive share of $a + rM_b$ (where $r = \sum_{i=1..n} r_i$) which is also relatively prime to M_b.

Fig. 3: Malkin, Wu, and Boneh Distributed Sieving in the Honest-but-Curious Model

ing which aims to improve the efficiency of distributed random prime generation. Specifically, the technique constructively produces numbers without small prime factors, *rough numbers*. Using such numbers is a common approach in classical prime number generation. The technique described in [2] to find rough numbers is distributed trial division on random candidates. Unfortunately this approach is probabilistic and may take many iterations. Distributed sieving requires only a small constant number of multiparty computations. The algorithm of Malkin, Wu, and Boneh is listed in Figure 3.

The empirical experiments of [5] showed a factor of 10 improvement in the speed of prime generation using this method. However this protocol is secure only in the honest-but-curious model. Specifically, there is no check that the multiplicative shares a_i are being produced correctly. In other words, if even one of the servers chooses a number with a prime factor less than or equal to p_b, the protocol will *never* find a prime number as the sum $a + rM_b$ will always be divisible by that prime. Moreover, unverified additive sharing is central in this protocol and thus requires that a fixed threshold set of the servers be available and honest throughout the protocol.

Thus we need a way to prove and verify that each a_i is relatively prime to M_b using a non-interactive zero-knowledge argument (assuming the random oracle model[2]). After these proofs, we will create a verifiable polynomial sharing of a (resistant to failing or malicious servers) that allows easy computation of $a+rM_b$ and with comparable efficiency to the scheme used in [5].

3.1 Proving a number is k-rough

In order for distributed sieving to work correctly, each player needs to produce a number which is relatively prime to M_b (equivalently, we say that the number

[2] Shoup's RSA signature scheme already invokes the random oracle model, so we lose no security in making this assumption.

Step	Bits
Find integers x_i, y_i such that $a_i x_i + M_b y_i = 1$ holds using the extended Euclidean algorithm.	–
Prove that $x_i \neq 0$, $y_i \neq 0$, and $a_i \neq 0$.	$33\|p\|_2$
Produce $g^{a_i}h^{r_{a_i}}$, $g^{x_i}h^{r_{x_i}}$, $g^{y_i}h^{r_{y_i}}$, and $g^{a_i x_i}h^{r_{a_i x_i}}$, integer commitments to a_i, x_i, y_i, and $a_i x_i$, respectively.	$4\|p\|_2$
Show the multiplicative relationship between the commitments of a_i, x_i and $a_i x_i$.	$8\|p\|_2$
Prove $a_i xi + M_b y_i$ to be 1 by showing knowledge of the discrete logarithm $log_h g^{-1}(g^{a_i x_i}h^{r_{a_i x_i}})(g^{y_i}h^{r_{y_i}})^{M_b}$.	$2\|p\|_2$
Total	$47\|p\|_2$

Fig. 4: Proof that a_i is p_{b+1}-rough. Sizes given are for the random oracle, non-interactive proof versions. See the extended version of this paper for more information.

is p_{b+1}-rough or has no prime factors less than p_{b+1}, where p_i is the i^{th} prime number). The protocol of [5] assumes honesty on the part of the players, but this assumption may not always be acceptable.

Using the properties of integer commitment and the multiplication proof protocol of [12], we can prove that a number is relatively prime to M_b. Note that showing relatively primality of a and M_b is equivalent to showing that there exist integers x, y such that $ax + M_b y = 1$. Since we are actually working in a finite field however, we also need to make sure that none of a, x, or y is 0 since a dishonest prover could make $y = M_b^{-1} \mod p$ and $x = 0$ to prove $ax + M_b y = 1$ while making a of any form the prover desires. A protocol for this proof is given in the extended version of the paper. Thus each player distributing an a_i proves its p_b-roughness by the protocol in Figure 4. If we invoke the random oracle model, we can do all of these proofs without interaction.

Although the integer commitments that we are using are homomorphic and are the basis of our VSS scheme, we will not use the commitment of a_i directly in the sharing. Recall that because we want to make Lagrange interpolation easier, we multiply the secret in our VSS scheme by $n!$. Thus, we commit to a_i and prove the properties we need, then share $n!a_i$ and prove the multiplicative relationship between the two commitments. As mentioned in Section 2.2, the commitment scheme we use for this proof may be larger than the one we need for the rest of the computations. We may choose to reshare $a + rM_b$ over a smaller finite field after its computation.

