Certifying RSA Public Keys with an Efficient NIZK

Foteini Baldimtsi[†], Sharon Goldberg[*], Leonid Reyzin[*], and Omar Sagga[*]

[†]George Mason University
[*]Boston University

January 12, 2018

Abstract

In many applications, it is important to verify that an RSA public key (N, e) specifies a permutation, in order to prevent attacks due to adversarially-generated public keys. We design and implement a simple and efficient noninteractive zero-knowledge protocol (in the random oracle model) for this task. The key feature of our protocol is compatibility with existing RSA implementations and standards. The protocol works for any choice of e. Applications concerned about adversarial key generation can just append our proof to the RSA public key without any other modifications to existing code or cryptographic libraries. Users need only perform a one-time verification of the proof to ensure that raising to the power e is a permutation of the integers modulo N. For typical parameter settings, the proof consists of nine integers modulo N; generating the proof and verifying it both require about nine modular exponentiations.

1 Introduction

Many applications use an RSA public key (N, e) that is chosen by a party who may be adversarial. In such applications, it is often necessary to ensure that the public key defines a permutation over $\mathbb{Z}_N$: that is, raising to the power e modulo N must be bijective, or, equivalently, every integer between 0 and $N-1$ must have an eth root modulo N.

The folklore solution to this problem (used by, for example, [MRV99], [CMS99], [MPS00], [LMRS04]) is to choose the public RSA exponent e such that e is prime and larger than N. This solution has three drawbacks.

First, because the folklore solution requires $e > N$, e is not in the set of standard values typically used for e in RSA implementations *e.g.,* $e \in \{3, 17, 2^{16}+1\}$. Unless a large prime value for e is standardized, before using the public key, one would have to perform a one-time primality test of on e, to ensure that that e really is prime. This primality test is quite expensive (see Section 3).

Second, most RSA implementations choose a small value for e, typically from a set of standard values $e \in \{3, 17, 2^{16}+1\}$. Choosing a small e significantly reduces the cost of performing an RSA public key operation. However, this efficiency advantage is eliminated in the folklore solution, which requires $e > N$. Unlike the previous drawback, which results in a one-time cost for each public key used, this drawback makes *every* public-key operation about two orders of magnitude more expensive.

Third, the folklore solution is not compatible with existing RSA standards and off-the-shelf implementations. This is because the folklore solution does not ensure that the public key operation

is a permutation over $\mathbb{Z}_N$, where $\mathbb{Z}_N = \{0, 1, ..., N-1\}$. Instead, it only ensures only that the public key operation defines a permutation over the set $\mathbb{Z}_N^*$, where $\mathbb{Z}_N^*$ is the set of values in $\mathbb{Z}_N$ that are relatively prime with N.

Thus, there are no assurances about the values in the set $\mathbb{Z}_N - \mathbb{Z}_N^*$, *i.e.,* the set of values that are less than N but *not* relatively prime with N. (To see this, consider the example $N = 9$ and $e = 11$.) If the RSA public key is generated honestly, this is not a problem, because the set $\mathbb{Z}_N - \mathbb{Z}_N^*$ contains only a negligible fraction of $\mathbb{Z}_N$. However, if an adversary chooses the RSA public key (N, e), it could choose N so that the set $\mathbb{Z}_N - \mathbb{Z}_N^*$ is a large fraction of $\mathbb{Z}_N$. To address this attack, the folklore solution additionally requires the a gcd check along with *every* RSA pubic-key operation, to ensure that the exponentiated value is also relatively prime with N.

Performing a gcd check is not expensive, when compared to a full-length modular exponentiation. However, it does violate compliance with existing RSA standards, which do *not* specify that every RSA public-key operation be accompanied with a gcd check. (See, for instance, the RSA PKCS #1, Version 2.2 specification in RFC8017 [MKJR16], Section 5.) For this reason, typical implementations of RSA do not check that the input is in $\mathbb{Z}_N^*$. Instead, they will perform public-key operations over any value in $\mathbb{Z}_N$. Thus, the folklore solution is not compatible with off-the-shelf RSA implementations, so using it requires modifications to existing code and/or cryptographic libraries, a non-trivial task for developers and practitioners which comes with the risk of introducing new bugs (*e.g.,* because a developer might forget to add the gcd to an RSA operation somewhere in the code, *etc.*).

1.1 Our Contribution.

We present a simple noninteractive zero-knowledge proof (NIZK) in the random oracle model, that allows the holder of an RSA secret key to prove that the corresponding public key defines a permutation over all of $\mathbb{Z}_N$, without leaking information about the corresponding secret key. Our NIZK can be used even when the RSA exponent e is small, which is useful for applications that require fast RSA public key operations. In addition to the NIZK algorithm and a concrete security proof, we present a detailed specification of the prover and verifier algorithms, as well as production-quality implementation and an analysis of its performance.

Because our NIZK is for all values in $\mathbb{Z}_N$, it is compliant with existing cryptographic specifications of RSA (*e.g.,* RFC8017 [MKJR16]). This allows implementors to cleanly combine our NIZK with off-the-shelf RSA cryptographic libraries and primitives, without modifying any existing code. We present a production-quality implementation[cod] of our NIZK in C#, that is ready to be added to the bouncycastle [bou] cryptographic library. In fact, our NIZK implementation has already been integrated into the TumbleBit open-source project [Ntu], where it is being used to prevent the TumbleBit Tumbler from choosing an adversarial RSA key that allows it to steal bitcoins from its users. Integration with TumbleBit was easy, because our NIZK did not require modifications of any of the existing TumbleBit codebase. Instead, the Tumbler just publishes our NIZK along with his RSA public key (N, e), and each user that interacts with the Tumbler simply performs a one-time verification of the NIZK before the interaction begins.

For typical parameter settings, our NIZK consists of 10 elements of $\mathbb{Z}_N$. Generating the NIZK costs roughly 10 full-length RSA exponentiations modulo N. Meanwhile, each verifier pays the one-time cost of verifying our NIZK, which is also roughly equal to 10 full-length exponentiations. When compared to the folklore solution we described earlier, our solution (1) avoids the more expensive one-time primality test and (2) allows the verifier to continue using a small value of e, resulting in better performance for every public-key verification. Beyond this, our NIZK is simple to understand and easy to implement.

1.2 Related Work

Kakvi, Kiltz, and May [KKM12] show how to verify that RSA is a permutation by providing only the RSA public key (N, e) and no additional information, as long as $e > N^{1/4}$. They also show that when e is small, it is impossible, under reasonable complexity assumptions, to verify that (N, e) is a permutation without any additional information [KKM12, Section 1]. Thus, their approach cannot be used when e is small. We circumvent their impossibility by having the prover additionally provide our NIZK (rather than just (N, e)) to the verifier.

