Workgroup: Network Working Group
Internet-Draft:
draft-amjad-cfrg-partially-blind-rsa-00
Published: 13 March 2023
Intended Status: Informational
Expires: 14 September 2023
Authors: G. A. Amjad
Google
S. Hendrickson
Google
C. A. Wood
Cloudflare
K. W. L. Yeo
Google

Partially Blind RSA Signatures

Abstract

This document specifies a blind RSA signature protocol that supports public metadata. It is an extension to the RSABSSA protocol recently specified by the CFRG.

Discussion Venues

This note is to be removed before publishing as an RFC.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=cfrg.

Source for this draft and an issue tracker can be found at https://github.com/chris-wood/draft-amjad-cfrg-partially-blind-rsa.

Status of This Memo

Table of Contents

1. Introduction

[RSABSSA] specifies the RSA blind signature protocol, denoted RSABSSA. This is a two-party protocol between client and server (or signer) where they interact to compute sig = Sign(skS, input_msg), where input_msg = Prepare(msg) is a prepared version of the private message msg provided by the client, and skS is the signing key

provided by the server. Upon completion of this protocol, the server learns nothing, whereas the client learns sig. In particular, this means the server learns nothing of msg or input_msg and the client learns nothing of skS.

RSABSSA has a variety of applications, with [PRIVACY-PASS] being a canonical example. While useful, this protocol is limited in that it does not easily accommodate public metadata to be associated with a (message, signature) pair. In this context, public metadata is information that's publicly known to both client and server at the time of computation. This has useful applications in practice. For example, metadata might be used to encode expiration information for a (message, signature) pair. In practice, metadata can be encoded using signing key pairs, e.g., by associating one metadata value with one key pair, but this does not scale well for applications that have large or arbitrary amounts of metadata.

This document specifies a variant of RSABSSA that supports public metadata, denoted RSAPBSSA (RSA Partially Blind Signature with Appendix). Similar to RSABSSA in [RSABSSA], RSAPBSSSA is defined in such a way that the resulting (unblinded) signature can be verified with a standard RSA-PSS library.

2. Conventions and Definitions

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

3. Notation

The following terms are used throughout this document to describe the protocol operations in this document:

* bytes_to_int and int_to_bytes: Convert a byte string to and from a non-negative integer. bytes_to_int and int_to_bytes are implemented as OS2IP and I2OSP as described in [RFC8017], respectively. Note that these functions operate on byte strings in big-endian byte order.

* random_integer_uniform(M, N): Generate a random, uniformly distributed integer R between M inclusive and N exclusive, i.e., M <= R < N.

* bit_len(n): Compute the minimum number of bits needed to represent the positive integer n.

- *inverse_mod(x, n): Compute the multiplicative inverse of x mod n or fail if x and n are not co-prime.

- *is_coprime(x, n): Return true if x and n are co-prime, and false otherwise.

- *len(s): The length of a byte string, in bytes.

- *random(n): Generate n random bytes using a cryptographically-secure random number generator.

- *concat(x0, ..., xN): Concatenation of byte strings. For example, concat(0x01, 0x0203, 0x040506) = 0x010203040506.

- *slice(x, i, j): Return bytes in the byte string x starting from offset i and ending at offset j, inclusive. For example, slice(0x010203040506, 1, 5) = 0x0203040506.

- *random_prime(b): Return a random prime number of length b bits.

- *is_prime(p): Return true if the input integer p is prime, and false otherwise.

4. RSAPBSSA Protocol

The RSAPBSSA protocol consists of two helper functions -- AugmentPrivateKey and AugmentPublicKey -- and four core functions -- Prepare, Blind, BlindSign, and Finalize -- and requires one round of interaction between client and server. Let msg be the client's private input message, info be the public metadata shared between client and server, and (skS, pkS) be the server's private and public key pair. The **REQUIRED** key generation procedure for RSAPBSSA is specified in Section 4.1.

The protocol begins by the client preparing the message to be signed by computing:

```
input_msg = Prepare(msg)
```

The client then initiates the blind signature protocol by computing:

```
blinded_msg, inv = Blind(pkS, input_msg, info)
```

The client then sends blinded_msg to the server, which then processes the message by computing:

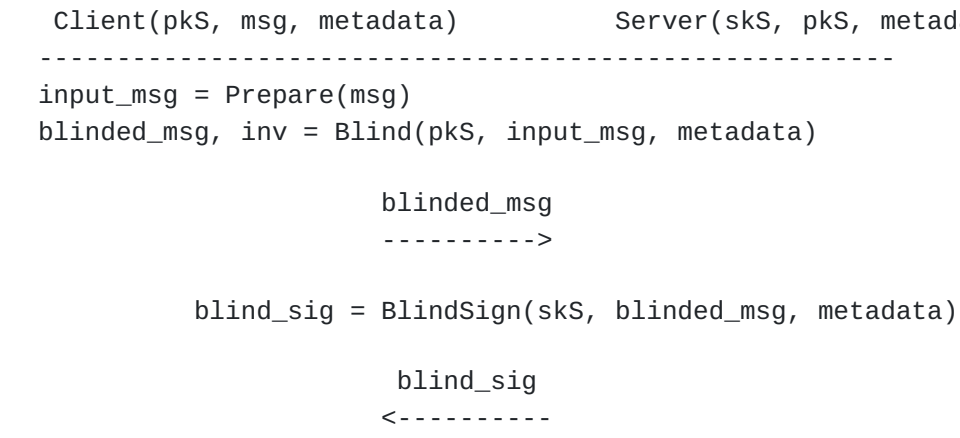
```
blind_sig = BlindSign(skS, blinded_msg, info)
```

The server then sends blind_sig to the client, which then finalizes the protocol by computing:

```
sig = Finalize(pkS, input_msg, info, blind_sig, inv)
```

The output of the protocol is input_msg and sig. Upon completion, correctness requires that clients can verify signature sig over the prepared message input_msg and metadata metadata using the server public key pkS by invoking the RSASSA-PSS-VERIFY routine defined in Section 8.1.2 of [RFC8017]. The Finalize function performs this check before returning the signature. See Section 4.5 for more details about verifying signatures produced through this protocol.

In pictures, the protocol runs as follows:



In the remainder of this section, we specify the Blind, BlindSign, and Finalize functions that are used in this protocol. The Prepare function is as specified in Section 4.1 of [RSABSSA].

4.1. Key Generation

The protocol in this document requires signing key pairs to be generated such that they satisfy a particular criteria. In particular, each RSA modulus for a key pair **MUST** be the product of two safe primes p and q. A safe prime p is a prime number such that p = 2q + 1, where q is also a prime number.