At this point, each player should also prove that $2a_i + 1$ is relatively prime to M_b as well. This fact will assure us that $2(a + rM_b) + 1$ is also p_{b+1}-rough – a helpful optimization when we later test for safe primality. Each proof of relative primality requires a message of size $47|p|_2$ in addition to the commitment of a_i, so each player will need to send messages of size $95|p|_2$ for each a_i. If we consider the example from Section 2.2 where $|p|_2 = 988$, the message size is $95|p|_2 \approx 11732$ bytes ≈ 11kB.

3.2 Computing the primality candidate

The previous section showed how to produce a polynomial secret sharing of p_{b+1}-rough number a_i. Now we need to multiply the a_i together to produce a. Note of course that we do not need all the a_i of the previous section to create a p_{b+1}-rough number for primality testing – any subset of $\{a_i\}$ will do. This fact is convenient if one of the players was malicious or unavailable in the previous sharing round. The classic technique for multiplying a number shared polynomially was shown in [14]. This method simply multiplies two polynomial shares together, rerandomizes the new (double degree) polynomial, then reduces the degree of the polynomial through a linear transformation. We will need to do this step once for each remaining good player. This multiplication requires the same amount of communication as the multiplication scheme in [5], but produces a polynomial (instead of additive) sharing of a at the conclusion.

Finally, each player chooses a random number $r_i \in_R [0..\frac{B}{M_b}]$. The players all share and commit to these numbers, then each player multiplies their share by M_b and adds the result to their share of a. This arithmetic is all done non-interactively. Now each player has a polynomial share of $a + rM_b$. Note that each player should prove that their r_i is within the range $[0..\frac{B}{M_b}]$ so that the final prime is of the appropriate size. To this end, will use Mao's proof of bit length [15]. This proof requires $\frac{B}{M_b}(6|p|_2 + 1)$ bits in the non-interactive form.

3.3 Ensuring $a + rM_b \equiv 3 \mod 4$

We would like to use an algorithm from [2] for safe primality testing, however this algorithm makes one additional requirement on $a + rM_b$: it must be congruent to $3 \mod 4$. Going back to the distribution of the a_i, each player needs to produce a claim and a proof of each $a_i \mod 4$ in addition to the proofs of relative primality of a_i and M_b. The proofs for $a_i \mod 4$ ensure that no player can force $a + rM_b \not\equiv 3 \mod 4$, thus avoiding progress in the safe primality generation. A full description of this technique is given in the extended paper.

A similar procedure must be performed for the r_i. Then all the players can compute $a + rM_b \mod 4$ and if $a + rM_b \equiv 3 \mod 4$, they do nothing, otherwise they add 2 to their share of $a + rM_b$.

Notice that the proof of congruence $\mod 4$ serves another purpose: it proves that the length of a_i is correct. Thus although the proof seems expensive, it is actually necessary and dual use.

3.4 Communication Efficiency

We now summarize the efficiency of the robust distributed sieving protocol. Specifically we address the size of all the messages sent by a single player. The proofs necessary for each players a_i are the roughness proofs ($95|p|_2$ bits), the bit length proof of r_i ($\frac{B}{M_b}(6|p|_2 + 1)$ bits), the claims of congruence $\mod 4$ ($((|B|_2(|p|_2 + 1) - 9|p|_2) + (4|p|_2 + \frac{B}{M_b}(6|p|_2 + 1) + 2)$ bits), and the proofs of equivalence of the commitment of a_i to the VSS commitment of $a_i n!$ ($2|p|_2$). Using the example numbers from Section 2.2 ($B = 2^{512}$ and $p > 2^{988}$), we see that the proof size is approximately 418267 bytes or about 408kB. These proofs must be broadcast.

We also have to share a_i and r_i using VSS. Sharing two values requires us to broadcast $2t|p|_2$ bits to all players. We can reasonably assume[3] that there is a small constant c such that $n(2t|p|_2 + c)$ is cost of this broadcast to n players. The remaining messages[4] are point-to-point and total $2 \cdot 2n|p|_2$ bits. Assuming the player and threshold numbers from Section 2.2 ($t = 5$ and $n = 11$) along with $c = 1$kB, we see that each player will broadcast about 24.3kB and send about 5.3kB in point-to-point messages. Broadcasting the proofs above costs approximately 4.3MB and dominates the communication costs. This number is large, but as we will see in Section 4.3, we can amortize the cost with a reuse trick.