Wong, Chan, and Zhu [WCZ03, Section 3.2] and Catalano, Pointcheval, and Pornin [CPP07, Appendix D.2] present protocols (similar to ours) that work only over $\mathbb{Z}_N^*$ rather than the entire $\mathbb{Z}_N$. Thus, these protocols, unlike ours, should only be used in applications that additionally verify $\gcd(\rho, N) = 1$ each time an RSA public-key operation is performed on input ρ. Apart from adding complexity and potentially harming performance, this is a deviation from cryptographic standards and thus requires modifications to cryptographic libraries, which risks introducing new bugs.

The protocols of Camenisch and Michels [CM99, Section 5.2] and Benhamouda et al. [BFGN17] achieve much stronger goals. The former proves $N = pq$ is a product of two safe primes (*i.e.,* $p, q, (p-1)/2$, and $(q-1)/2$ are all prime); the second can prove that any prespecified procedure for generating the primes p and q was followed. These protocols can be used to prove that (N, e) specifies a permutation by imposing mild additional conditions on e (and the prime generation procedure for [BFGN17]). However, these stronger goals are not necessary for our purposes. Our protocol is considerably simpler and more efficient, and does not restrict p and q in any way.

Our protocol builds on the protocol of Bellare and Yung [BY96], who showed how to prove that any function is "close" to a permutation. However, "close" is not good enough for our purposes, because the adversary may be able to force the honest parties to use the few values in $\mathbb{Z}_N$ at which the permutation property does not hold. Thus, additional work is required for our setting. This additional work is accomplished with the help of a simple sub-protocol from Gennaro, Micciancio, and Rabin [GMR98, Section 3.1] for showing the square-freeness of N (a similar sub-protocol in the interactive setting was discovered earlier by Boyar, Friedl, and Lund [BFL89, Section 2.2]). We demonstrate how to combine the ideas of [BY96] and [GMR98] to achieve our design goals.

2 ZK Proof that RSA is a Permutation over $\mathbb{Z}_N$

Let ϕ denote the Euler's totient function, $\phi(N) = |\mathbb{Z}_N^*|$ (see, e.g., [Sho09, Section 2.6] for the relevant background). Recall that $\phi(N) = (p-1)(q-1)$ if $N = pq$.

In this section we present a two-message public-coin honest-verifier zero-knowledge proof for the following promise problem:

$$\mathrm{L_{yes}} = \{(N, e) \text{ where } \gcd(N, \phi(N)) = 1 \text{ and the map } x \mapsto x^e \bmod N \text{ is a permutation of } \mathbb{Z}_N\}$$

$$\mathrm{L_{no}} = \{(N, e) \text{ where the map } x \mapsto x^e \bmod N \text{ is not permutation of } \mathbb{Z}_N\}$$

Recall that in a zero-knowledge proof for a promise problem, we need completeness and zero-knowledge for instances in $\mathrm{L_{yes}}$, and soundness against any instance in $\mathrm{L_{no}}$. (No guarantees are provided in all other instances.) In our case, the gap between $\mathrm{L_{yes}}$ and the complement of $\mathrm{L_{no}}$ is caused by the additional constraint $\gcd(N, \phi(N)) = 1$ in $\mathrm{L_{yes}}$. This constraint enables our efficient construction without hampering its usefulness. That is, the additional constraint in $\mathrm{L_{yes}}$ will hold for an honestly generated RSA public key, because an honestly generated N will be such that

$N = pq$ for distinct equal-length primes p and q. Thus, p will not divide $q-1$ and q will not divide $p-1$, which ensures that $\gcd(N, \phi(N)) = 1$ and the additional constraint is satisfied.

The prover's witness (for instances in $\mathrm{L_{yes}}$) is the prime factorization of N.

Our protocol has perfect completeness, perfect honest-verifier zero-knowledge, and statistical soundness.

In the first message of our protocol, the verifier simply sends some random values from $\mathbb{Z}_N$ to the prover, and the prover responds by using her prime factorization of N to perform a modular exponentiation on each value. In Section 2.3, we use the Fiat-Shamir transform [FS86] to turn the protocol into a noninteractive zero-knowledge proof (NIZK) in the random oracle model. Specifically, both parties will generate the verifier's first message by hashing the input (N, e).

2.1 Overview of our Protocol.

The starting point of our protocol is the observation that if the RSA modulus is *square free* and values in $\mathbb{Z}_N^*$ have eth roots modulo N, then it follows that the values in $\mathbb{Z}_N \backslash \mathbb{Z}_N^*$ also have e^{th} roots modulo N. Thus, our protocol simultaneously tests that N is square free, and that raising to the power e is permutation over $\mathbb{Z}_N^*$.

Square-freeness. Recall that a number N is *square free* if it can be written as $N = p_1 p_2 \dots p_k$ for *distinct* prime numbers p_i. (N is not square free if it is divisible by p^2, where p is some prime.)

It is important to note that RSA will not be a permutation over $\mathbb{Z}_N$ if N is not square free. Specifically, if N is not square free, there will be values in $\mathbb{Z}_N \backslash Z_N^*$ that do not have unique eth roots modulo N. (For instance, suppose that N is not square free because $N = p^3 q$ for a small p (*e.g.,* around 1000) and let $e = 3$ so that $\gcd(e, \phi(N)) = 1$ (i.e. $e = 3$ does not divide $p-1$ or $q-1$). Then raising to the power of $e = 3$ is a permutation of $\mathbb{Z}_N^*$, but not of $\mathbb{Z}_N$. Why? Any $x \in Z_N \backslash Z_N^*$ that is divisible by p will also become divisible by p^3 after being raised to $e = 3$. This implies that any element of $\mathbb{Z}_N$ that is divisible by p but not p^3 will not have an eth root. Thus, an almost $1/p$ fraction of elements of $\mathbb{Z}_N$ will have no eth roots.)

To prove the square-freeness of N, we adapt the protocol of Gennaro, Micciancio, and Rabin [GMR98, Section 3.1] which shows that random elements of $\mathbb{Z}_N^*$ have Nth roots modulo N. We modify their protocol so that it works over all of $\mathbb{Z}_N$, rather than just $\mathbb{Z}_N^*$. This way, we avoid requiring a gcd computation each time a random element is chosen to check that element is in $\mathbb{Z}_N^*$ and not in $\mathbb{Z}_N - \mathbb{Z}_N^*$ (see Lemma 4.3). Apart from reducing the complexity of our protocol, this modification also allows us to easily combine our square-freeness protocol with the permutation-certifying protocol described in the next paragraph.