A signing key pair is a tuple (skS, pkS), where each element is as follows:

- skS = (p, q, phi, d), where phi = (p - 1)(q - 1)

- pkS = (n, e), where n = p * q and d * e == 1 mod phi.

The procedure for generating a key pair satisfying this requirement is below.

```
KeyGen(bits)

Inputs:
- bits, length in bits of the RSA modulus, a multiple of 2

Outputs:
- (skS, pkS), a signing key pair

Steps:
1. p = SafePrime(bits / 2)
2. q = SafePrime(bits / 2)
3. while p == q, go to step 2.
4. phi = (p - 1) * (q - 1)
5. e = 65537
6. d = inverse_mod(e, phi)
7. skS = (p, q, phi, d)
8. pkS = (p * q, e)
9. output (skS, pkS)
```

The procedure for generating a safe prime, denoted SafePrime, is below.

```
SafePrime(bits)

Inputs:
- bits, length in bits of the safe prime

Outputs:
- p, a safe prime integer

Steps:
1. q = random_prime(bits - 1)
2. p = (2 * q) + 1
3. if is_prime(p) is True, output p, else go to step 1.
```

4.2. Blind

The Blind function encodes an input message with the corresponding metadata value and blinds it with the server's public key. It outputs the blinded message to be sent to the server, encoded as a byte string, and the corresponding inverse, an integer. RSAVP1 and EMSA-PSS-ENCODE are as defined in Sections 5.2.2 and 9.1.1 of [RFC8017], respectively.

If this function fails with an "blinding error" error, implementations **SHOULD** retry the function again. The probability of one or more such errors in sequence is negligible. This function can also fail with an "invalid input" error, which indicates that one of the inputs (likely the public key) was invalid. Implementations

SHOULD update the public key before calling this function again. See Section 5.1 for more information about dealing with such errors.

Note that this function invokes RSAVP1, which is defined to throw an optional error for invalid inputs. However, this error cannot occur based on how RSAVP1 is invoked, so this error is not included in the list of errors for Blind.

```
Blind(pkS, msg, metadata)

Parameters:
- kLen, the length in bytes of the RSA modulus n
- Hash, the hash function used to hash the message
- MGF, the mask generation function
- sLen, the length in bytes of the salt

Inputs:
- pkS, server public key (n, e)
- msg, message to be signed, a byte string
- metadata, public metadata, a byte string

Outputs:
- blinded_msg, a byte string of length kLen
- inv, an integer

Errors:
- "message too long": Raised when the input message is too long (raised
- "encoding error": Raised when the input message fails encoding (raised
- "blinding error": Raised when the inverse of r cannot be found.
- "invalid input": Raised when the message is not co-prime with n.

Steps:
1. msg_prime = concat("msg", int_to_bytes(len(metadata), 4), metadata, m
2. encoded_msg = EMSA-PSS-ENCODE(msg_prime, bit_len(n))
   with Hash, MGF, and sLen as defined in the parameters
3. If EMSA-PSS-ENCODE raises an error, raise the error and stop
4. m = bytes_to_int(encoded_msg)
5. c = is_coprime(m, n)
6. If c is false, raise an "invalid input" error
   and stop
7. r = random_integer_uniform(1, n)
8. inv = inverse_mod(r, n)
9. If inverse_mod fails, raise an "blinding error" error
   and stop
10. pkM = AugmentPublicKey(pkS, metadata)
11. x = RSAVP1(pkM, r)
12. z = m * x mod n
13. blinded_msg = int_to_bytes(z, kLen)
14. output blinded_msg, inv
```

The blinding factor r **MUST** be randomly chosen from a uniform distribution. This is typically done via rejection sampling. The function AugmentPublicKey is defined in Section 4.6.

4.3. BlindSign

BlindSign performs the RSA private key operation on the client's blinded message input and returns the output encoded as a byte string. RSASP1 is as defined in Section 5.2.1 of [RFC8017].

```
BlindSign(skS, blinded_msg, metadata)

Parameters:
- kLen, the length in bytes of the RSA modulus n

Inputs:
- skS, server private key
- blinded_msg, encoded and blinded message to be signed, a
  byte string
- metadata, public metadata, a byte string

Outputs:
- blind_sig, a byte string of length kLen

Errors:
- "signing failure": Raised when the signing operation fails
- "message representative out of range": Raised when the message represe
  to sign is not an integer between 0 and n - 1 (raised by RSASP1)

Steps:
1. m = bytes_to_int(blinded_msg)
2. skM = AugmentPrivateKey(skS, pkS, metadata)
3. pkM = AugmentPublicKey(pkS, metadata)
4. s = RSASP1(skM, m)
5. m' = RSAVP1(pkM, s)
6. If m != m', raise "signing failure" and stop
7. blind_sig = int_to_bytes(s, kLen)
8. output blind_sig
```

4.4. Finalize

Finalize validates the server's response, unblinds the message to produce a signature, verifies it for correctness, and outputs the signature upon success. Note that this function will internally hash the input message as is done in Blind.

```
Finalize(pkS, msg, metadata, blind_sig, inv)

Parameters:
- kLen, the length in bytes of the RSA modulus n
- Hash, the hash function used to hash the message
- MGF, the mask generation function
- sLen, the length in bytes of the salt

Inputs:
- pkS, server public key (n, e)
- msg, message to be signed, a byte string
- metadata, public metadata, a byte string
- blind_sig, signed and blinded element, a byte string of
  length kLen
- inv, inverse of the blind, an integer

Outputs:
- sig, a byte string of length kLen

Errors:
- "invalid signature": Raised when the signature is invalid
- "unexpected input size": Raised when a byte string input doesn't
  have the expected length.

Steps:
1. If len(blind_sig) != kLen, raise "unexpected input size" and stop
2. z = bytes_to_int(blind_sig)
3. s = z * inv mod n
4. sig = int_to_bytes(s, kLen)
5. msg_prime = concat("msg", int_to_bytes(len(metadata), 4), metadata, m
6. pkM = AugmentPublicKey(pkS, metadata)
7. result = RSASSA-PSS-VERIFY(pkM, msg_prime, sig) with
   Hash, MGF, and sLen as defined in the parameters
8. If result = "valid signature", output sig, else
   raise "invalid signature" and stop
```

Note that pkM can be computed once during Blind and then passed to Finalize directly, rather than being recomputed again.

4.5. Verification

As described in Section 4, the output of the protocol is the prepared message input_msg and the signature sig. The message that applications consume is msg, from which input_msg is derived, along with metadata metadata. Clients verify the signature over msg and info using the server's public key pkS as follows:

1. Compute pkM = AugmentPublicKey(pkS, info).

2. Compute msg_prime = concat("msg", int_to_bytes(len(metadata), 4), metadata, msg).

3. Invoke and output the result of RSASSA-PSS-VERIFY (Section 8.1.2 of [RFC8017]) with (n, e) as pkM, M as msg_prime, and S as sig.

