Workgroup: Network Working Group
Internet-Draft:
draft-amjad-cfrg-partially-blind-rsa-02
Published: 11 January 2024
Intended Status: Informational
Expires: 14 July 2024
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(sk, input_msg), where input_msg = Prepare(msg) is a prepared version of the private message msg provided by the client, and sk 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 sk.

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 is 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 that does not impose a range limit on the public exponent.

## 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 -- DeriveKeyPair and DerivePublicKey -- 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 (sk, pk) 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:

```
blind_msg, inv = Blind(pk, input_msg, info)
```

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

```
blind_sig = BlindSign(sk, blind_msg, info)
```

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

```
sig = Finalize(pk, 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 info using the server public key pk 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:

```
Client(pk, msg, info)          Server(sk, pk, info)
-----------------------------------------------------
input_msg = Prepare(msg)
blind_msg, inv = Blind(pk, input_msg, info)

                       blind_msg
                       ---------->

            blind_sig = BlindSign(sk, blind_msg, info)

                        blind_sig
                       <----------

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

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 = 2p' + 1, where p' is also a prime number.

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

- sk = (n, p, q, phi, d), where phi = (p - 1)(q - 1), n = p * q, and d is the private exponent

- pk = (n, e), where n = p * q, and e is the public exponent such that 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:
- sk, metadata-specific private key (n, p, q, phi, d)
- pk, metadata-specific public key (n, e)

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. n = p * q
7. sk = (n, p, q, phi, d)
8. pk = (n, e)
9. output (sk, pk)
```

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. p' = random_prime(bits - 1)
2. p = (2 * p') + 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 a "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(pk, msg, info)

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

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

Outputs:
- blind_msg, a byte string of length modulus_len
- 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(info), 4), info, msg)
2. encoded_msg = EMSA-PSS-ENCODE(msg_prime, bit_len(n) - 1)
   with Hash, MGF, and salt_len 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. pk_derived = DerivePublicKey(pk, info)
11. x = RSAVP1(pk_derived, r)
12. z = m * x mod n
13. blind_msg = int_to_bytes(z, modulus_len)
14. output blind_msg, inv
```

The blinding factor r **MUST** be randomly chosen from a uniform distribution. This is typically done via rejection sampling. The function DerivePublicKey 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(sk, blind_msg, info)

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

Inputs:
- sk, private key (n, p, q, phi, d)
- blind_msg, encoded and blinded message to be signed, a
  byte string
- info, public metadata, a byte string

Outputs:
- blind_sig, a byte string of length modulus_len

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(blind_msg)
2. sk_derived, pk_derived = DeriveKeyPair(sk, info)
3. s = RSASP1(sk_derived, m)
4. m' = RSAVP1(pk_derived, s)
5. If m != m', raise "signing failure" and stop
6. blind_sig = int_to_bytes(s, modulus_len)
7. 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(pk, msg, info, blind_sig, inv)

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

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

Outputs:
- sig, a byte string of length modulus_len

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) != modulus_len, raise "unexpected input size" and s
2. z = bytes_to_int(blind_sig)
3. s = z * inv mod n
4. sig = int_to_bytes(s, modulus_len)
5. msg_prime = concat("msg", int_to_bytes(len(info), 4), info, msg)
6. pk_derived = DerivePublicKey(pk, info)
7. result = RSASSA-PSS-VERIFY(pk_derived, msg_prime, sig) with
   Hash, MGF, and salt_len as defined in the parameters
8. If result = "valid signature", output sig, else
   raise "invalid signature" and stop

   Note that pk_derived 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 info. Clients verify the signature over msg and info using the server's public key pk as follows:

1. Compute pk_derived = DerivePublicKey(pk, info).

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

3. Invoke and output the result of RSASSA-PSS-VERIFY (Section 8.1.2 of [RFC8017]) with (n, e) as pk_derived, 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 Derivation