Our modified square-freeness protocol simply shows that random elements modulo N have Nth roots. First, the verifier chooses random elements ρ_i of $\mathbb{Z}_N$. Then, the prover responds by computing their Nth roots $\sigma_i = (\rho_i)^{1/N} \pmod N$. Finally, the verifier accepts if $\rho_i = (\sigma_i)^N \pmod N$. We note that some some square-free integers N will fail this test (for example the integer 21, and more generally those integers for which $\gcd(N, \phi(N)) > 1$). However, all square-free integers N where $\gcd(N, \phi(N)) = 1$ and will pass this test, which suffices for our purposes.

Certifying permutations. Next, we adapt the protocol of Bellare and Yung [BY96], who showed how to certify that any function is close to a permutation. We observe that for RSA with a square-free modulus N, raising to eth power is either a permutation or very far from one (see Lemma 4.4). Thus, the protocol from [BY96] can be used to certify that (N, e) specifies a permutation. In this protocol, the verifier chooses random elements of $\mathbb{Z}_N$ and the prover responds with their eth roots modulo N.

Combining both protocols. Because any value that has an (eN)th root also has an eth root and an Nth root, we combine the two protocols simply by checking that random values modulo N have eNth roots. Specifically, the prover (who holds the RSA secret key) sends the verifier $\sigma_1 ... \sigma_m$, where each σ_i is (eN)th root of a random values $\rho_i \in Z_N$ selected by the verifier. The verifier validates that each $\sigma_i^{(eN)} = \rho_i \pmod N$.

2.2 Detailed Description of our Interactive Protocol

We are now ready to describe the honest-verifier zero-knowledge protocol.

Our protocol depends on two parameters α and e', which are both primes, at most about 16 bits long. The verifier will reject any N that is divisible by a prime less than α and any e that is divisible by a prime less than e'. (Formally, let $L_{\alpha,e'} = \{(N, e)$ such N is divisible by a prime less than α or e is divisible by a prime less than $e'\}$. We remove $L_{\alpha,e'}$ from $\mathrm{L_{yes}}$ and add its complement to $\mathrm{L_{no}}$.) Any setting of α and e' is valid for security; varying these parameters affects only efficiency. Setting $\alpha = e' = 2$ gives the protocol for the original promise problem, but a higher setting gives better performance. An optimal setting of these parameters is implementation-dependent, since larger e' and α will result in some additional work for the verifier, but will also reduce work for the prover and verifier since m_1 and m_2 in equation (1) below become smaller. When e is a fixed prime like 3, 17, or $2^{16} + 1$, as is standard for many RSA implementations, then we set e' equal to e. We further discuss parameter settings in Section 3.

The prover's witness is the prime factorization of N. Let κ be a security parameter. The protocol will achieve statistical soundness error $2^{-\kappa}$.

Basic Protocol.

1. Both prover and verifier let

$$m_1 = \left\lceil -\kappa / \log_2 \frac{1}{\alpha} \right\rceil \quad \text{and} \quad m_2 = \left\lceil -\kappa / \log_2 \left(\frac{1}{\alpha} + \frac{1}{e'} \left(1 - \frac{1}{\alpha} \right) \right) \right\rceil . \tag{1}$$

   Notice that $m_2 \geq m_1$ since $e' > 1$.

2. The verifier chooses m_1 random values $\rho_i \in Z_N$ and m_2 random values $\rho_j \in Z_N$ and sends them to Prover.

3. The Prover sends back

$$\sigma_i = (\rho_i)^{N^{-1} \bmod \phi(N)} \bmod N$$

   for $i = 1 \ldots m_1$ (these are the Nth roots of ρ_i modulo N) and

$$\sigma_j = (\rho_j)^{e^{-1} \bmod \phi(N)} \bmod N$$

   for $j = 1 \ldots m_2$ (these are the eth roots of ρ_j modulo N).

   (Note that $N^{-1} \bmod \phi(N)$ exists because $\gcd(N, \phi(N)) = 1$, and $e^{-1} \bmod \phi(N)$ exists by Lemma 4.1. The factorization of N allows the prover to compute these values efficiently.)

4. The verifier accepts that (N, e) defines a permutation if all of the following checks pass.

   (a) Check that $N > 0$ and N is not divisible by all the primes less than α.
   (Equivalently, one can let P be the product of all primes less than α (also known as $\alpha - 1$ primorial) and verify that $\gcd(N, P) = 1$.)

(b) Check that $e > 0$ and is e not divisible by all the primes less than e'. (In most implementations of RSA, e is a fixed prime, so the verifier can just check that $e = e'$.)

(c) Verify that $\rho_i = (\sigma_i)^N \bmod N$ for $i = 1 \ldots m_1$. It follows that the $\gcd(N, \phi(N)) = 1$ and therefore N is square-free (with soundness error $2^{-\kappa}$) by Lemma 4.3.

(d) Verify that $\rho_j = (\sigma_j)^e \bmod N$ for $j = 1 \ldots m_2$. It follows that (N, e) define a permutation given that N is square-free (with soundness error $2^{-\kappa}$) by Lemma 4.4.

Improved Protocol. To improve performance, we observe that any value that has an (eN)th root also has an eth root and an Nth root, obtained by raising the (eN)th root to the power N or e, respectively). Thus, instead of requiring the protocol to test $m_1 + m_2$ distinct values, we can instead test just m_2 values (because $m_1 \leq m_2$). Specifically, we modify Steps 2-4 of the basic protocol above as follows:

1. As above.
2. The verifier chooses m_2 random values $\rho_i \in Z_N$.
3. The Prover sends back
$$\sigma_i = (\rho_i)^{(eN)^{-1} \bmod \phi(N)} \pmod N$$
for $i = 1 \ldots m_1$ (for convenience, we call this a "weird RSA signature") and
$$\sigma_i = (\rho_i)^{e^{-1} \bmod \phi(N)} \pmod N$$
for $i = m_1 + 1 \ldots m_2$ (which is just a regular RSA signature).
4. The verifier accepts that (N, e) defined a permutation if all of the following checks pass.
   (a) As above.
   (b) As above.
   (c) Verify that $\rho_i = (\sigma_i)^{eN} \pmod N$ for $i = 1 \ldots m_1$. (This is a "weird RSA verification".)
   (d) Verify that $\rho_i = (\sigma_i)^e \pmod N$ for $i = m_1 + 1 \ldots m_2$. (This is a regular RSA verification.)

Note that for many natural choices of parameters (e, κ, α), we have $m_1 = m_2$, and so step 4d disappears.