Verification and the message that applications consume therefore depends on which preparation function is used. In particular, if the PrepareIdentity function is used, then the application message is input_msg. In contrast, if the PrepareRandomize function is used, then the application message is slice(input_msg, 32, len(input_msg)), i.e., the prepared message with the random prefix removed.

4.6. Public Key Augmentation

The public key augmentation function (AugmentPublicKey) derives a per-metadata public key that is used in the core protocol. The hash function used for HKDF is that which is associated with the RSAPBSSA instance and denoted by the Hash parameter. Note that the input to HKDF is expanded to account for bias in the output distribution.

```
AugmentPublicKey(pkS, metadata)

Parameters:
- kLen, the length in bytes of the RSA modulus n
- Hash, the hash function used to hash the message

Inputs:
- pkS, server public key (n, e)
- metadata, public metadata, a byte string

Outputs:
- pkM, augmented server public key (n, e')

Steps:
1. hkdf_input = concat("key", metadata, 0x00)
2. hkdf_salt = int_to_bytes(n, kLen)
3. lambda_len = kLen / 2
4. hkdf_len = lambda_len + 16
5. expanded_bytes = HKDF(IKM=hkdf_input, salt=hkdf_salt, info="PBRSA", L
6. expanded_bytes[0] &= 0x3F // Clear two-most top bits
7. expanded_bytes[lambda_len-1] |= 0x01 // Set bottom-most bit
8. e' = bytes_to_int(slice(expanded_bytes, lambda_len))
9. output pkM = (n, e * e')
```

4.7. Private Key Augmentation

The public key augmentation function (AugmentPrivateKey) derives a per-metadata private signing key that is used by the server in the core protocol.

```
AugmentPrivateKey(skS, pkS, metadata)

Parameters:
- kLen, the length in bytes of the RSA modulus n
- Hash, the hash function used to hash the message

Inputs:
- skS, server private key (p, q, phi, d)
- pkS, server public key (n, e)
- metadata, public metadata, a byte string

Outputs:
- skM, augmented server private key (p, q, phi, d')

Steps:
1. (n, e') = AugmentPublicKey(pkS, metadata)
2. d' = inverse_mod(e', phi)
3. output pkM = (p, q, phi, d')
```

5. Implementation and Usage Considerations

This section documents considerations for interfaces to implementations of the protocol in this document. This includes error handling and API considerations.

5.1. Errors

The high-level functions specified in Section 4 are all fallible. The explicit errors generated throughout this specification, along with the conditions that lead to each error, are listed in the definitions for Blind, BlindSign, and Finalize. These errors are meant as a guide for implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, implementations might run out of memory.

Moreover, implementations can handle errors as needed or desired. Where applicable, this document provides guidance for how to deal with explicit errors that are generated in the protocol. For example, "blinding error" is generated in Blind when the client produces a prime factor of the server's public key. Section 4.2 indicates that implementations **SHOULD** retry the Blind function when this error occurs, but an implementation could also handle this exceptional event differently, e.g., by informing the server that the key has been factored.

5.2. Signing Key Generation and Usage

The **RECOMMENDED** method for generating the server signing key pair is as specified in FIPS 186-4 [DSS].

A server signing key **MUST NOT** be reused for any other protocol beyond RSAPBSSA. Moreover, a server signing key **MUST NOT** be reused for different RSAPBSSA encoding options. That is, if a server supports two different encoding options, then it **MUST** have a distinct key pair for each option.

If the server public key is carried in an X.509 certificate, it **MUST** use the RSASSA-PSS OID [RFC5756]. It **MUST NOT** use the rsaEncryption OID [RFC5280].

6. RSAPBSSA Variants

In this section, we define named variants of RSAPBSSA. These variants consider different sets of RSASSA-PSS parameters as defined in Section 9.1.1 of [RFC8017] and explicitly specified in Section 5 of [RSABSSA]. For algorithms unique to RSAPBSSA, the choice of hash function specifies the instantation of HKDF in AugmentPublicKey in Section 4.6. The different types of Prepare functions are specified in Section 4.1 of [RSABSSA].

1. RSAPBSSA-SHA384-PSS-Randomized: This named variant uses SHA-384 as the hash function, MGF1 with SHA-384 as the PSS mask generation function, a 48-byte salt length, and uses the randomized preparation function (PrepareRandomize).

2. RSAPBSSA-SHA384-PSSZERO-Randomized: This named variant uses SHA-384 as the hash function, MGF1 with SHA-384 as the PSS mask generation function, an empty PSS salt, and uses the randomized preparation function (PrepareRandomize).

3. RSAPBSSA-SHA384-PSS-Deterministic: This named variant uses SHA-384 as the hash function, MGF1 with SHA-384 as the PSS mask generation function, 48-byte salt length, and uses the identity preparation function (PrepareIdentity).

4. RSAPBSSA-SHA384-PSSZERO-Deterministic: This named variant uses SHA-384 as the hash function, MGF1 with SHA-384 as the PSS mask generation function, an empty PSS salt, and uses the identity preparation function (PrepareIdentity). This is the only variant that produces deterministic signatures over the client's input message msg.

The **RECOMMENDED** variants are RSAPBSSA-SHA384-PSS-Randomized or RSAPBSSA-SHA384-PSSZERO-Randomized.

See Section 5 of [RSABSSA] for discussion about interoperability considerations and deterministic signatures.

7. Security Considerations

Amjad et al. proved the following properties of RSAPBSSA:

- One-more-unforgeability: For any adversary interacting with the server (i.e., the signer) as a client that interacts with the server at most n times is unable to output n+1 valid message and signature tuples (i.e., the signature verifies for the corresponding message). This holds for any n that is polynomial in the security parameter of the scheme.

- Concurrent one-more-unforgeability: The above holds even in the setting when an adversarial client is interacting with multiple servers (signers) simultaneously.

- Unlinkability: Consider any adversary acting as the server (signer) interacting with n clients using the same public metadata. Afterwards, the adversary randomly receives one of the n resulting signatures as a challenge. Then, the adversary cannot guess which of the n interactions created the challenge signature better than a random guess.

The first two unforgeability properties rely on the Strong RSA Known Target Inversion Problem. This is slightly stronger assumption that the RSA Known Target Inversion Problem used in RSABSSA. In the RSA Known Target Inversion Problem, the challenger is given a fixed public exponent e with the goal of computing the e-th root of n+1 random elements while using an e-th oracle at most n times. In comparison, the Strong RSA Known Target Inversion Problem enables the challenger to choose any public exponents e_1,...,e_n+1 > 1 such that it can be the e_i-th root for the i-th random element. One can view the difference between the Strong RSA Known Target Inversion and RSA Known Target Inversion problems identical to the differences between the Strong RSA and RSA problems.