The public key derivation function (DerivePublicKey) derives a per-metadata public key that is used in the core protocol. The hash function used for HKDF as defined in [RFC5869] 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.

```
DerivePublicKey(pk, info)

Parameters:
- modulus_len, the length in bytes of the RSA modulus n. This MUST be a
- Hash, the hash function used to hash the message

Inputs:
- pk, public key (n, e)
- info, public metadata, a byte string

Outputs:
- pk_derived, metadata-specific public key (n, e')

Steps:
1. hkdf_input = concat("key", info, 0x00)
2. hkdf_salt = int_to_bytes(n, modulus_len)
3. lambda_len = modulus_len / 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 pk_derived = (n, e')
```

### 4.7. Key Pair Derivation

The key pair derivation function (DeriveKeyPair) derives a pair of private and public keys specific to a metadata value that are used by the server in the core protocol.

```
DeriveKeyPair(sk, info)

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

Inputs:
- sk, private key (n, p, q, phi, d)
- info, public metadata, a byte string

Outputs:
- sk_derived, metadata-specific private key (n, p, q, phi, d')
- pk_derived, metadata-specific public key (n, e')

Steps:
1. (n, e') = DerivePublicKey(n, info)
2. d' = inverse_mod(e', phi)
3. sk_derived = (n, p, q, phi, d')
4. pk_derived = (n, e')
5. Output (sk_derived, pk_derived)
```

## 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 implementers. 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 Usage

A server signing key **MUST NOT** be reused for any other protocol beyond RSAPBSSA. In particular, the same signing key **MUST NOT** be used for both the RSAPBSSA and RSABSSA protocols. 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 instantiation of HKDF in DerivePublicKey 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 7 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 DerivePublicKey in Section 4.6 do cause errors in DeriveKeyPair in Section 4.7. We note that an error in DeriveKeyPair would incur if the output of DerivePublicKey does not have an inverse modulo phi. By choosing the RSA modulus as the product of two strong primes, we guarantee the output of DerivePublicKey will never incur errors in DeriveKeyPair.

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 DeriveKeyPair 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 remains 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 DerivePublicKey function 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 DerivePublicKey. 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 susceptible to Denial of Service (DoS) attacks due to the flexibility of choosing public metadata used in DerivePublicKey in Section 4.6. In particular, an attacker can pick public metadata such that the output of DerivePublicKey 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.

## 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>.

**[RFC5869]** Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

**[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>.

**[RSABSSA]** Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-14, 10 July 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-14>.

### 9.2. Informative References

**[PRIVACY-PASS]** Celi, S., Davidson, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-16, 3 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-16>.

**[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>.

## Appendix A. Acknowledgments

The authors would like to thank Nikita Borisov for pointing out an issue with a prior attempt at mitigating DoS attacks.

## Appendix B. Test Vectors

This section includes test vectors for the RSAPBSSA-SHA384-PSS-Deterministic variant defined in Section 4. The following parameters are specified for each test vector, where each hexidecimal value uses an unsigned big-endian convention:

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

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

* info: 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.