We prove that the above protocols are statistically sound honest-verifier zero-knowledge in Section 4.

2.3 Making our Protocol Noninteractive.

We use the Fiat-Shamir paradigm [FS86] to make the improved protocol described in Section 2.2 non-interactive in the random oracle model.

Instead of having the verifier select the random values $\rho_i \in \mathbb{Z}_N$ in Step 2, the Prover samples $\rho_i \in \mathbb{Z}_N$ by himself, by computing the output of the random oracle over the concatenation of (1) the RSA public key (N, e), (2) a salt given as a system parameter , and (3) the index i. Thus, the RSA key (N, e) and the salt determine a deterministic set of $\rho_i \in \mathbb{Z}_N$. The verifier can therefore compute $\rho_i \in Z_N$ on his own, by following the same procedure as the prover, and subsequently verification proceeds as in Step 4.

3 Specification, Implementation and Performance

Specification. The full specification of our NIZK is available in Appendix A. We specify the Improved Protocol of Section 2.2 made non-interactive using the Fiat-Shamir paradigm as described in Section 2.3. Our specification assumes e is a fixed prime and thus sets $e' = e$. It takes in α and the salt as system parameters. The random oracle used to determinstically select the ρ_i values is a "full-domain hash" [BR93] instantiated with the industry-standard MGF1 Mask Generation Function as defined in [MKJR16, Sec. B.2.1]. We use the industry-standard I2OSP and OS2IP to convert between octet strings and integers [MKJR16, Sec. 4.1] and the industry-standard RSASP to perform an RSA secret key operations [MKJR16, Sec. 5.2.1], and RSAVP for RSA public-key operations [MKJR16, Sec. 5.2.2].

Implementation. An open-source implementation of our specification in C#, based on the bouncycastle cryptographic library [bou], is publicly available[cod]. We hope that our implementation will become a part of bouncycastle.

Integration with TumbleBit. Our implementation has already been integrated into the open-source reference implementation of TumbleBit, which is currently being developed for production use [Ntu] [Str17]. TumbleBit [HAB+17] is a unidirectional bitcoin payment hub that allows parties to make fast, anonymous, off-blockchain payments through an untrusted intermediary called the Tumbler. The security of the TumbleBit protocol rests on the assumption that the Tumbler's RSA public key (N, e) defines a permutation over $\mathbb{Z}_N$. In the absence of this assumption, the Tumbler can steal bitcoins from payers.[1] Thus, in addition to publishing (N, e), a Tumbler publishes our NIZK proof that (N, e) defines a permutation, which is verified, during a setup phase, by any payer or payee who wants to participate in the protocol with this Tumbler. Integration with TumbleBit was easy. No modification to the existing TumbleBit protocol or codebase were required; instead, our NIZK was simply added to TumbleBit's setup phase.

[1]Specifically, if RSA is not permutation, then the Tumbler can provide the payee Bob with a puzzle z that has two valid solutions $\epsilon_1 \neq \epsilon_2$ where $z = (\epsilon_1)^e = (\epsilon_2)^e \mod N$, where ϵ_1 allows Bob to decrypt the Tumbler's signature on the transaction that allows Bob to claim his bitcoin, but ϵ_2 does not. Then, to steal a bitcoin, the Tumbler gives payer Alice the solution ϵ_2, so that the Tumbler can claim Alice's bitcoin (because the TumbleBit puzzle-solver protocol will complete correctly, since $\epsilon_2^e = z$) without passing it on to Bob (because ϵ_2 cannot be used to decrypt the Tumbler's signature on the transaction that allows Bob to claim his bitcoin).

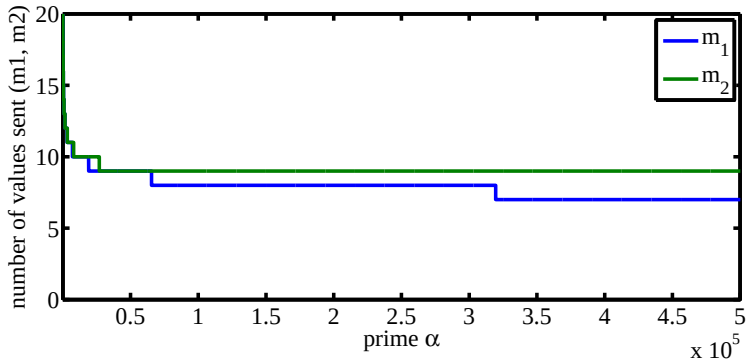

Figure 1: Values of m_1 and m_2 versus the choice of parameter α for our NIZK, when $\kappa = 128$ and $e = e' = 65537$.

	Parameters		Permutation Proof	
α	m_1	m_2	Prove	Verify
41	24	24	632	2326
89	20	20	518	1925
191	17	17	443	1612
937	13	13	334	1216
1667	12	12	311	1127
3187	11	12	308	1042
3347	11	11	281	1025
7151	10	11	284	943
8009	10	10	256	948
19121	9	10	254	853
26981	9	9	233	854
65537	8	9	230	768
319567	7	9	237	713
2642257	6	9	234	956
50859013	5	9	230	6756

Table 1: Proving and verifying times for our C# implementation as observed on an Azure DS1 v2 virtual machine running Windows Server 2016 Datacenter (single-core 2.4 GHz Intel Xeon E5-2673 v3 Haswell processor, 3.5GiB RAM). Time is given in ms. Public exponent is $e = e' = 65537$ and security parameter is $\kappa = 128$.

Parameters and performance for TumbleBit. When used with TumbleBit, our NIZK has parameters $\kappa = 128$, the RSA key length is $|N| = 2048$, the public RSA exponent is $e = e' = 65537$, and the salt is the SHA256 hash of the Genesis block of the Bitcoin blockchain.

The performance of our NIZK largely depends on our choice of the parameter α. A shorter α means that the verifier has to spend less time trying to divide N by primes less than α, but also increases m_1 and m_2, the number of RSA values in the NIZK. The relationship between α and m_1, m_2 is determined by equation (1). Specifically for the TumbleBit parameters, we show this relationship in Figure 1. To evaluate the performance of our NIZK, we choose the smallest value of α that corresponds a given pair of (m_1, m_2) values, and benchmark proving and verifying times for our NIZK for the RSA key length $|N| = 2048$ bits in Table 1 on a single-core of an Intel Xeon processor. We can see from the table that choosing $\alpha = 319567$ (so that $m_1 = 7$ and $m_2 = 9$) gives optimal performance, though performance for $\alpha = 65537$ is roughly similar and the optimal choice is likely implementation-dependent. .