The final property of unlinkability relies only on the fact that the underlying hash functions are modelled as random oracles.

All the security considerations of RSABSSA in Section 8 of [RSABSSA] also apply to RSAPBSSA here. We present additional security considerations specific to RSAPBSSA below.

7.1. Strong RSA Modulus Key Generation

An essential component of RSAPBSSA is that the KeyGen algorithm in Section 4.1 generates a RSA modulus that is the product of two strong primes. This is essential to ensure that the resulting

outputs of AugmentPublicKey in Section 4.6 does cause errors in AugmentPrivateKey in Section 4.7. We note that an error in AugmentPrivateKey would incur if the output of AugmentPublicKey does not have an inverse modulo phi. By choosing the RSA modulus as the product of two strong primes, we guarantee the output of AugmentPublicKey will never incur errors in AugmentPrivateKey.

It is integral that one uses the KeyGen algorithm for RSAPBSSA instead of the standard RSA key generation algorithms (such as those used in [RSABSSA]). If one uses standard RSA key generation, there are no guarantees provided for the success of the AugmentPrivateKey function and, thus, being able to correctly sign messages for certain choices of public metadata.

7.2. Domain Separation for Public Key Augmentation

The purpose of domain separation is to guarantee that the security analysis of any cryptographic protocol remain true even if multiple instances of the protocol or multiple hash functions in a single instance of the protocol are instantiated based on one underlying hash function.

The AugmentPublicKey in Section 4.6 of this document already provide domain separation by using the RSA modulus as input to the underlying HKDF as the info argument. As each instance of RSAPBSSA will have a different RSA modulus, this effectively ensures that the outputs of the underlying hash functions for multiple instances will be different even for the same input.

Additionally, the hash function invocation used for computing the message digest is domain separated from the hash function invocation used for augmenting the public key in AugmentPublicKey. This domain separation is done by prepending the inputs to each hash function with a unique domain separation tag.

7.3. Choosing Public Metadata

The unlinkability property of RSAPBSSA guarantees anonymity for any signature amongst the set of all interactions with the server (signer) with the same choice of public metadata. In other words, the server is unable to identify the interaction that created the signature. The unlinkability guarantee of RSAPBSSA is only useful when there are a significant number of server (signer) interactions for any value of public metadata. In the extreme case where each server interaction is performed with a different value of public metadata, then the server can uniquely identify the server interaction that created the given signature.

Applications that use RSAPBSSA **MUST** guarantee that the choice of public metadata is limited such that there is a significant number

of server (signer) interactions across many clients for any individual value of public metadata that is signed. This should be contextualized to an application's user population size.

7.4. Denial of Service

RSAPBSSA is suspectible to Denial of Service (DoS) attacks due to the flexibility of choosing public metadata used in AugmentPublicKey in Section 4.6. In particular, an attacker can pick public metadata such that the output of AugmentPublicKey is very large, leading to more computational cost when verifying signatures. Thus, if attackers can force verification with metadata of their choosing, DoS attacks are possible.

For applications where the values of potential public metadata choices are fixed ahead of time, it is possible to try and mitigate DoS attacks. If the set of possible metadata choices is small, then applications **SHOULD** use one of the protocol variants in [RSABSSA] with distinct keys for each metadata value. However, if the set of possible metadata choices is large, rendering this approach infeasible for key management and distribution reasons, other mitigations are possible. As one possible mitigation, first recall that there are only two requirements for the choice of e' in AugmentPublicKey in Section 4.6. First, e' must be smaller than both prime factors of phi. Secondly, the possible values of e' must be large enough to avoid collisions such that two public metadata choices will result in the same e' and, thus, the same augmented public key. During KeyGen in Section 4.1, the server (signer) can pick the smallest length output for the HKDF in AugmentPublicKey such that the output will be different for all relevant public metadata choices while ensuring augmented public keys are smaller.

8. IANA Considerations

This document has no IANA actions.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

[RFC5756] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Updates for RSAES-OAEP and RSASSA-PSS Algorithm Parameters", RFC 5756, DOI 10.17487/RFC5756, January 2010, <https://www.rfc-editor.org/rfc/rfc5756>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/rfc/rfc8174>.

9.2. Informative References

[DSS] "Digital Signature Standard (DSS)", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.186-4, July 2013, <https://doi.org/10.6028/nist.fips.186-4>.

[PRIVACY-PASS] Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-10, 6 March 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-10>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/rfc/rfc5280>.

[RSABSSA] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-11, 16 February 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-11>.

Appendix A. Test Vectors

This section includes test vectors for the RSAPBSSA-SHA384-PSS-Randomized variant defined in Section 4. The following parameters are specified for each test vector:

* p, q, n, e, d: RSA private and public key parameters, each encoded as a hexadecimal string.

* msg: Input messsage being signed, encoded as a hexadecimal string. The hash is computed using SHA-384.

* metadata: Public metadata bound to the signature, encoded as a hexadecimal string.

*eprime: The augmented public key exponent corresponding to e and metadata, encoded as a hexadecimal string.

*rand: Message randomizer prefix, encoded as a hexadecimal string.

*blind: The message blinding value, encoded as a hexadecimal string.

*salt: Randomly-generated salt used when computing the signature. The length is 48 bytes.

*blinded_msg, blinded_sig: The protocol values exchanged during the computation, encoded as hexadecimal strings.