We also have to multiply the a_i together. Recall from [14] that the communication required for multiplication is simply a secret sharing with a polynomial of order $2t$. We need n such random polynomials. The broadcast cost for these larger polynomials is again 24.3kB, but we only need 2.7kB in point-to-point messages.

In terms of round efficiency, we have only two concerns: the secret sharing of a_i and r_i and the multiplication of the a_i. Thus the number of communication rounds that we use is $2 + n$. Section 4.3 also shows how to parallelized this procedure to use only 1 round of multi-secret VSS.

3.5 Application in Safe Prime Finding

We performed a simulation of this algorithm to get an empirical estimate on the number of iterations required to find a safe prime. When we constructed 4858 1024-bit prime candidates of the form $a + rM_{128}$, we found that the median number of iterations between finding safe primes is approximately 45,000 and the mean is approximately 63,000. When using purely random numbers, we found that the mean number of iterations was about 436,000 and the median was 275,000. Safe primes are unfortunately less dense than unrestricted primes, but distributed sieving seems to be a great help in finding them. Based on our experiment, sieving requires only about 15% of the time required by random searching.

[3] Practical Byzantine broadcast schemes can use secure hashes of the message to verify correct transmission.

[4] We ignore the size of complaint messages, which are relatively small.

INPUT: Shares of the prime candidate ϕ.

1. Locally compute $e = \frac{\phi-1}{2}$ (recall that since $\phi \equiv 3 \mod 4$, we can do the division correctly in the finite field).
2. Compute shares of the base-η representation of e, $[e^{\eta_0}]_j^p, [e^{\eta_1}]_j^p, \cdots, [e^{\eta_{\omega-1}}]_j^p$.
3. Precompute the values needed for modular exponentiation and multiplication.
4. Repeat the following step m times (in parallel):
   (a) Choose $[r]_j^p \in_R \{0,1\}^{2B}$ and set $[g]_j^p = \text{MOD}([r]_j^p, [\phi]_j^p, [\tilde{\phi}]_j^p)$
   (b) Compute $g^e \mod q$
   (c) If $g^e \mod q \notin \{-1, 1\}$ (using the SETMEM algorithm of [2]), output failure.
5. Output success.

Fig. 5: Distributed Miller-Rabin Algorithm

4 Optimizing the Distributed Miller-Rabin Test

In this section, we describe the distributed Miller-Rabin test we will use to check for safe primes. We also give two improvements which improve on the performance of the test, namely optimization of modular exponentiation and multiplication.

The algorithm for our distributed Miller-Rabin test is given in Figure 5. On the whole, the test is not significantly different than the version given in [2], so we will not discuss it thoroughly. The main differences between our version and the original is our preparation for the modular exponentiation step. Instead of converting from additive shares of the candidate to polynomial shares of the bits, we convert from polynomial shares of the candidate to polynomial shares of the base-η representation. We also do precomputations of the values needed for the modular exponentiation and multiplication procedures. In the next few sections, we describe our optimizations to algorithms used by the Miller-Rabin test.

4.1 Optimizing Modular Exponentiation

The modular exponentiation method of [2], shown in Figure 6, uses the familiar square-and-multiply technique with a clever trick to decide when to square and when to multiply. Suppose that $\beta_1, ..., \beta_n$ are the bits of the exponent e and are shared polynomially among the players (the details of how to perform this sharing are given in Section 5.4). The algorithm uses the observation that $g^{\beta_i} = (g-1) * \beta_i - 1$ to decide when to square and when to multiply.

We generalize this algorithm to improve the running time by a constant factor. Suppose we think of step 3a as a lookup instead of a algebraic manipulation – when β_i is 0, we assign d the value 1 and when β_i is 1, we assign d the value g. Thus the modular exponentiation procedure is based on (albeit very immediate) precomputations of the values 1 and g which are referenced based on the value of β_i. We can extend this idea of a precomputed lookup table. Suppose that instead

INPUT: $[g]_j^p$, $[e]_j^p$, $[\phi]_j^p$.
OUTPUT: $[g^e \mod \phi]_j^p$.

1. Reshare the bits of e as $\beta_1, ..., \beta_n$ where β_n is the most significant bit.
2. $c = (g-1) * \beta_n + 1$
3. For $i = n-1$ downto 1, Do
   (a) $d = (g-1) * \beta_i + 1$
   (b) $c = ((c^2 \mod \phi) * d) \mod \phi$
4. Output c.