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

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

*blind_msg, blind_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
info: 6d65746164617461
eprime: 30581b1adab07ac00a5057e2986f37caaa68ae963ffbc4d36c16ea5f3
689d6f00db79a5bee56053adc53c8d0414d4b754b58c7cc4abef99d4f0d0b2e29
cbddf746c7d0f4ae2690d82a2757b088820c0d086a40d180b2524687060d768ad
5e431732102f4bc3572d97e01dcd6301368f255faae4606399f91fa913a6d699d
6ef1
r: d55491221c9a9ce5687b84669880abbc4db57c8f82864a450a5bf7c3f0
902884fa418c74bf663f3bfcff74a4792356f3ce052f128b084f8b028cf432533
27514f4b38430c69f19f155634429803badd1f6849d8603882eb9b648b697cb2f
2c4069b504562e19bb9f1cf99da47c198c2ae04f4bd3add78025e80f146edce48
dc3e9dc0ba3ee14bc97489050e26dc8935f3ecfcaea07c9c1a3d8e41be1e49dc8
aa171ac4cec9d1cddd8066b13767901dcb339e2cce40d11f5cff6c870012bca49
109ce6e81e165d3831531cbf8503f3cfde68340789979cba96602e70613a13869
aff57f2170e31ebe85564e3f026d8cd1835e59144fb8c008391c55d2fb1a5488
salt: 648ea74482fbab69876817ee3c2055a6921a458648c802c09a23f8825b2
59724e41c960ef29febe16a04e120c8b1cc1a
blind_msg: cfd613e27b8eb15ee0b1df0e1bdda7809a61a29e9b6e9f3ec7c3
45353437638e85593a7309467e36396b0515686fe87330b312b6f89df26dc1cc8
8dd222186ca0bfd4ffa0fd16a9749175f3255425eb299e1807b76235befa57b28
f50db02f5df76cf2f8bcb55c3e2d39d8c4b9a0439e71c5362f35f3db768a5865b
864fdf979bc48d4a29ae9e7c2ea259dc557503e2938b9c3080974bd86ad8b0daa
f1d103c31549dcf767798079f88833b579424ed5b3d700162136459dc29733256
```

```
f18ceb74ccf0bc542db8829ca5e0346ad3fe36654715a3686ceb69f73540efd20
530a59062c13880827607c68d00993b47ad6ba017b95dfc52e567c4bf65135072
b12a4
blind_sig: ca7d4fd21085de92b514fbe423c5745680cace6ddfa864a9bd97
d29f3454d5d475c6c1c7d45f5da2b7b6c3b3bc68978bb83929317da25f491fee8
6ef7e051e7195f3558679b18d6cd3788ac989a3960429ad0b7086945e8c4d38a1
b3b52a3903381d9b1bf9f3d48f75d9bb7a808d37c7ecebfd2fea5e89df59d4014
a1a149d5faecfe287a3e9557ef153299d49a4918a6dbdef3e086eeb264c0c3621
bcd73367195ae9b14e67597eaa9e3796616e30e264dc8c86897ae8a6336ed2cd9
3416c589a058211688cf35edbd22d16e31c28ff4a5c20f1627d09a71c71af372e
dc18d2d7a6e39df9365fe58a34605fa1d9dc53efd5a262de849fb083429e20586
e210e
sig: cdc6243cd9092a8db6175b346912f3cc55e0cf3e842b4582802358dddf6f
61decc37b7a9ded0a108e0c857c12a8541985a6efad3d17f7f6cce3b5ee20016e
5c36c7d552c8e8ff6b5f3f7b4ed60d62eaec7fc11e4077d7e67fc6618ee092e20
05964b8cf394e3e409f331dca20683f5a631b91cae0e5e2aa89eeef4504d24b45
127abdb3a79f9c71d2f95e4d16c9db0e7571a7f524d2f64438dfb32001c00965f
f7a7429ce7d26136a36ebe14644559d3cefc477859dcd6908053907b325a34aaf
654b376fade40df4016ecb3f5e1c89fe3ec500a04dfe5c8a56cad5b086047d2f9
63ca73848e74cf24bb8bf1720cc9de4c78c64449e8af3e7cddb0dab1821998