*sig: The output message signature.

```
// Test vector 1
p: dcd90af1be463632c0d5ea555256a20605af3db667475e190e3af12a34a332
4c46a3094062c59fb4b249e0ee6afba8bee14e0276d126c99f4784b23009bf616
8ff628ac1486e5ae8e23ce4d362889de4df63109cbd90ef93db5ae64372bfe1c5
5f832766f21e94ea3322eb2182f10a891546536ba907ad74b8d72469bea396f3
q: f8ba5c89bd068f57234a3cf54a1c89d5b4cd0194f2633ca7c60b91a795a56f
a8c8686c0e37b1c4498b851e3420d08bea29f71d195cfbd3671c6ddc49cf4c1db
5b478231ea9d91377ffa98fe95685fca20ba4623212b2f2def4da5b281ed0100b
651f6db32112e4017d831c0da668768afa7141d45bbc279f1e0f8735d74395b3
d: 4e21356983722aa1adedb084a483401c1127b781aac89eab103e1cfc522154
94981d18dd8028566d9d499469c25476358de23821c78a6ae43005e26b394e305
1b5ca206aa9968d68cae23b5affd9cbb4cb16d64ac7754b3cdba241b72ad6ddfc
000facdb0f0dd03abd4efcfee1730748fcc47b7621182ef8af2eeb7c985349f62
ce96ab373d2689baeaea0e28ea7d45f2d605451920ca4ea1f0c08b0f1f6711eaa
4b7cca66d58a6b916f9985480f90aca97210685ac7b12d2ec3e30a1c7b97b65a1
8d38a93189258aa346bf2bc572cd7e7359605c20221b8909d599ed9d38164c9c4
abf396f897b9993c1e805e574d704649985b600fa0ced8e5427071d7049d
e: 010001
N: d6930820f71fe517bf3259d14d40209b02a5c0d3d61991c731dd7da39f8d69
821552e2318d6c9ad897e603887a476ea3162c1205da9ac96f02edf31df049bd5
5f142134c17d4382a0e78e275345f165fbe8e49cdca6cf5c726c599dd39e09e75
e0f330a33121e73976e4facba9cfa001c28b7c96f8134f9981db6750b43a41710
f51da4240fe03106c12acb1e7bb53d75ec7256da3fddd0718b89c365410fce61b
c7c99b115fb4c3c318081fa7e1b65a37774e8e50c96e8ce2b2cc6b3b367982366
a2bf9924c4bafdb3ff5e722258ab705c76d43e5f1f121b984814e98ea2b2b8725
cd9bc905c0bc3d75c2a8db70a7153213c39ae371b2b5dc1dafcb19d6fae9
msg: 68656c6c6f20776f726c64
metadata: 6d65746164617461
eprime: 30584b72f5cb557085106232f051d039e23358feee9204cf30ea56762
0e90d79e4a7a81388b1f390e18ea5240a1d8cc296ce1325128b445c48aa5a3b34
fa07c324bf17bc7f1b3efebaff81d7e032948f1477493bc183d2f8d94c947c984
c6f0757527615bf2a2f0ef0db5ad80ce99905beed0440b47fa5cb9a2334fea40a
d88e6ef1
rand:
64b5c5d2b2ca672690df59bab774a389606d85d56f92a18a57c42eb4cb164d43
blind: d0261d61ce06309219fbac512db680b381a9fbf2cf08a11a0a1af9424d
cc639777872408c58821c4fc2aa8b607e519b37d79a2beb71ddf48ed9323c2fd0
c87fc0343d0d00e35f4601bb6f3462950437c7f9a42e7c1ae1a9d72e5090b9739
0f94b8764e11173aabcad5a8254e15d4f0e9b047ba0640a8ed4e340b5bcd4b005
12c1474ffae384934a72c7af9010ae147602238502469c6860ddbc43f35a73348
521501a32e7a40fdde6180718755a25de849b82f7dacaea05871a8122057d8726
1172ba8871f2bb8a2c16cd668ae0cfb1fc03b20d4fdf5046a15c59366f0ce631a
887a64c621a1fb1090584cdd6cc958b6af024c3bb8476f12d093d0df352f126e
salt: 59bfc4f4b56aaee9389ba95517a932900971c3417fa5fe4a309477d9c08
87f9b601be62557c67bc4e6f6eb0325eac46d
blinded_msg: 1b9e1057dd2d05a17ad2feba5f87a4083cc825fe06fc70f0b782
062ea0043fa65ec8096ce5d403cfa2aa3b11195b2a655d694386058f626645071
5a936b5764f42977c0a0933ff3054d456624734fd2c019def792f00d30b3ac2f2
7859ea56d835f80564a3ba59f3c876dc926b2a785378ca83f177f7b378513b36a
```

074e7db59448fd4007b54c64791a33b61721ab3b5476165193af30f25164d4806
84d045a8d0782a53dd73774563e8d29e48b175534f696763abaab49fa03a055ec
9246c5e398a5563cc88d02eb57d725d3fc9231ae5139aa7fcb9941060b0bf0192
b8c81944fa0c54568b0ab4ea9c4c4c9829d6dbcbf8b48006b322ee51d784ac93e
4bf13
blinded_sig: 7ef75d9887f29f2232602acab43263afaea70313a0c90374388d
f5a7a7440d2584c4b4e5b886accc065bf4824b4b22370ddde7fea99d4cd67f8ed
2e4a6a2b7b5869e8d4d0c52318320c5bf7b9f02bb132af7365c471e799edd111c
a9441934c7db76c164b0515afc5607b8ceb584f5b1d2177d5180e57218265c07a
ec9ebde982f3961e7ddaa432e47297884da8f4512fe3dc9ab820121262e6a7385
0920299999c293b017cd800c6ec994f76b6ace35ff4232f9502e6a52262e19c03
de7cc27d95ccbf4c381d698fcfe1f200209814e04ae2d6279883015bbf36cabf3
e2350be1e175020ee9f4bb861ba409b467e23d08027a699ac36b2e5ab988390f3
c0ee9
sig: abd6813bb4bbe3bc8dc9f8978655b22305e5481b35c5bdc4869b60e2d5cc
74b84356416abaaca0ca8602cd061248587f0d492fee3534b19a3fe089de18e4d
f9f3a6ad289afb5323d7934487b8fafd25943766072bab873fa9cd69ce7328a57
344c2c529fe96983ca701483ca353a98a1a9610391b7d32b13e14e8ef87d04c0f
56a724800655636cfff280d35d6b468f68f09f56e1b3acdb46bc6634b7a1eab5c
25766cec3b5d97c37bbca302286c17ff557bcf1a4a0e342ea9b2713ab7f935c81
74377bace2e5926b39834079761d9121f5df1fad47a51b03eab3d84d050c99cf1
f68718101735267cca3213c0a46c0537887ffe92ca05371e26d587313cc3f4