Fig. 6: Algesheimer et al. Modular Exponentiation

INPUT: $[\kappa]_j^p$ and $[g^0 \mod \phi]_j^p$, $[g^1 \mod \phi]_j^p, \cdots, [g^{\eta-1} \mod \phi]_j^p$.
OUTPUT: $[g^\kappa \mod \phi]_j^p$.

1. In parallel:
   For $i = 0$ to $\eta - 1$, Do
   $\sigma_i = 1 - ||\kappa - i||$
2. In parallel:
   For $i = 0$ to $\eta - 1$, Do
   $\rho_i = \sigma_i * (g^i \mod \phi)$
3. Locally compute $\sum_{i=0..\eta-1} \rho_i$.

Fig. 7: Lookup

INPUT: $[g]_j^p$, $[e]_j^p$, $[\phi]_j^p$.
OUTPUT: $[g^e \mod \phi]_j^p$.

1. Reshare e in base-η: $e^{(\eta_0)}, ..., e^{(\eta_{\omega-1})}$ where $e^{(\eta_{\omega-1})}$ is the most significant digit.
2. $c = \text{LOOKUP}(e^{(\eta_{\omega-1})})$
3. For $i = \omega - 2$ downto 0, Do
   (a) $d = \text{LOOKUP}(e^{(\eta_i)})$
   (b) $c = ((c^\eta \mod \phi) * d) \mod \phi$
4. Output c.

Fig. 8: Revised Modular Exponentiation

of a shared binary representation of e, we have a shared base-η representation and we precompute the values $g^0 \mod q, g^1 \mod q, \cdots, g^{\eta-1} \mod q$. Then we can use the algorithm in Figure 7 to perform a lookup of these values.

Note that in this algorithm, we use a "normalization" procedure defined simply as:

$$||x|| = \begin{cases} 0 & \text{if } x = 0, \\ 1 & \text{otherwise} \end{cases}$$

The implementation of this procedure is given later in Section 5.1.

With this lookup procedure, we can now rewrite the modular exponentiation algorithm of Algesheimer et al. to use generic lookups. The revised algorithm is shown in Figure 8. (The technique for resharing a secret in a different base is given in Section 5.4.) Clearly, this approach uses a smaller number of outer loops, but there is still one concern in step 3b. Specifically, this step requires exponentiating by η in $\mathbb{Z}_q$ and would appear at first glance to remove the advantage of the reduced outer loop. There are however two reasons that this step saves time. First, we are exponentiating by a known, public constant. Thus no extra lookups are necessary in this step. Second, we still only have to perform

Let p be the prime associated with our VSS scheme. Let p' be the smallest prime greater than p.
INPUT: $[A]_j^p$, $[B]_j^p$, and $[\phi]_j^p$.
OUTPUT: $[ABp^{-1} \mod \phi]_j^p$.

Precomputation

1. Reshare A, B, and ϕ over $\mathbb{Z}_{p'}$.
2. Compute shares of $\phi^{-1} \mod p$ and $\phi^{-1} \mod p'$.

Multiplication

1. Compute $-A * B \mod p$ and $A * B \mod p'$ simultaneously.
2. Compute $q = (-A * B \mod p) * (\phi^{-1} \mod p)$.
3. Convert q to a sharing over p'.
4. Compute $q * \phi \mod p'$.
5. Locally compute $r = (A * B \mod p') + (q * \phi \mod p') * (p^{-1} \mod p')$.
6. Convert r to shares over p.

Fig. 9: Modular Multiplication

ω lookups and multiplications by d. Overall we have reduced the number of modular multiplications from $2|e|_2$ to $|e|_2 + \omega$.

Our generic lookup procedure is clearly more expensive than the special case used in [2]. Specifically, it requires η normalizations. However, we use it only ω times during the loop. Moreover, the normalization protocol of Section 5.1 is simpler than modular multiplication, though it requires larger message sizes.