// 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
info:
```

```
eprime: 2ed579fcdf2d328ebc686c52ccaec247018832acd530a2ac72c0ec2b9
2db5d6bd578e91b6341c1021142b45b9e6e5bf031f3dd62226ec4a0f9ef99e45d
d9ccd60aa60a0c59aac271a8caf9ee68a9d9ff281367dae09d588d3c7bca7f18d
e48b6981bbc729c4925c65e4b2a7f054facbb7e5fc6e4c6c10110c62ef0b94eec
397b
r: 532103acf62670e3176eb1cfee7c2c46c7986704b869387924c33e8358
8c7cac67882570aede836b51b44a565c872a91bbf4f0f8396019113ef382963d3
a51b91429993e821217d3e85b2253e0daa0e9cfc440c37a37707f7aed383d98b3
150f21e1146c58c28d4a49046b8e97f834e4cb95e5483dfc42eaa17bdce947631
7f710b7488cc06cf61a1c449faa1d34119f2c3cd6ead79f9de14358b1c750bf2c
312fcbba3c511341fd4952ba2fcd486a9e81fd829e47cd8a0ac0273d7594c69eb
4aebfaec3c59aa1a016582410d9f4be14dac4b1a66f61eeb3e108af3868e410f7
7436765ba1df7c9a5cf37d8ec3dced6f5689da9703618a5cc7bf6d60f7b4209c
salt: 134520fb9ae6076594b4488fa31cae4e8e3efaca5ae4377bd586aac58e9
0f8925826b4b4fff2e21fdb933c4fbb6467a2
blind_msg: 5e6568cd0bf7ea71ad91e0a9708abb5e97661c41812eb994b672
f10aa8983151113aeaabcf1306fa5a493e3dbdd58fc8bdb61aac934fae832676b
cab7abacdcc1b9c1f2af3586ae009042293b6945fee0aeffb2d2b8a24f82614b8
be39bab71a535f6d65f1631e927dbd471b0753e7a63a201c7ecd26e7fbbb5e21e
02f865b64e20731004c395b0e059a92fffa4c636ac4c00db9aa086b5dd1a3dd10
1bb04970b12ca3f4936f246e32d394f328cea2510554060e8d291acdbee04b8bc
91e967241ba45f3509d63ded5f9b358f4216f37a885e563b7baa93a717ca7cdbe
10e398d14bb2d5a1376b4a5f83226ce2c575087bc28d743caeff9c1b11cc8bd02
f5f14
blind_sig: 72c4e0f4f677aa1dbb686e23b5944b3afdc7f824711a1f7486d1
ed6fa20aad255a1412885aee04c64359964e694a713da2a1684325c1c31401cac
1ea39a9e454675b55f743ff144ac605d0ed254b12d9bdd43b0e8a17c0d4711239
732e45e4166261d0b16d2f29403c5f2584a29b225daa7530ba15fc9af15ed2ce8
fcb126ad0b0758fd522fbf99a83e4cfe0539aa264d06a1633deee0053f45fc8a9
44f1468a0c0c449155139779a3230c8fa41a81858418151fa195f57ea645699f5
50d3cb37c549542d436071d1af74e629f938fa4717ca9def382fc35089e4caec9
e5d740c38ecb2aa88c90176d2f322866acfd50e2b92313161e81327f889aca0c9
4bcb8
sig: a7ace477c1f416a40e93ddf8a454f9c626b33c5a20067d81bdfef7b88bc1
5de2b04624478b2134b4b23d91285d72ca4eb9c6c911cd7be2437f4e3b24426bc
e1a1cb52e2c8a4d13f7fd5c9b0f943b92b8bbcba805b847a0ea549dbc249f2e81
2bf03dd6b2588c8af22bf8b6bba56ffd8d2872b2f0ebd42ac8bd8339e5e638061
99deec3cf392c078f66e72d9be817787d4832c45c1f192465d87f6f6c333ce1e8
c5641c7069280443d2227f6f28ff2045acdc368f2f94c38a3c909591a27c93e17
78630aeeeb623805f37c575213091f096be14ffa739ee55b3f264450210a4b2e6
1a9b12141ca36dd45e3b81116fc286e469b707864b017634b8a409ae99c9f1