For the optimal choice of α, proving takes about 237 ms (a small fraction of the key generation cost, which is 2022 ms) and verifying takes about 713 ms. For comparison, verification of our NIZK is about 8 times faster than the folklore solution discussed in Section 1, which requires the verifier to spend 5588 ms to perform the Rabin-Miller primality test on a 2048-bit RSA exponent, and also slows down every public-key operation by a factor of about 60 because e is 2048 bits long (instead of $e = 65537$, which is 17 bits long). We should note that even though our solution is much faster than the folklore one, and adds only 12% to the prover's normal RSA key generation cost, it is still relatively expensive for the verifier: for comparison, the public key operation (encryption or signature verification) with $e = 65537$ takes only about 1.4ms.

From Table 1 we also see that verifying is generally slower than proving (until α gets so big that divisibility testing takes too long for the verifier). This follows because proving involves m_1

modular exponentiations (using RSASP), which can be done separately modulo p and modulo q for $N = pq$ (with the exponent reduced modulo $p-1$ and $q-1$), and then combined using the Chinese Remainder Theorem (CRT). Meanwhile, the verifier does not know p and q, and so cannot use (CRT); moreover, the exponent used for modular exponentiations (using RSAVP) is slightly longer than $\phi(n)$, but the verifier does not know $\phi(N)$ and so cannot reduce it. Thus, exponentiations performed by the verifier are slower than those performed by the prover.

4 Security Proofs

We present concrete security proofs for completeness, soundness, and honest-verifier zero-knowledge for the Basic Protocol and Improved Protocol described in Section 2.2. Thus, the security of our NIZK is standard by the Fiat-Shamir paradigm.

4.1 Perfect Completeness

We need to show that the honest prover will be able to carry out step 3, and the verifier's checks in step 4 will pass.

Lemma 4.1. *If for some positive integers N and e, $\gcd(N, \phi(N)) = 1$ and $x \mapsto x^e \bmod N$ is a permutation of $\mathbb{Z}_N$, then $\gcd(e, \phi(N)) = 1$.*

Proof. Since $\gcd(N, \phi(N)) = 1$, there is no prime p such that p^2 divides N (else p divides $\phi(N)$ by [Sho09, Theorem 2.10] and therefore $\gcd(N, \phi(N)) \geq p > 1$). Let $N = p_1 p_2 \dots p_k$ for distinct prime numbers p_i. By Chinese Remainder Theorem (CRT) [Sho09, Theorem 2.8], the ring $\mathbb{Z}_N$ is isomorphic to the product of rings $Z_{p_1} \times \cdots \times \mathbb{Z}_{p_k}$, and therefore $x \mapsto x^e \bmod p_i$ is a permutation of $\mathbb{Z}_{p_i}$ for every i. It suffices to show that e is relatively prime to $p_i - 1$ for every i, because $\phi(N) = (p_1 - 1) \times (p_2 - 1) \times \cdots \times (p_k - 1)$.

Let g_i be the primitive root modulo p_i (also known as generator of $\mathbb{Z}_{p_i}$; it exists by [Sho09, Theorem 7.28]). Let $h_i = g_i^{(p_i-1)/\gcd(e,p_i-1)} \bmod p_i$. Note that $h_i^e \bmod p_i = (h_i^{(p_i-1)})^{e/\gcd(e,p_i-1)} \bmod p_i = 1$ by Fermat's little theorem. Because $1^e \bmod p_i$ is also 1, and raising to the e is a permutation of Z_{p_i}, we know $h_i = 1$. But since g_i is a generator of $Z_{p_i}^*$, the lowest power of g_i that is 1 is $p_i - 1$, and therefore $\gcd(e, p_i - 1) = 1$. □

Lemma 4.2. *If for some positive integers N and f, we have $\gcd(N, \phi(N)) = 1$ and $\gcd(f, \phi(N)) = 1$, then for $g = f^{-1} \bmod \phi(N)$ and for all $x \in \mathbb{Z}_N$, $x^{gf} \bmod N = x$.*

Proof. By the same argument as in the proof of 4.1, we know $N = p_1 p_2 \dots p_k$ for distinct prime numbers p_i. By CRT, it suffices to show that $x^{ef} \bmod p_i = x$ for each i. Indeed, $fg = t\phi(N) + 1$ for some integer t, and therefore $x^{fg} = (x^{p_i-1})^s \cdot x$ for some integer s, and the result follows by Fermat's little theorem when $x \bmod p_i \neq 0$, and trivially when $x \bmod p_i = 0$. □

Thus, Lemma 4.1 shows that the prover will be able to carry out Step 3, because N, e, and therefore Ne are relatively prime to $\phi(N)$, and therefore their inverses modulo $\phi(N)$ (which the prover can compute given the prime factorization of N) exist and can be obtained via extended Euclidean algorithm [Sho09, Section 4.3]. Lemma 4.2 shows that the verifier's checks 4c and 4d will pass. The verifier's checks 4a and 4b will pass by construction of $L_{\alpha,e'}$.

4.2 Statistical Soundness

Let $\phi(N)$ denote $|\mathbb{Z}_N^*|$. The notation $p|N$ means "p divides N".

The following lemma shows that one can validate if an integer N is square-free by checking if random values in $\mathbb{Z}_N$ have Nth roots. This lemma generalizes the result of Gennaro, Micciancio, and Rabin [GMR98, Section 3.1], which worked over $\mathbb{Z}_N^*$ and thus required a gcd computation every time a random value was selected.

Lemma 4.3. *Let $N > 1$ be an integer and p be a prime such that p^2 divides N (i.e., N is not square free). Then, the fraction of elements of $\mathbb{Z}_N$ that have an Nth root modulo N is at most $1/p$.*

Proof. Suppose x has an Nth root modulo N. Then there is a value r such that $r^N \equiv x \pmod N$. Hence, N divides $r^N - x$, which means p^2 divides $r^N - x$ (since p^2 divides N), and therefore r is the Nth root of x modulo p^2. Thus, in order to have an Nth root modulo N, x must have an Nth root modulo p^2. Since a uniformly random element x of $\mathbb{Z}_N$ is also uniform modulo p^2, it suffices to consider what fraction of Z_{p^2} has Nth roots.

By Claim 4.5 below, the number of elements of $Z_{p^2}^*$ that have Nth roots is at most $\phi(p^2)/e'$, where e' is the largest prime divisor of $\gcd(N, \phi(p^2)) = \gcd(N, p(p-1))$. Since $p|N$, we have $e' = p$. Thus, the number of elements of $Z_{p^2}^*$ that have Nth roots is at most $\phi(p^2)/p = p - 1$.