4.2 Optimizing Modular Multiplication

We present an alternative algorithm for modular multiplication which is based on the Montgomery method [16]. In [17], Bajard et al. modified Montgomery multiplication to work by manipulating representations in two different residue number systems (RNS's). We use a highly specialized case of this technique in which the two RNS's are simply prime finite fields. Although this approach requires us to do some pre- and post-computations, we are able to parallelize slightly more than with the algorithm of [2] and we also avoid some additional zero-knowledge proofs in the robust case. The algorithm is listed in Figure 9

Most of the operations performed during the multiplication are familiar: they are modular multiplication and addition in the same field as our shared secrets. These steps are performed relatively quickly. The conversion steps 3 and 6 are new. To convert a sharing over $\mathbb{Z}_p$ to a sharing over $\mathbb{Z}_{p'}$, we use the method of [2] which entails converting the polynomial sharing over $\mathbb{Z}_p$ to an additive sharing over $\mathbb{Z}_p$, converting that sharing to an additive sharing over the integers, converting that sharing to an additive sharing over $\mathbb{Z}_{p'}$, and finally converting that additive sharing to a polynomial sharing over $\mathbb{Z}_p$. This approach is complicated and expensive, but the best way known.

In comparing the algorithm here to the one in [2], we notice that the latter has a much simpler form. Specifically, the algorithm of Algesheimer et al. simply multiplies in the finite field, then takes the remainder of the product $\bmod \phi$. The complexity of the algorithm is in the remainder functionality. Taking a remainder requires two multiplications, a subtraction, and two truncation operations. The truncations involve converting a polynomial sharing $\bmod p$ to an additive sharing over the integers, shifting the additive shares right by some number of bits, then resharing the shifted shares as polynomial shares $\bmod p$. Our algorithm has the same number of multiplication and addition rounds, but we avoid this additional bit shifting. In the honest-but-curious model, the bit shifting is a local operation, so at first it may seem cheap. However since we are in the robust setting, each player must produce a proof of correctness of their truncated share, so we do end up saving some processing time. [5]

Moreover, more of the multiplications in this algorithm are grouped together, rather than being split by conversions as in the [2] algorithm. As mentioned in [14], we can multiply polynomial shares together several times before rerandomizing so long as the degree of the polynomial does not exceed the number of players. The closeness of the multiplications makes this optimization feasible here, but not in [2].

Note that we are doing Montgomery multiplication in this algorithm; the output is actually $(ABp^{-1} \bmod \phi)$, the Montgomery product. When we do exponentiation, we will work with Montgomery products and then at the end, we will convert this product by removing the p^{-1} factor [18]. This step requires one additional Montgomery multiplication at the beginning and the end of the exponentiation.

4.3 Parallel Optimizations

There are (at least) two parallelization tricks that we can employ to improve the speed of our algorithm. The most obvious trick is to generate and test several k-rough candidates simultaneously. Unfortunately, the message sizes required for robustness in the distribute sieving algorithm can grow to be quite large when trying to generate safe primes suitable for RSA.

Thus we suggest that each player can generate and share some small number of k-rough components (i.e. the a_i). The proofs will be large initially, but once the players have shared these numbers, they can recombine them in different ways to produce new candidates. Specifically, let the number of players be l and have each player share m different k-rough numbers. Then if we require that each player gets to contribute one component rough number to each primality

[5] We are not able to avoid truncation proofs entirely – truncation is necessary for the algorithm to convert from additive shares over a finite field to additive shares over the integers [2]. We provide an interactive proof for truncation correctness in the extended paper.

INPUT: $[x]_j^p$.
OUTPUT: $[||x||]_j^p$.

1. Generate p^2 shared secret pairs $(r_i, s_i) \in \mathbb{Z}_p \times \mathbb{Z}_p$
2. Compute in parallel for each pair $u_i = r_i * (1 - r_i * s_i)$ and $v_i = s_i * (1 - s_i * r_i)$
3. Reveal all the u_i and v_i
4. For every i such that $u_i = v_i = 0$, compute and reveal $x - r_i$
5. Let $s = s_i$ where i is the smallest index such that $x - r_i = 0$ or return to step 1 and try again if no such i exists
6. Output $||x|| = x * s$

Fig. 10: Normalization based on Bar-Ilan and Beaver's algorithm

candidate, then there are l^m different combinations possible. Recombinations can proceed in the usual lexigraphical order, for example. A more thorough exploration of these and other parallel techniques is available in the full paper.

5 Multiparty Arithmetic Circuits

This section develops the multiparty circuits that we will need to convert a polynomial secret sharing into a sharing of the same number in base-η. Proofs of secrecy and correctness for the protocols in this section are straightforward since they are the composition of secure protocols.