// Test vector 2
p: dcd90af1be463632c0d5ea555256a20605af3db667475e190e3af12a34a332
4c46a3094062c59fb4b249e0ee6afba8bee14e0276d126c99f4784b23009bf616
8ff628ac1486e5ae8e23ce4d362889de4df63109cbd90ef93db5ae64372bfe1c5
5f832766f21e94ea3322eb2182f10a891546536ba907ad74b8d72469bea396f3
q: f8ba5c89bd068f57234a3cf54a1c89d5b4cd0194f2633ca7c60b91a795a56f
a8c8686c0e37b1c4498b851e3420d08bea29f71d195cfbd3671c6ddc49cf4c1db
5b478231ea9d91377ffa98fe95685fca20ba4623212b2f2def4da5b281ed0100b
651f6db32112e4017d831c0da668768afa7141d45bbc279f1e0f8735d74395b3
d: 4e21356983722aa1adedb084a483401c1127b781aac89eab103e1cfc522154
94981d18dd8028566d9d499469c25476358de23821c78a6ae43005e26b394e305
1b5ca206aa9968d68cae23b5affd9cbb4cb16d64ac7754b3cdba241b72ad6ddfc
000facdb0f0dd03abd4efcfee1730748fcc47b7621182ef8af2eeb7c985349f62
ce96ab373d2689baeaea0e28ea7d45f2d605451920ca4ea1f0c08b0f1f6711eaa
4b7cca66d58a6b916f9985480f90aca97210685ac7b12d2ec3e30a1c7b97b65a1
8d38a93189258aa346bf2bc572cd7e7359605c20221b8909d599ed9d38164c9c4
abf396f897b9993c1e805e574d704649985b600fa0ced8e5427071d7049d
e: 010001
N: d6930820f71fe517bf3259d14d40209b02a5c0d3d61991c731dd7da39f8d69
821552e2318d6c9ad897e603887a476ea3162c1205da9ac96f02edf31df049bd5
5f142134c17d4382a0e78e275345f165fbe8e49cdca6cf5c726c599dd39e09e75
e0f330a33121e73976e4facba9cfa001c28b7c96f8134f9981db6750b43a41710
f51da4240fe03106c12acb1e7bb53d75ec7256da3fddd0718b89c365410fce61b
c7c99b115fb4c3c318081fa7e1b65a37774e8e50c96e8ce2b2cc6b3b367982366
a2bf9924c4bafdb3ff5e722258ab705c76d43e5f1f121b984814e98ea2b2b8725
cd9bc905c0bc3d75c2a8db70a7153213c39ae371b2b5dc1dafcb19d6fae9

msg: 68656c6c6f20776f726c64
metadata:
eprime: 2ed5a8d2592a11bbeef728bb39018ef5c3cf343507dd77dd156d5eec7
f06f04732e4be944c5d2443d244c59e52c9fa5e8de40f55ffd0e70fbe9093d3f7
be2aafd77c14b263b71c1c6b3ca2b9629842a902128fee4878392a950906fae35
d6194e0d2548e58bbc20f841188ca2fceb20b2b1b45448da5c7d1c73fb6e83fa5
8867397b
rand:
ebb56541b9a1758028033cfb085a4ffe048f072c6c82a71ce21d40842b5c0a89
blind: b8b99b94b48725b059cf26279a79e6cc00d6b98024621f69a6a80e7b74
5050da88ecd81eac3326c96b9ce3695fb7730c17f334d6e4d42088879a42fcb23
bacbc3e0329eab4e9cd3637de07fe64ff2d79fcb44fed2e978a2efb693e87fc9e
971a3a79f746d980d56a6701dcbb6b11a83c8b80d1321e7a4c8ccf541dfd0ceb7
cf71bb0c91be635c979c7a395849ce5c51dd5ab5e7910af1036ffd63da61a56e7
cce6863062ab168dab78ee561e5cc09f303d5aad1c4f6219a9fef85328a2eecb0
6c79272d18c6badfac24c83dac9928d96a2b53e08a60399a8daf656582d6eb7a6
c9fd7d87f5fc3b3752c755c76acb96c103d260690d3ca86383c0dba28c54bb50
salt: a4840382f7404877d0825bb254dd6038a0f3c9e90fb36120bd05fd126ab
c1327c9e00eb82ffb2cac58fa51c7f3a6f3e1
blinded_msg: d1fc97f30efbf116fadd9895130cdd55f939211f7db19ce9a852
87227a02b33fb698b52399f81be0e1f598482000202ec89968085753eae1810f1
4676b514e08238c8aa79d8b999af54e9f4282c6220d4d760716e48e5413f3228c
c59ce10b8252916640de7b9b5c7dc9c2bff9f53b4fb5eb4a5f8bab49af3fd1b95
5d34312073d15030e7fdb44bdb23460d1c5662597f9947092def7fff955a5f3e6
3419ae9858c6405f9609b63c4331e0cf90d24c196bee554f2b78e0d8f6da3d430
8c8d4ae9fbe18a8bb7fa4fc3b9cacd4263e5bd6e12ed891cfdfba8b50d0f37d7a
9abe065238367907c685ed2c224924caf5d8fe41f5db898b09a0501d318d9f65d
88cb8
blinded_sig: 400c1bcdfa56624f15d04f6954908b5605dbeff4cd56f384d753
1669970290d706529d44cde4c972a1399635525a2859ef1d914b4130068ed407c
fda3bd9d1259790a30f6d8c07d190aa98bf21ae9581e5d61801565d96e9eec134
335958b3d0b905739e2fd9f39074da08f869089fe34de2d218062afa16170c150
5c67b65af4dcc2f1aeccd48275c3dacf96116557b7f8c7044d84e296a0501c511
ba1e6201703e1dd834bf47a96e1ac4ec9b935233ed751239bd4b514b031522cd5
1615c1555e520312ed1fa43f55d4abeb222ee48b4746c79006966590004714039
bac7fd18cdd54761924d91a4648e871458937061ef6549dd12d76e37ed417634d
88914
sig: 4062960edb71cc071e7d101db4f595aae4a98e0bfe6843aca3e5f48c9dfb
46d505e8c19806ffa07f040313d44d0996ef9f69a86fa5946cb818a32627fe2df
2a0e80350288ae4fedfbee4193554cc1433d9d27639db8b4635265504d87dca70
54c85e0c882d32887534405e6cc4e7eb4b174383e5ce4eebbfffb217f353102f6
d1a0461ef89238de31b0a0c134dfac0d2a8c533c807ccdd557c6510637596a490
d5258b77410421be4076ecdf2d7e9044327e36e349751f3239681bba10fe633f1
b246f5a9f694706316898c900af2294f47267f2e9ad1e61c7f56bf64328025887
5d29f3745dfdb74b9bbcd5fe3dea62d9be85e2c6f5aed68bc79f8b4a27b3de

// Test vector 3
p: dcd90af1be463632c0d5ea555256a20605af3db667475e190e3af12a34a332
4c46a3094062c59fb4b249e0ee6afba8bee14e0276d126c99f4784b23009bf616