If $x \in Z_{p^2} - Z_{p^2}^*$, then $p|x$. If x has an Nth root r modulo p^2, then $p^2|(r^N - x)$, hence $p|(r^N - x)$, hence $p|r^N$ (because $p|x$ and $p|(r^N - x)$), hence $p|r$ (because p is prime), hence $p^2|r^2$, hence $p^2|r^N$ (because $N > 1$), and hence $p^2|x$ (because $p^2|(r^N - x)$ and $p^2|r^N$). We therefore have that $x \in Z_{p^2}$ and $p^2|x$, which means that $x = 0$.

Thus, the total number of elements of Z_{p^2} that have an Nth root is at most $p-1$ elements from $Z_{p^2}^*$ and one element from $Z_{p^2} - Z_{p^2}^*$ (namely, the element $x = 0$), for a total of at most p elements from Z_{p^2}. Thus, at most a $p/|Z_{p^2}| = 1/p$ fraction of elements of Z_{p^2} have Nth roots. It follows that at most a $1/p$ fraction of elements of $\mathbb{Z}_N$ has Nth roots. □

The following lemma shows that if we know N is square free (which we can test using Lemma 4.3), then we can check whether raising to the power e is a permutation of $\mathbb{Z}_N$, by checking if random values in $\mathbb{Z}_N$ have eth roots.

Lemma 4.4. *Suppose $N > 0$ is a square-free integer so that $N = p_1 p_2 \dots p_k$ for distinct prime numbers p_i, and $e > 0$ is an integer. If raising to the power e modulo N is not a permutation over $\mathbb{Z}_N$, then the fraction of elements of $\mathbb{Z}_N$ that have a root of degree e is at most*

$$\frac{1}{p} + \frac{1}{e'}\left(1 - \frac{1}{p}\right),$$

where e' is the smallest prime divisor of e and p is the smallest prime divisor of N (these are well-defined, because if $N = 1$ or $e = 1$, then raising to the eth power is a permutation over $\mathbb{Z}_N$).

Proof. By Chinese Remainder Theorem (CRT) [Sho09, Theorem 2.8], the ring $\mathbb{Z}_N$ is isomorphic to the product of rings $Z_{p_1} \times \dots \times Z_{p_k}$. Note that if raising to the power e modulo N is not a permutation over $\mathbb{Z}_N$, then there exist $x \not\equiv y \pmod N$ such that $x^e \equiv y^e \pmod N$. Let i be such that $x \not\equiv y \pmod{p_i}$ (it must exist by CRT); then raising to the power e modulo p_i is not an permutation of Z_{p_i}, because $x^e \equiv y^e \pmod{p_i}$ (by CRT).

Since a uniformly random element x of $\mathbb{Z}_N$ is uniform modulo p_i, it suffices to consider what fraction of Z_{p_i} has eth roots. By Claim 4.5 below, the number of elements of $Z_{p_i}^*$ that have eth

roots is at most $\phi(Z_{p_i}^*)/e' = (p_i - 1)/e'$. The only element in $Z_{p_i} - Z_{p_i}^*$ is the element 0. So, in total, at most $(p_i - 1)/e' + 1$ elements of Z_{p_i} have eth roots. Since $p_i \geq p$,

$$\frac{(p_i - 1)/e' + 1}{p_i} = \frac{1}{e'} + \frac{1}{p_i}\left(1 - \frac{1}{e'}\right) \leq \frac{1}{e'} + \frac{1}{p}\left(1 - \frac{1}{e'}\right) = \frac{1}{p} + \frac{1}{e'}\left(1 - \frac{1}{p}\right).$$

□

The proofs of both lemmas above relied on the claim below.

Claim 4.5. *For any integers $N > 1$ and $e > 1$, if raising to the power e modulo N is not a permutation over $\mathbb{Z}_N^*$, then $\gcd(e, \phi(N)) > 1$ and the number of elements of $\mathbb{Z}_N^*$ that have a root of degree e is at most $\phi(N)/e'$, where e' is the largest prime divisor of $\gcd(e, \phi(N))$.*

Proof. Suppose there exist x and y in $\mathbb{Z}_N^*$ such that $x^e \equiv y^e \pmod N$ but $x \not\equiv y \pmod N$. Then $x/y \not\equiv 1 \pmod N$ but $(x/y)^e \equiv 1 \pmod N$. Therefore, the multiplicative order of (x/y) is greater than 1 and divides e [Sho09, Theorem 2.12] and $\phi(N)$ [Sho09, Theorem 2.13], which implies that $\gcd(e, \phi(N)) > 1$. Let e' be the largest prime divisor of $\gcd(e, \phi(N))$.

Because e' is a prime that divides $\phi(N)$, $\mathbb{Z}_N^*$ contains an element z of order e' [Sho09, Theorem 6.42]. Therefore, the homomorphism that takes each element of $\mathbb{Z}_N^*$ to the power e has kernel of size at least e' (because this kernel contains distinct values $z, z^2, \ldots, z^{e'}$ which are all eth roots of 1 because e' divides e). The image of this homomorphism contains exactly the elements that have roots of degree e, and the size of this image is equal to $\phi(N)$ divided by the size of the kernel [Sho09, Theorem 6.23], i.e., at most $\phi(N)/e'$. □

We now combine the results of the above lemmas to demonstrate soundness. Suppose $(N, e) \in \mathrm{L_{no}} \cup \overline{L_{\alpha,e'}}$. If $x \in \overline{L_{\alpha,e'}}$, the verifier will reject in steps 4a or 4b, and soundness holds. Therefore, assume $(N, e) \in L_{\alpha,e'}$. Thus, $(N, e) \in \mathrm{L_{no}}$.

Suppose N is not square free. Since the smallest prime divisor of N is at least α, by applying Lemma 4.3, we know at most $1/\alpha$ fraction of $\mathbb{Z}_N$ will have an Nth root. By choosing m_1 elements of $\mathbb{Z}_N$ and verifying that they have Nth roots, we ensure that the chances that the prover passes Step 4c with N that is not square-free are at most $(1/\alpha)^{m_1} \leq 2^{-\kappa}$.