5.1 Normalization

Recall the normalization procedure we used previously in Section 4.1. Note that the output from this procedure is a *shared secret* containing $||x||$; $||x||$ is neither public nor revealed. We derive our algorithm for normalization from Bar-Ilan and Beaver's algorithm for "extended inverses" [19]. Their method computes either the inverse $x^{-1} \in \mathbb{Z}_p$ of an element $x \in \mathbb{Z}_p$ if $x \neq 0$ and 0 otherwise. We compute this value as well, then multiply x by x^{-1} or 0, respectively, to obtain $||x||$. The full procedure is given in Figure 10. Note that we optimistically generate only p^2 shared secret pairs in step 1, a reduction from the suggested p^4 of [19].

We usually expect we will need only one iteration of this algorithm to calculate $||x||$. During one iteration, we must generate and share $2p^2$ random numbers, do $4p^2$ multiplications, and reveal between $2p^2$ and $4p^2$ numbers. While this complexity may seem high at first, we are saved by the fact that p will quite small in practice.

Notice that these multiplications and random number generations can be batched in advance (as described in Section 4.3) and the addition and scalar multiplications are local operations. All the revelations can be done in parallel.

We will consider the bandwidth required by one normalization. Suppose we choose $p = 37$ (for reasons we will see in the next few sections). We will need

INPUT: $[x]_j^p$ and a publicly known set $S \subset \mathbb{Z}_p$.
OUTPUT: $[0]_j^p$ if $x \notin S$ and $[1]_j^p$ otherwise.

1. $\delta = \prod_{s \in S}(x - s)$
2. Output $1 - ||\delta||$.

Fig. 11: Secret set membership protocol

to share $p^2 = 1369$ random pairs and do $4p^2 + 1 = 5477$ multiplications with upto $4p^2 = 5476$ revelations. The random pairs and multiplications are simply VSS operations, which we can batch. Each random number we share requires broadcasting $t|p|_2$ bits and sending $2n|p|_2$ bits point-to-point. Batching makes the broadcast costs much smaller (since in practice, confirmation messages are secure hashes of the broadcast message), so we will generously assume there is a 1kB per player overhead for this operation. Random secret sharing for multiplication requires a polynomial of degree $2t$, so broadcast costs are higher, but point-to-point bits remain the same. For $p = 37$, $|p|_2 = 6$, so we arrive at a total of $5477 \cdot n \cdot |p|_2(2t + 2) + 1369 \cdot n \cdot |p|_2(t + 2) + 1024n$ bits. If we again use the example $t = 5$ and $n = 11$ from Section 2.2, we need to send about 608kB.

We also need to account for the revelations. Each revelation requires broadcasting 2 numbers to all parties. We can batch these revelations, but we need two steps instead of one because of a dependency in the normalization algorithm. Each batch of revelations requires broadcasting (at most) $2 * 2p^2 = 5476$ numbers. The total, with broadcast costs, is $n(5476 * |p|_2 + 1024) \approx 55$kB for the revelations. To summarize our example, each normalization requires sending about 718kB over 3 rounds. This primitive is our most expensive.

5.2 Secret Set Membership

In this section, we describe an algorithm for "secret set membership." Given $x \in \mathbb{Z}_p$ and $S \subset \mathbb{Z}_p$, this algorithm outputs a shared secret containing 1 if $x \in S$ and a shared secret containing 0 otherwise. We denote this method as computing $x \in_? S$. Readers familiar with the `SETMEM` algorithm in [2] should notice that our algorithm is much simpler than `SETMEM`. This reduction is possible because we do not test whether a shared secret is congruent to a member of S modulo another shared secret modulus p' – we need only test congruence modulo p, which is public. See Figure 11.

Computation of the product in step 1 requires $|S|$ multiplications which we must do in serial. Note of course that we can share all the rerandomizing polynomials for this step in advance, so we only incur one round of secret sharing. The secret set membership algorithm is dominated by the cost of the normalization in step 2. See Section 5.1 for the complexity of that step.

INPUT:

- A radix $\eta = 2^\nu$ which is a power of 2
- Numbers x and y shared in base-η representation over $\mathbb{Z}_p$. Let $x = \sum_{i=0}^{\omega-1} x^{(\eta_i)}\eta^i$ and $y = \sum_{i=0}^{\omega-1} y^{(\eta_i)}\eta^i$.