8ff628ac1486e5ae8e23ce4d362889de4df63109cbd90ef93db5ae64372bfe1c5
5f832766f21e94ea3322eb2182f10a891546536ba907ad74b8d72469bea396f3
q: f8ba5c89bd068f57234a3cf54a1c89d5b4cd0194f2633ca7c60b91a795a56f
a8c8686c0e37b1c4498b851e3420d08bea29f71d195cfbd3671c6ddc49cf4c1db
5b478231ea9d91377ffa98fe95685fca20ba4623212b2f2def4da5b281ed0100b
651f6db32112e4017d831c0da668768afa7141d45bbc279f1e0f8735d74395b3
d: 4e21356983722aa1adedb084a483401c1127b781aac89eab103e1cfc522154
94981d18dd8028566d9d499469c25476358de23821c78a6ae43005e26b394e305
1b5ca206aa9968d68cae23b5affd9cbb4cb16d64ac7754b3cdba241b72ad6ddfc
000facdb0f0dd03abd4efcfee1730748fcc47b7621182ef8af2eeb7c985349f62
ce96ab373d2689baeaea0e28ea7d45f2d605451920ca4ea1f0c08b0f1f6711eaa
4b7cca66d58a6b916f9985480f90aca97210685ac7b12d2ec3e30a1c7b97b65a1
8d38a93189258aa346bf2bc572cd7e7359605c20221b8909d599ed9d38164c9c4
abf396f897b9993c1e805e574d704649985b600fa0ced8e5427071d7049d
e: 010001
N: d6930820f71fe517bf3259d14d40209b02a5c0d3d61991c731dd7da39f8d69
821552e2318d6c9ad897e603887a476ea3162c1205da9ac96f02edf31df049bd5
5f142134c17d4382a0e78e275345f165fbe8e49cdca6cf5c726c599dd39e09e75
e0f330a33121e73976e4facba9cfa001c28b7c96f8134f9981db6750b43a41710
f51da4240fe03106c12acb1e7bb53d75ec7256da3fddd0718b89c365410fce61b
c7c99b115fb4c3c318081fa7e1b65a37774e8e50c96e8ce2b2cc6b3b367982366
a2bf9924c4bafdb3ff5e722258ab705c76d43e5f1f121b984814e98ea2b2b8725
cd9bc905c0bc3d75c2a8db70a7153213c39ae371b2b5dc1dafcb19d6fae9
msg:
metadata: 6d65746164617461
eprime: 30584b72f5cb557085106232f051d039e23358feee9204cf30ea56762
0e90d79e4a7a81388b1f390e18ea5240a1d8cc296ce1325128b445c48aa5a3b34
fa07c324bf17bc7f1b3efebaff81d7e032948f1477493bc183d2f8d94c947c984
c6f0757527615bf2a2f0ef0db5ad80ce99905beed0440b47fa5cb9a2334fea40a
d88e6ef1
rand:
f2a4ed7c5aa338430c7026d7d92017f994ca1c8b123b236dae8666b1899059d0
blind: 5b9a334afde50dd24399d1ba9fc1e60be09c7e0cc9e8b8808dfcbc67fe
adeceb34effc9164903149e9de9fb545789fa9885064c58257397500bcaef4e1f
179b88af951636f5c45c1504e4989f9d5af4fb0b171804e2278d1ca85668cabec
6ed9c0f057d5b4f897522b623e5aa3674a4b5dac6174b871ff92f86c1cc81c8f1
d4446210f59219647dbe75bf3e42a17ebba99291e7c45acfa820badae10420c09
3b62e2805932da4e56b1a57a8e31f3a57e2bc59bc72e6ae3563386a7753afeaae
ba1e36a4ab8abe9d0609de4244ae3669157f78f5acd82007f187dc93498bfe3d1
d09ac62b2cb6e5bcd4a202995031690cde0c318ecc13ff5680119ed2773bc751
salt: 5bab5bfebba9eeaa5249e043fcaac352721937e1c3643f35bf815bfc362
73bf17933a0ebc0d0ae90f46077d0e5aadcd6
blinded_msg: 7756a1f89fa33cfc083567e02fd865d07d6e5cd4943f030a2f94
b5c23f3fe79c83c49c594247d02885e2cd161638cff60803184c9e802a659d76a
1c53340972e62e728cc70cf684ef03ce2d05cefc729e6eee2ae46afa17b6b27a6
4f91e4c46cc12adc58d9cb61a4306dac732c9789199cfe8bd28359d1911678e97
09bc159dae34ac7aa59fd0c95962c9f4904bf04aaba8a7e774735bd03be4a02fb
0864a53354a2e2f3502506318a5b03961366005c7b120f0e6b87b44bc15658c3e
8985d69f6adea38c24fe5e7b4bafa1ad6dc7d729281c26dffc88bd34fcc5a5f9d

f9b9781f99ea47472ba8bd679aaada59525b978ebc8a3ea2161de84b7398e4878
b751b
blinded_sig: 2a13f73e4e255a9d5bc6f76cf48dfbf189581c2b170600fd3ab1
a3def148846213239b9d0a981537541cb4f481a602aeebca9ef28c9fcdc63d15d
4296f85d864f799edf08e9045180571ce1f1d3beff293b18aae9d8845068cc0d9
a05b822295042dc56a1a2b604c51aa65fd89e6d163fe1eac63cf603774797b793
6a8b7494d43fa37039d3777b8e57cf0d95227ab29d0bd9c01b3eae9dde5fca714
1919bd83a17f9b1a3b401507f3e3a8e8a2c8eb6c5c1921a781000fee65b6dd851
d53c89cba2c3375f0900001c04855949b7fa499f2a78089a6f0c9b4d36fdfcac2
d846076736c5eaedaf0ae70860633e51b0de21d96c8b43c600afa2e4cc64cd66d
77a8f
sig: 67985949f4e7c91edd5647223170d2a9b6611a191ca48ceadb6c568828b4
c415b6270b037cd8a68b5bca1992eb769aaef04549422889c8b156b9378c50e8a
31c07dc1fe0a80d25b870fadbcc1435197f0a31723740f3084ecb4e762c623546
f6bd7d072aa565bc2105b954244a2b03946c7d4093ba1216ec6bb65b8ca8d2f3f
3c43468e80b257c54a2c2ea15f640a08183a00488c7772b10df87232ee7879bee
93d17e194d6b703aeceb348c1b02ec7ce202086b6494f96a0f2d800f12e855f9c
33dcd3abf6bd8044efd69d4594a974d6297365479fe6c11f6ecc5ea333031c57d
eb6e14509777963a25cdf8db62d6c8c68aa038555e4e3ae4411b28e43c8f57