// 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:
info: 6d65746164617461
eprime: 30581b1adab07ac00a5057e2986f37caaa68ae963ffbc4d36c16ea5f3
689d6f00db79a5bee56053adc53c8d0414d4b754b58c7cc4abef99d4f0d0b2e29
cbddf746c7d0f4ae2690d82a2757b088820c0d086a40d180b2524687060d768ad
5e431732102f4bc3572d97e01dcd6301368f255faae4606399f91fa913a6d699d
6ef1
r: 6e1de89fc58417836aa76fefe4876b8b311af2eb94a8226d5796273171
48551d90b6f9db614b590e7f66f34644a2f6a3568ec78852b7f45876f576a7ee6
0c19bb0fbbdf1c85d7b36cf7bdf80fb925830c07285efae69e0c019d8d99fd5c6
20f83361c9411541fddf4bfe27e73f756bf594742a8253119d134e1ad67f02228
59c4ab243868bb23a6468c01ead9a617657056685f19fcd423b9e916c5e3e3b21
f92d0e12667d695084a42ae97a548d5982a51b67dd09c188c051d20236e24b231
e80a96449390e9032bad350645f5d4a162ddf3d61506ef6737b4f9fe6064a1d2f
afc7849e5039a98ebf14a800dc2423fccc1293f28a2c66ec22983cab922c1cc6
salt: 1ade5e965d1946a69dc495e78c8524910094f08405471664d4898fa3612
bf03fd03b3ae8140a737cb13e223e35219b58
blind_msg: 92d5456738e0cfe0fa770b51e6a72d633d7cec3a945459f1db96
dbc500a5d1bca34a839059579759301c098231b102fb1e114bf9f892f42f902a3
36f4a3585b23efa906dfcb94213f4d3b39951551cedecbf51efa213ad030cf821
ee3fa46a57d67429f838ff728f47111f7f1b22000a979c0f56cc581396935780d
76173410d2a8a5688cd59622903008fe50af1fcc5e7cf96affad7e60fbed67996
c7a377effa0f08d9273cd33536b2625c9575d10636cc964636a1500f4fcb22aab
bef77fe415cbc7245c1032d34bd480ee338f55be0a79c0076d9cf9c94c0db3003
a33b23c62dbd1a85f2b15db5d153b318cca53c6d68e1e63bafa39c9a43be72f36
d2569
blind_sig: a76a1c53566a9781de04d87e8c3a0bc902b47819e7b900580654
215b0a710cb563b085b5e9fff150791f759da03a139dfc9159c21410f1e3d345b
8c5dcca35211772900f85c5eec065987cbdbf303e9651196223263a713e4135d6
b20bfa8fb8212341665647a9a7e07a831ccbf9e62d9366ec9ac0bbe96228e6fbb

```
848f8f6f474cce68e3556dc882847e9e61b5b5e02bbfd6152aeca74e8782a54ff
e6552d63fb837738a05044b38f7e908c4989b202bd858695c61e12cf9d47ef276
a17917e39f942871defd9747541957b1e2f8950da43c9a05ba4835bded23c24cf
64edfee10dd0c70b071427cfcbb8b5eb225daf149a6b4d42bebcc536380a9d753
a8b1e
sig: 02bc0f2728e2b8cd1c1b9873d4b7f5a62017430398165a6f8964842eaa19
c1de292207b74dc25ee0aa90493216d3fbf8e1b2947fd64335277b34767f987c4
82c69262967c8a8aaf180a4006f456c804cdc7b92d956a351ad89703cc76f69ed
45f24d68e1ae0361479e0f6faf10c3b1582de2dcd2af432d57c0c89c8efb1cf3a
c5f991fe9c4f0ad24473939b053674a2582518b4bd57da109f4f37bc91a2f806e
82bb2b80d486d0694e663992c9517c946607b978f557bbb769d4cd836d693c77d
a480cd89b916e5e4190f317711d9c7e64528a314a14bf0b9256f4c60e9ddb5505
83c21755ab882bdfdf22dc840249389b1e0a2189f58e19b41c5f313cddce29

// 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:
info:
eprime: 2ed579fcdf2d328ebc686c52ccaec247018832acd530a2ac72c0ec2b9
2db5d6bd578e91b6341c1021142b45b9e6e5bf031f3dd62226ec4a0f9ef99e45d
d9ccd60aa60a0c59aac271a8caf9ee68a9d9ff281367dae09d588d3c7bca7f18d
e48b6981bbc729c4925c65e4b2a7f054facbb7e5fc6e4c6c10110c62ef0b94eec
397b
r: 35deeb769ae3dce60033cbaeceeec511cd980307f53810c1b33934eeee
c194283834419fb190881213cfc0d91ff6307862ce143586ad9580057e3af6bb4
```