Assume without loss of generality that $|x|_2 = |y|_2$, $\nu \big| |x|_2$, and $\omega = \frac{|x|_2}{\nu}$.
OUTPUT: Shares of $z^{(\eta_i)}$ for $i = 0, \cdots, \omega - 1$, where $z = x + y$.

1. Generate shared zero $c_0 = 0$
2. For $i = 0$ to $\omega - 1$ Do
   (a) $z^{(\eta_i)} = x^{(\eta_i)} + y^{(\eta_i)} + c_i$
   (b) $c_{i+1} = z^{(\eta_i)} \in_? \{\eta, \eta+1, ..., 2\eta-1\}$
   (c) $z^{(\eta_i)} = z^{(\eta_i)} + (p - \eta) * c_{i+1}$

Fig. 12: Addition in base-η representation

5.3 Base-η Addition Circuit

Assume we have shared base-η representations of two numbers x and y. We will show how to add these numbers together via the normal "elementary school algorithm." While there are more advanced circuits to perform this addition, we describe this simple addition to show the underlying mechanisms at play. Smaller depth circuits may be possible using these mechanisms.

We draw inspiration from the classic binary-coded-decimal addition algorithm. Since we can easily do arithmetic on shared secrets over fields a prime $p > 2\eta$, this model makes sense. See Figure 12 for the full details.

We now give an example to illustrate the costs associated with the protocol. Suppose we have 512-bit numbers x and y with $\eta = 16$. Then we may choose $p = 37$ since two base-16 digits with carry can add to at most 31. All the additions and subtractions are local operations, so we ignore them. Choosing the initial carry bit in step 1 requires one degree t secret sharing and the multiplication in step 2c requires us to do $\omega = 128$ degree $2t$ secret sharings.

Clearly the cost of the set membership to compute the carry bit in step 2b dominates this algorithm. Our set has size $\eta - 1 = 15$, so we must perform this many multiplications in each round. We must also perform 128 total normalizations. Thus we end up doing 1 degree t secret sharing, $15 * 128 + 128 = 2048$ degree $2t$ secret sharings, and 128 normalizations. The normalizations dwarf the other costs. With $t = 5$ and $n = 11$ as before, the messages sent for the whole protocol will total between 70MB and 80MB, depending on the random factors in the normalization algorithm.

5.4 Converting a Number to Base-η Representation

We now have all the tools that we need to convert a polynomial secret sharing of a number x to its base-η representation. The method we use is inspired by the

INPUT:
- A radix $\eta = 2^\nu$ which is a power of 2
- Polynomial shares of secret x.

Assume without loss of generality that $\nu \| |x|_2$ and and $\omega = \frac{|x|_2}{\nu}$.
OUTPUT: Polynomial shares of the base-η representation of x, $[x^{(\eta_0)}]_j^p, \cdots, [x^{(\eta_{\omega-1})}]_j^p$.

1. Convert the polynomial shares of x to additive shares such that $x = \sum_{j=1..n} x_j$. (Recall that we also have verifiers for the additive shares when we use the poly-to-sum protocol of [20].)
2. Reshare each ν-bit block of x_j in polynomial form as $x_j^{(\eta_0)}, x_j^{(\eta_1)}, \cdots, x_j^{(\eta_{\omega-1})}$.
   (a) Prove that these numbers are the base-η form of x_j by showing that the commitment, $g^{x_i} h^{r_{x_i}}$, to x_i contains the same value as $\prod_{i=0..\omega-1} (g^{x_j^{(\eta_i)}} h^{r_{x_j^{(\eta_i)}}})^{\eta^i}$.
   (b) Prove that $|x_j^{(\eta_i)}|_2 = \nu$ for each $i = 0, \cdots, \omega - 1$.
3. Convert all the shares $x_j^{(\eta_i)}$ over p to shares over a smaller prime (e.g. the smallest prime $p' > 2\eta$).
4. Add all the base-*eta* shares of the x_j together.

Fig. 13: Conversion to base-η representation

one from [2] which produces the binary representation of a number. The basic idea is that the secret is reshared as an additive secret, each η digit is reshared as a polynomial, then we use the addition circuit to add all the numbers together in base-η. The conversion algorithm is detailed in Figure 13.

