WG Working Group
Internet-Draft
Intended status: Informational
Expires: 18 September 2025

F. Denis
Fastly Inc.
E. Eaton
University of Waterloo
T. Lepoint

C. A. Wood
Cloudflare, Inc.
17 March 2025

Key Blinding for Signature Schemes
draft-irtf-cfrg-signature-key-blinding-08

Abstract

This document describes extensions to existing digital signature schemes for key blinding. The core property of signing with key blinding is that a blinded public key and all signatures produced using the blinded key pair are independent of the unblinded key pair. Moreover, signatures produced using blinded key pairs are indistinguishable from signatures produced using unblinded key pairs. This functionality has a variety of applications, including Tor onion services and privacy-preserving airdrop for bootstrapping cryptocurrency systems.

About This Document

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

The latest revision of this draft can be found at https://cfrg.github.io/draft-irtf-cfrg-signature-key-blinding/draft-irtf-cfrg-signature-key-blinding.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-irtf-cfrg-signature-key-blinding/.

Discussion of this document takes place on the CFRG Working Group mailing list (mailto:cfrg@irtf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg/. Subscribe at https://www.ietf.org/mailman/listinfo/cfrg/.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-signature-key-blinding.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

This Internet-Draft will expire on 18 September 2025.

Copyright Notice

Table of Contents

1. Introduction

Digital signature schemes allow a signer to sign a message using a private signing key and produce a digital signature such that anyone can verify the digital signature over the message with the public verification key corresponding to the signing key. Digital signature schemes typically consist of three functions:

* KeyGen: A function for generating a private signing key skS and the corresponding public verification key pkS.

* Sign(skS, msg): A function for signing an input message msg using a private signing key skS, producing a digital signature sig.

* Verify(pkS, msg, sig): A function for verifying the digital signature sig over input message msg against a public verification key pkS, yielding true if the signature is valid and false otherwise.

In some applications, it's useful for a signer to produce digital signatures using the same long-term private signing key such that a verifier cannot link any two signatures to the same signer. In other words, the signature produced is independent of the long-term private-signing key, and the public verification key for verifying the signature is independent of the long-term public verification key. This type of functionality has a number of practical applications, including, for example, in the Tor onion services protocol [TORDIRECTORY] and privacy-preserving airdrop for bootstrapping cryptocurrency systems [AIRDROP]. It is also necessary for a variant of the Privacy Pass issuance protocol [RATELIMITED].

One way to accomplish this is by signing with a private key which is a function of the long-term private signing key and a freshly chosen blinding key, and similarly by producing a public verification key which is a function of the long-term public verification key and same blinding key. A signature scheme with this functionality is referred to as signing with key blinding.

A signature scheme with key blinding aims to achieve unforgeability and unlinkability. Informally, unforgeability means that one cannot produce a valid (message, signature) pair for any blinding key without access to the private signing key. Similarly, unlinkability means that one cannot distinguish between two signatures produced from two separate key signing keys, and two signatures produced from the same signing key but with different blinding keys.

This document describes extensions to EdDSA [RFC8032] and ECDSA [ECDSA] to enable signing with key blinding. Security analysis of these extensions is currently underway; see Section 8 for more details.

This functionality is also possible with other signature schemes, including some post-quantum signature schemes [ESS21], though such extensions are not specified here.

1.1. DISCLAIMER

This document is a work in progress and is still undergoing security analysis. As such, it MUST NOT be used for real world applications. See Section 8 for additional information.

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.

The following terms are used throughout this document to describe the blinding modification.

* G: The standard base point.

* sk: A signature scheme private key. For EdDSA, this is a a randomly generated private seed of length 32 bytes or 57 bytes according to [RFC8032], Section 5.1.5 or [RFC8032], Section 5.2.5, respectively. For [ECDSA], sk is a random scalar in the prime-order elliptic curve group.

* pk(sk): The public key corresponding to the private key sk.

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

* ScalarMult(pk, k): Multiply the public key pk by scalar k, producing a new public key as a result.

* ModInverse(x, L): Compute the multiplicative inverse of x modulo L.

In pseudocode descriptions below, integer multiplication of two scalar values is denoted by the * operator. For example, the product of two scalars x and y is denoted as x * y.

3. Key Blinding

At a high level, a signature scheme with key blinding allows signers to blind their private signing key such that any signature produced with a private signing key and blinding key is independent of the private signing key. Similar to the signing key, the blinding key is also a private key. For example, the blind is a 32-byte or 57-byte random seed for Ed25519 or Ed448 variants, respectively, whereas the blind for ECDSA over P-256 is a random value in the scalar field for the P-256 elliptic curve group.

In more detail, consider first the basic digital signature syntax, which is a combination of the following functionalities:

* KeyGen: A function for generating a private and public key pair (skS, pkS).

* Sign(skS, msg): A function for signing a message msg with the given private key skS, producing a signature sig.

* Verify(pkS, msg, sig): A function for verifying a signature sig over message msg against the public key pkS, which returns 1 upon success and 0 otherwise.

Key blinding introduces three new functionalities for the signature scheme syntax:

* BlindKeyGen: A function for generating a private blind key.