a405075ed9abf1f3b5a5cfb0c7fc59b952401cb28c04f9f85a9a3708fd51c74af
a1a1028a821beb2f8165678657f0d2effb7a1add5421216f3c50253a6f8d0d8ed
492f947d58b42527a2249b08e5ab05c25362ad8112bdea931544711b2a3d30e61
e2c8a130ea7f90fe915b5c3adaa24a6c300c23d8f670d330b592a7c05f7588324
688ae10e06290ab160096ce132a5ed220d2fbe6b4722cc30e05316b30f500b2ac
15038555289ea405025740a1d2a3098d34d094b566d0b973e661d855fb90be3c
salt: df4fbdf415184c20fad0418f27c35974db8c321e84c54b21e1e2619dbfa
0ad70db62c01783ffe796e8474596d7eb3fd8
blind_msg: ba562cba0e69070dc50384456391defa410d36fa853fd235902f
f5d015d688a44def6b6a7e71a69bff8ee510f5a9aa44e9afddd3e766f2423b3fc
783fd1a9ab618586110987c1b3ddce62d25cae500aa92a6b886cb609829d06e67
fbf28fbbf3ee7d5cc125481dd002b908097732e0df06f288cc6eb54565f8153d4
80085b56ab6cb5801b482d12f50558eb3cb0eb7a4ff8fcc54d4d7fcc2f8913a40
1ae1d1303ead7964f2746e4804e2848bba87f53cf1412afedc82d9c383dd095e0
eb6f90cc74bc4bb5ea7529ded9cde2d489575d549b884379abe6d7b71969e6a9c
09f1963d2719eefccd5f2a407845961ccc1fa580a93c72902b2499d96f89e6c53
fc888
blind_sig: 280c5934022fd17f7f810d4f7adf1d29ced47d098834411d6721
63cc793bcaad239d07c4c45048a682995950ce84703064cd8c16d6f2579f7a65b
66c274faccc6c73c9d299dcf35c96338c9b81af2f93554a78528551e04be931c8
502ee6a21ef65d1fa3cd049a993e261f85c841b75857d6bf02dd4532e14702f8f
5e1261f7543535cdf9379243b5b8ca5cd69d2576276a6c25b78ab7c69d2b0c568
eb57cf1731983016dece5b59e75301ca1a148154f2592c8406fee83a434f7b319
2649c5be06000866ff40bf09b558c7af4bbb9a79d5d13151e7b6e602e30c4ab70
bbbce9c098c386e51b98aefab67b8efc03f048210a785fd538ee6b75ecd484c13
40d91
sig: b7d45ec4db11f9b74a6b33806e486f7ee5f87c4fa7c57d08caf0ca6d3ba5
5e66bf0769c84b9187b9a86e49ba0cb58348f01156ac5bc2e9570fe0a8c33d0ad
049a965aeb2a8d8a3cbb30f89a3da6732a9bb3d9415141be4e9052f49d422301a
9cfce49947db7d52a1c620b7106ae43afbcb7cb29b9c215e0c2b4cf8d62db6722
4dd3de9f448f7b6607977c608595d29380b591a2bff2dff57ea2c77e9cdf69c18
21ff183a7626d45bbe1197767ac577715473d18571790b1cf59ee35e64362c826
246ae83923d749117b7ec1b4478ee15f990dc200745a45f175d23c8a13d2dbe58
b1f9d10db71917708b19eeeab230fe6026c249342216ee785d9422c3a8dc89

## Authors' Addresses

Ghous A. Amjad
Google

Email: gamjad@google.com

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