Most of the cost of this algorithm is in the addition step which we addressed in the previous section. The proofs in step 2 are non-trivial, however. Step 2a is relatively simple because of the homomorphic commitment scheme – it requires only $2|p|_2$ additional bits to be broadcast. Step 2b requires a proof of size proportional to the size of x. Specifically, each base-η digit requires a proof of size $\eta(6|p|_2 + 1)$ (See the extended paper for more details). Since we have ω of these digits, the proof expands to $\omega\eta(6|p|_2 + 1) = |x|_2(6|p|_2 + 1)$.

6 Summary

We presented a robust algorithm to generate shared secret, safe prime numbers. Our algorithm owes much to the work of [2] and [5] in the general form. Using this framework, we developed efficient zero-knowledge proofs of knowledge making the algorithm robust. We also borrowed ideas ([17]) from the computer arithmetic world that reduced the number of such proofs we have to transmit during the algorithm. We generalized the modular exponentiation algorithm of [2] to general precomputed lookup tables. We believe our techniques make shared generation of a safe prime much more feasible in the robust setting. Using this primitive and the works of Catalano et al. [7], Shoup's RSA scheme is much closer to practical use without a trusted dealer.

References

[1] Shoup, V.: Practical Threshold Signatures. Lecture Notes in Computer Science **1807** (2000)

[2] Algesheimer, J., Camenisch, J., Shoup, V.: Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In: Proceedings of CRYPTO 2002, Springer Verlag (2002) 417–432

[3] Fouque, P.A., Stern, J.: Fully distributed threshold RSA under standard assumptions. In: Proceedings of Asiacrypt. (2001) 310–330

[4] Damgård, I.B., Koprowski, M.: Practical Threshold RSA Signatures Without a Trusted Dealer. Technical Report RS-00-30, Basic Research in Computer Science, University of Aarhus (2000)

[5] Malkin, M., Wu, T., Boneh, D.: Experimenting with Shared Generation of RSA keys. In: Symposium on Network and Distributed System Security. (1999) 43–56

[6] Cramer, R., Shoup, V.: Signature Schemes Based on the Strong RSA Assumption. ACM Transactions on Information and System Security **3** (2000) 161–185

[7] Catalano, D., Gennaro, R., Halevi, S.: Computing inverses over a secret shared modulus. In: EUROCRYPT 2000. Volume 1807 of LNCS., Springer-Verlag (2000) 190–207

[8] Boneh, D., Franklin, M.: Efficient generation of shared RSA keys. Journal of the ACM (JACM) **48** (2001) 702–722

[9] Frankel, Y., MacKenzie, P.D., Yung, M.: Robust Efficient Distributed RSA-Key Generation. In: Annual ACM Symposium on Theory of Computing. (1998)

[10] Goldwasser, S., Lindell, Y.: Secure Multi-Party Computation Without Agreement. In: 16th International Symposium on DIStributed Computing. Volume 2508 of LNCS. (2002) 17–32

[11] Shamir, A.: How to share a secret. Communications of the ACM **22** (1979)

[12] Damgård, I., Fujisaki, E.: A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order. In: ASIACRYPT. (2002) 125–142

[13] Pedersen, T.: Non-interactive and information-theoretic secure verifiable secret sharing. In: CRYPTO 1991. Volume 576 of LNCS. (1991) 129–140

[14] Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Annual ACM Symposium on Theory of Computing. (1988) 1–10

[15] Mao, W.: Guaranteed Correct Sharing of Integer Factorization with Off-line Shareholders. In: Public Key Cryptography. Volume 1431 of LNCS. (1998) 60–71

[16] Montgomery, P.L.: Modular Multiplication Without Trial Division. Mathematics of Computation **44** (1985) 519–521

[17] Bajard, J.C., Didier, L.S., Kornerup, P.: Modular Multiplication and Base Extensions in Residue Number Systems. In: Proceedings of the 15th IEEE Symposium on Computer Arithmetic. (2001) 59–65

[18] Ç.K. Koç, Acar, T.: Fast Software Exponentiation in GF(2^k). In: Symposium on Computer Arithmetic. (1997) 225–231

[19] Bar-Ilan, J., Beaver, D.: Non-Cryptographic Fault-Tolerant Computing in a Constant Number of Rounds of Interaction. In: 8th ACM Symposium on Principles of Distributed Computation. (1989) 201–209

[20] Frankel, Y., MacKenzie, P., Yung, M.: Adaptively secure distributed public-key systems. Theoretical Computer Science **287** (2002) 535–561