Now suppose N is square-free but in $\mathrm{L_{no}}$. Since N is square free, the smallest prime divisor of N is at least α, and the smallest prime divisor of e is at least e', we can apply Lemma 4.4 to conclude that at most $1/\alpha + (1 - 1/\alpha)/e'$ fraction of $\mathbb{Z}_N$ have an eth root. By choosing m_2 elements of $\mathbb{Z}_N$ and verifying that they have eth roots, we ensure that the chances that the prover passes Step 4d (of the basic protocol) or 4c and 4d (of the improved protocol) are at most

$$\left(\frac{1}{\alpha} + \frac{1}{e'}\left(1 - \frac{1}{\alpha}\right)'\right)^{m_2} \leq 2^{-\kappa}.$$

4.3 Honest-verifier Zero-Knowledge

If $(N, e) \in \mathrm{L_{yes}}$, then the functions $x \mapsto x^N \bmod N$, $x \mapsto x^e \bmod N$, and $x \mapsto x^{eN} \bmod N$ are all permutations of $\mathbb{Z}_N$ (by Lemma 4.2). Therefore, the honest verifier's view consists of uniformly random values $\rho \in \mathbb{Z}_N$ and their inverses σ under one of these permutations. Equivalently, the honest verifier's view consists of uniformly random $\sigma \in \mathbb{Z}_N$ and their corresponding images ρ under one of these permutations.

We now build simulators for the basic protocol and for the improved protocol. The simulator for the basic protocol will generate m_1 values σ_i and m_2 values σ_j chosen uniformly in $\mathbb{Z}_N$, and set

$\rho_i = \sigma_i^N \bmod N$ and $\rho_j = \sigma_j^e \bmod N$. The simulator for the basic protocol will generate m_1 values σ_i and $m_2 - m_1$ values σ_j chosen uniformly in $\mathbb{Z}_N$, and set $\rho_i = \sigma_i^{eN} \bmod N$ and $\rho_j = \sigma_j^e \bmod N$. Both simulators will output all their σ and ρ values.

The output distributions of the simulator and the honest verifier are identical: each value σ is uniform, and each value ρ is uniquely determined by σ.

Acknowledgements

The authors thank Ethan Heilman and Alessandra Scafuro for useful discussions. This research was supported, in part, by US NSF grants 1717067, 1350733, and 1422965.

References

[BFGN17] Fabrice Benhamouda, Houda Ferradi, Rémi Géraud, and David Naccache. Non-interactive provably secure attestations for arbitrary RSA prime generation algorithms. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part I*, volume 10492 of *Lecture Notes in Computer Science*, pages 206–223. Springer, 2017. `https://eprint.iacr.org/2017/640`.

[BFL89] Joan Boyar, Katalin Friedl, and Carsten Lund. Practical zero-knowledge proofs: Giving hints and using deficiencies. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 1989.

[bou] bouncycastle c# api. `https://www.bouncycastle.org/csharp/index.html`.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM conference on Computer and communications security*, pages 62–73. ACM, 1993.

[BY96] Mihir Bellare and Moti Yung. Certifying permutations: Noninteractive zero-knowledge based on any trapdoor permutation. *J. Cryptology*, 9(3):149–166, 1996. `https://cseweb.ucsd.edu/~mihir/papers/cct.html`.

[CM99] Jan Camenisch and Markus Michels. Proving in zero-knowledge that a number is the product of two safe primes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 107–122. Springer, 1999. `http://www.brics.dk/RS/98/29/BRICS-RS-98-29.pdf`.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Eurocrypt*, volume 99, pages 402–414. Springer, 1999.

[cod] Tumblebit setup implementation. `https://github.com/osagga/TumbleBitSetup`.

[CPP07] Dario Catalano, David Pointcheval, and Thomas Pornin. Trapdoor hard-to-invert group isomorphisms and their application to password-based authentication. *J. Cryptology*, 20(1):115–149, 2007. http://www.di.ens.fr/~pointche/Documents/Papers/2006_joc.pdf.

[FS86] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[GMR98] Rosario Gennaro, Daniele Micciancio, and Tal Rabin. An efficient non-interactive statistical zero-knowledge proof system for quasi-safe prime products. In Li Gong and Michael K. Reiter, editors, *CCS '98, Proceedings of the 5th ACM Conference on Computer and Communications Security, San Francisco, CA, USA, November 3-5, 1998.*, pages 67–72. ACM, 1998. http://eprint.iacr.org/1998/008.

[HAB+17] Ethan Heilman, Leen Alshenibr, Foteini Baldimtsi, Alessandra Scafuro, and Sharon Goldberg. Tumblebit: An untrusted bitcoin-compatible anonymous payment hub. In *24th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2017. https://eprint.iacr.org/2016/575.pdf.

[KKM12] Saqib A. Kakvi, Eike Kiltz, and Alexander May. Certifying RSA. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 404–414. Springer, 2012. http://www.cits.rub.de/imperia/md/content/may/paper/main.pdf.

[LMRS04] Anna Lysyanskaya, Silvio Micali, Leonid Reyzin, and Hovav Shacham. Sequential aggregate signatures from trapdoor permutations. In *Eurocrypt*, volume 3027, pages 74–90. Springer, 2004.

[MKJR16] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. *RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2*. Internet Engineering Task Force (IETF), 2016. https://tools.ietf.org/html/rfc8017.

[MPS00] Philip D. MacKenzie, Sarvar Patel, and Ram Swaminathan. Password-authenticated key exchange based on RSA. In Tatsuaki Okamoto, editor, *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, pages 599–613. Springer, 2000. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.3089&rep=rep1&type=pdf.

[MRV99] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 120–130. IEEE Computer Society, 1999.

[Ntu] Tumblebit implementation in .net core. https://github.com/NTumbleBit/NTumbleBit/.

[Sho09] Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press, second edition, 2009. `http://www.shoup.net/ntb/ntb-v2.pdf`.

[Str17] Stratis Blockchain. Bitcoin privacy is a breeze: Tumblebit successfully integrated into breeze. `https://stratisplatform.com/2017/08/10/bitcoin-privacy-tumblebit-integrated-into-breeze/`, August 2017.

[WCZ03] Duncan S. Wong, Agnes Hui Chan, and Feng Zhu. More efficient password authenticated key exchange based on RSA. In Thomas Johansson and Subhamoy Maitra, editors, *Progress in Cryptology - INDOCRYPT 2003, 4th International Conference on Cryptology in India, New Delhi, India, December 8-10, 2003, Proceedings*, volume 2904 of *Lecture Notes in Computer Science*, pages 375–387. Springer, 2003. `http://www.ccs.neu.edu/home/ahchan/wsl/papers/pake_indocrypt03.pdf`.

A Detailed Specification

The following specification is for the improved protocol made non-interactive, as described in Section 2.2. This specification assumes that the RSA exponent e is prime.

A.1 System Parameters

The system parameters are the RSA modulus length `len`, the security parameter κ (where by default $\kappa = 128$), a small prime α (about 16 bits long or less), and a publicly-known octet string `salt`.