// Test vector 4
p: dcd90af1be463632c0d5ea555256a20605af3db667475e190e3af12a34a332
4c46a3094062c59fb4b249e0ee6afba8bee14e0276d126c99f4784b23009bf616
8ff628ac1486e5ae8e23ce4d362889de4df63109cbd90ef93db5ae64372bfe1c5
5f832766f21e94ea3322eb2182f10a891546536ba907ad74b8d72469bea396f3
q: f8ba5c89bd068f57234a3cf54a1c89d5b4cd0194f2633ca7c60b91a795a56f
a8c8686c0e37b1c4498b851e3420d08bea29f71d195cfbd3671c6ddc49cf4c1db
5b478231ea9d91377ffa98fe95685fca20ba4623212b2f2def4da5b281ed0100b
651f6db32112e4017d831c0da668768afa7141d45bbc279f1e0f8735d74395b3
d: 4e21356983722aa1adedb084a483401c1127b781aac89eab103e1cfc522154
94981d18dd8028566d9d499469c25476358de23821c78a6ae43005e26b394e305
1b5ca206aa9968d68cae23b5affd9cbb4cb16d64ac7754b3cdba241b72ad6ddfc
000facdb0f0dd03abd4efcfee1730748fcc47b7621182ef8af2eeb7c985349f62
ce96ab373d2689baeaea0e28ea7d45f2d605451920ca4ea1f0c08b0f1f6711eaa
4b7cca66d58a6b916f9985480f90aca97210685ac7b12d2ec3e30a1c7b97b65a1
8d38a93189258aa346bf2bc572cd7e7359605c20221b8909d599ed9d38164c9c4
abf396f897b9993c1e805e574d704649985b600fa0ced8e5427071d7049d
e: 010001
N: d6930820f71fe517bf3259d14d40209b02a5c0d3d61991c731dd7da39f8d69
821552e2318d6c9ad897e603887a476ea3162c1205da9ac96f02edf31df049bd5
5f142134c17d4382a0e78e275345f165fbe8e49cdca6cf5c726c599dd39e09e75
e0f330a33121e73976e4facba9cfa001c28b7c96f8134f9981db6750b43a41710
f51da4240fe03106c12acb1e7bb53d75ec7256da3fddd0718b89c365410fce61b
c7c99b115fb4c3c318081fa7e1b65a37774e8e50c96e8ce2b2cc6b3b367982366
a2bf9924c4bafdb3ff5e722258ab705c76d43e5f1f121b984814e98ea2b2b8725
cd9bc905c0bc3d75c2a8db70a7153213c39ae371b2b5dc1dafcb19d6fae9
msg:
metadata:
eprime: 2ed5a8d2592a11bbeef728bb39018ef5c3cf343507dd77dd156d5eec7

f06f04732e4be944c5d2443d244c59e52c9fa5e8de40f55ffd0e70fbe9093d3f7
be2aafd77c14b263b71c1c6b3ca2b9629842a902128fee4878392a950906fae35
d6194e0d2548e58bbc20f841188ca2fceb20b2b1b45448da5c7d1c73fb6e83fa5
8867397b
rand:
ba3ea4b1e475eebe11d4bfe3a48521d3ba8cd62f3baed9ec29fbbf7ff0478bc0
blind: 910f6bcd2329245a244e7f1d81c4827d85c2687fd458fcf0bb4b7af6ae
81a79e00ab4f2c150a5215dd251cc3a5ed9c5bb2b0dde67dee87477d2786e55fe
f21ee74c8c6cd14208a81579ddf811f30b21595c6ef8e94f63599981da5f01638
a5d6345fab6c95acee5ba0e4e2d43ad77a3b9e4294a30df1db3294e5ca0c092e7
012bbc6cea7a83e73c0ea1a33c3fbf5015d2e4dc7bcbfd051c302eb6a7753d01c
2264f779020989e38206c4600365273323a763357b7819b8ff9e585610828c0ec
542caa68bf2f73f8fb981377699d750512cb66bd693d5e6784f95f86c43f72e9f
9be6b9201a85f1639311dd7e44a7565a3b7fe4b375c5a35cf3444059b05b3863
salt: 4b4ac879a22b5394d4259c817a2cc78713b0438a8de91ffb065f0d66829
e4048367be0278631d8178c3c0d48ad61b665
blinded_msg: 99d725c5613ff87d16464b0375b0976bf4d47319d6946e85f0d0
c2ca79eb02a4c0c282642e090a910b80fee288f0b3b6777e517b757fc6c96ea44
ac570216c8fcd868e15da4b389b0c70898c5a2ed25c1d13451e4d407fe1301c23
1b4dc76826b1d4cc5e64b0e28fb9c71f928ba48c87e308d851dd07fb5a7e0aa5d
0dce61d1348afb4233355374e5898f63adbd5ba215332d3329786fb7c30ef04c1
81b267562828d8cf1295f2ef4a05ef1e03ed8fee65efb7725d8c8ae476f61a359
87e40efc481bcb4b89cb363addfb2adacf690aff5425107d29b2a75b4665d49f2
55c5caa856cdc0c5667de93dbf3f500db8fcce246a70a159526729d82c34df69c
926a8
blinded_sig: a9678acee80b528a836e4784f0690fdddce147e5d4ac506e9ec5
1c11b16ee2fd5a32e382a3c3d276a681bb638b63040388d53894afab79249e159
835cd6bd65849e5d1397666f03d1351aaec3eae8d3e7cba3135e7ec4e7b478ef8
4d79d81039693adc6b130b0771e3d6f0879723a20b7f72b476fe6fef6f21e00b9
e3763a364ed918180f939c3510bb5f46b35c06a00e51f049ade9e47a8e1c3d568
9bd5a43df20b73d70dcacfeed9fa23cabfbe750779997da6bc7269d08b2620aca
a3daa0d9e9d4b87ef841ebcc06a4c0af13f1d13f0808f512c50898586b4fc76d2
b32858a7ddf715a095b7989d8df50654e3e05120a83cec275709cf79571d8f46a
f2b8e
sig: ba57951810dbea7652209eb73e3b8edafc56ca7061475a048751cbfb995a
eb4ccda2e9eb309698b7c61012accc4c0414adeeb4b89cd29ba2b49b1cc661d5e
7f30caee7a12ab36d6b52b5e4d487dbff98eb2d27d552ecd09ca022352c9480ae
27e10c3a49a1fd4912699cc01fba9dbbfd18d1adcec76ca4bc44100ea67b9f1e0
0748d80255a03371a7b8f2c160cf632499cea48f99a6c2322978bd29107d0dffd
d2e4934bb7dc81c90dd63ae744fd8e57bff5e83f98014ca502b6ace876b455d1e
3673525ba01687dce998406e89100f55316147ad510e854a064d99835554de894
9d3662708d5f1e43bca473c14a8b1729846c6092f18fc0e08520e9309a32de

Acknowledgments

Authors' Addresses

Ghous A. Amjad
Google

Email: ghous_amjad@alumni.brown.edu

Scott Hendrickson
Google

Email: scott@shendrickson.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net

Kevin W. L. Yeo
Google

Email: kwlyeo@cs.columbia.edu