A.2 Proving

System parameters:

1. `salt` (an octet string),
2. α (a prime number)
3. κ (the security parameter, use 128 by default)
4. e, the fixed prime RSA exponent
5. `len`, the RSA key length

Auxiliary Function: getRho, defined in Section A.4.

Input: Distinct equal-length primes p and q greater than α such that the RSA modulus is $N = pq$ is of length `len`, and e does not divide $(p-1)(q-1)$.

Output: $(N, e), \{\sigma_1, ..., \sigma_{m_2}\}$.

Algorithm:

1. Set m_1 and m_2 as in equation 1, Section 2.2, with $e' = e$.
2. Set $N = pq$.

3. Obtain the RSA secret key K as specified by [MKJR16, Sec. 3.2]:

$$K = (p, q, d_{NP}, d_{NQ}, q_{Inv})$$

4. Compute the "weird RSA" secret key corresponding to public key (N, eN) (with exponent eN and modulus N) in the [MKJR16, Sec. 3.2] as

$$K' = (p, q, d_{NP}, d_{NQ}, q_{Inv})$$

where p, q, q_{Inv} are the same as in the normal RSA secret key K and

$$d_{NP} = (eN)^{-1} \bmod (p-1) \qquad d_{NP} = (eN)^{-1} \bmod (q-1) \tag{2}$$

5. For integer $i = 1 \ldots m_2$

   (a) Sample ρ_i, a random element of Z_N, as

   $$\rho_i = \text{getRho}((N, e), \texttt{salt}, i, \texttt{len}, m_2)$$

   (b) If $i \leq m_1$, let

   $$\sigma_i = \text{RSASP1}(K', \rho_i)$$

   where RSASP1 is the RSA signature primitive of [MKJR16, Sec. 5.2.1]. In other words, σ is the RSA decryption of ρ_i using the "weird RSA" secret key K'.
   (It follows that σ_i is (eN)th root of ρ_i.)

   (c) Else let

   $$\sigma_i = \text{RSASP1}(K, \rho_i)$$

   where RSASP1 is the RSA signature primitive of [MKJR16, Sec. 5.2.1]. In other words, σ is the RSA decryption of ρ_i using the regular RSA secret key K.
   (It follows that σ_i is eth root of ρ_i.)

6. Output $(N, e), \{\sigma_1, ..., \sigma_{m_2}\}$.

A.3 Verifying.

System parameters:

1. `salt` (an octet string),
2. α (a prime number)
3. κ (the security parameter, use 128 by default)
4. e, the fixed prime RSA exponent
5. `len`, the RSA key length

Auxiliary Function: getRho, defined in Section A.4.

Input: RSA public key (N, e) and $\{\sigma_1, ..., \sigma_{m_2}\}$.

Output: VALID or INVALID

Algorithm:

1. Check that N is an integer and $N \geq 2^{\texttt{len}-1}$ and $N < 2^{\texttt{len}}$. If not, output INVALID and stop.

2. Check that e is prime. If not, output INVALID and stop.

3. Compute m_1 and m_2 per equation (1), Section 2.2, with $e' = e$.

4. Check that there are exactly m_2 values $\{\sigma_1, ..., \sigma_{m_2}\}$ in the input. If not, output INVALID and stop.

5. Generate the vector Primes($\alpha - 1$), which includes all primes up to and including $\alpha - 1$. (This can be efficiently implemented using the Sieve of Eratosthenes when α is small.)

   For each $p \in$ Primes$(\alpha - 1)$:

   - Check that N is not divisible by p. If not, output INVALID and stop.

   (Alternatively, let `primorial` be the product of all values in Primes($\alpha - 1$). `primorial` should be computed once and should be a system parameter. Check that gcd(`primorial`, N) = 1.)

6. For integer $i = 1 \ldots m_2$

   (a) Sample ρ_i, a random element of Z_N, as

   $$\rho_i = \text{getRho}((N, e), \texttt{salt}, i, \texttt{len}, m_2)$$

   (b) If $i \leq m_1$, check that

   $$\rho_i = \text{RSAVP1}((N, eN), \sigma_i)$$

   where RSAVP1 is the RSA verification primitive of [MKJR16, Sec. 5.2.2]. In other words, check that ρ_i is the RSA encryption of σ_i using the "weird RSA" public key (N, eN). If not, output INVALID and stop.
   (Thus, check that $\rho_i = \sigma_i^{eN} \bmod N$).

   (c) Else check that

   $$\rho_i = \text{RSAVP1}(PK, \sigma_i)$$

   In other words, check that the ρ_i is the RSA encryption of σ_i using the RSA public key (N, e)¿ If not, output INVALID and stop.
   (Thus, check that $\rho_i = \sigma_i^e \bmod N$).

7. Output VALID.

A.4 Auxiliary function: getRho

This function is for rejection sampling of a pseudorandom element $\rho_i \in Z_N$. It is "deterministic," always producing the same output for a given input.

Input:

1. RSA public key (N, e).

2. `salt` (an octet string)

3. Index integer i.

4. Length of RSA modulus `len`
5. Value m_2, with $i \leq m_2$.

Output: ρ_i

Algorithm:

1. Let
$$|m_2| = \lceil \tfrac{1}{8}(\log_2(m_2 + 1)) \rceil$$
be the length of m_2 in octets. (Note: This is an octet length, not a bit length!)
2. Let $j = 1$.
3. While true:
   (a) Let PK be the ASN.1 octet string encoding of the RSA public key (N, e) as specified in [MKJR16, Appendix A].
   (b) Let $EI = \text{I2OSP}(i, |m_2|)$ be the $|m_2|$-octet long string encoding of the integer i. (The I2OSP primitive is specified in [MKJR16, Sec. 4.2].)
   (c) Let $EJ = \text{I2OSP}(j, |j|)$ be the $|j|$-octet long string encoding of the integer j, where $|j| = \lceil \frac{1}{8} \log_2(j + 1) \rceil$.
   (d) Let $s = PK || \texttt{salt} || EI || EJ$ be the concatenation of these octet strings.
   (e) Let $ER = \text{MGF1-SHA256}(s, \texttt{len})$ where H_1 is the MGF1 Mask Generation Function based on the SHA-256 hash function as defined in [MKJR16, Sec. B.2.1], outputting values that are `len` bits long.
   (f) Let $\rho_i = \text{OS2IP}(ER)$ be an integer.
   (That is, convert ER to an `len` bit integer ρ_i using the OS2IP primitive specified in [MKJR16, Sec. 4.1].)
   (g) If $\rho_i \geq N$, then let $j = j + 1$ and continue; Else, break.
   (Note: This step tests if $\rho_i \in Z_N$.)
4. Output integer ρ_i.