* BlindPublicKey(pkS, bk, ctx): Blind the public verification key pkS using the private blinding key bk and context ctx, yielding a blinded public key pkR.

* BlindKeySign(skS, bk, ctx, msg): Sign a message msg using the private signing key skS with the private blind key bk and context ctx.

For a given bk produced from BlindKeyGen, key pair (skS, pkS) produced from KeyGen, a context value ctx, and message msg, correctness requires the following equivalence to hold with overwhelming probability:

```
Verify(BlindKeySign(skS, bk, ctx), msg, BlindPublicKey(pkS, bk, ctx)) = 1
```

Security requires that signatures produced using BlindKeySign are unlinkable from signatures produced using the standard signature generation function with the same private key.

When the context value is known, a signature scheme with key blinding may also support the ability to unblind public keys. This is represented with the following function.

* UnblindPublicKey(pkR, bk, ctx): Unblind the public verification key pkR using the private blinding key bk and context ctx.

For a given bk produced from BlindKeyGen, (skS, pkS) produced from KeyGen, and context value ctx, correctness of this function requires the following equivalence to hold:

```
UnblindPublicKey(BlindPublicKey(pkS, bk, ctx), bk, ctx) = pkS
```

Considerations for choosing context strings are discussed in Section 7.

4. Ed25519ph, Ed25519ctx, and Ed25519

This section describes implementations of BlindPublicKey, UnblindPublicKey, and BlindKeySign as modifications of routines in [RFC8032], Section 5.1. BlindKeyGen invokes the key generation routine specified in [RFC8032], Section 5.1.5 and outputs only the private key. This section assumes a context value ctx has been configured or otherwise chosen by the application.

4.1. BlindPublicKey and UnblindPublicKey

BlindPublicKey transforms a private blind bk into a scalar for the edwards25519 group and then multiplies the target key by this scalar. UnblindPublicKey performs essentially the same steps except that it multiplies the target public key by the multiplicative inverse of the scalar, where the inverse is computed using the order of the group L, described in [RFC8032], Section 5.1.

More specifically, BlindPublicKey(pk, bk, ctx) works as follows.

1. Construct the blind_ctx as concat(bk, 0x00, ctx), where bk is a 32-byte octet string, hash the result using SHA-512(blind_ctx), and store the digest in a 64-octet large buffer, denoted b. Interpret the lower 32 bytes buffer as a little-endian integer, forming a secret scalar s. Note that this explicitly skips the buffer pruning step in [RFC8032], Section 5.1.

2. Perform a scalar multiplication ScalarMult(pk, s), and output the encoding of the resulting point as the public key.

UnblindPublicKey(pkR, bk, ctx) works as follows.

1. Compute the secret scalar s from bk and ctx as in BlindPublicKey.

2. Compute the sInv = ModInverse(s, L), where L is as defined in [RFC8032], Section 5.1.

3. Perform a scalar multiplication ScalarMult(pk, sInv), and output the encoding of the resulting point as the public key.

4.2. BlindKeySign

BlindKeySign transforms a private key bk into a scalar for the edwards25519 group and a message prefix to blind both the signing scalar and the prefix of the message used in the signature generation routine.

More specifically, BlindKeySign(skS, bk, msg) works as follows:

1. Hash the private key skS, 32 octets, using SHA-512. Let h denote the resulting digest. Construct the secret scalar s1 from the first half of the digest, and the corresponding public key A1, as described in [RFC8032], Section 5.1.5. Let prefix1 denote the second half of the hash digest, h[32],...,h[63].

2. Construct the blind_ctx as concat(bk, 0x00, ctx), where bk is a 32-byte octet string, hash the result using SHA-512(blind_ctx), and store the digest in a 64-octet large buffer, denoted b. Interpret the lower 32 bytes buffer as a little-endian integer, forming a secret scalar s2. Note that this explicitly skips the buffer pruning step in [RFC8032], Section 5.1.5. Let prefix2 denote the second half of the hash digest, b[32],...,b[63].

3. Compute the signing scalar s = s1 * s2 (mod L) and the signing public key A = ScalarMult(G, s).

4. Compute the signing prefix as concat(prefix1, prefix2).

5. Run the rest of the Sign procedure in [RFC8032], Section 5.1.6 from step (2) onwards using the modified scalar s, public key A, and string prefix.

5. Ed448ph and Ed448

This section describes implementations of BlindPublicKey, UnblindPublicKey, and BlindKeySign as modifications of routines in [RFC8032], Section 5.2. BlindKeyGen invokes the key generation routine specified in [RFC8032], Section 5.1.5 and outputs only the private key. This section assumes a context value ctx has been configured or otherwise chosen by the application.

5.1. BlindPublicKey and UnblindPublicKey

BlindPublicKey and UnblindPublicKey for Ed448ph and Ed448 are implemented just as these routines are for Ed25519ph, Ed25519ctx, and Ed25519, except that SHAKE256 is used instead of SHA-512 for hashing the secret blind context, i.e., the concatenation of blind key bk and context ctx, to a 114-byte buffer (and using the lower 57-bytes for the secret), and the order of the edwards448 group L is as defined in [RFC8032], Section 5.2.1. Note that this process explicitly skips the buffer pruning step in [RFC8032], Section 5.2.5.

5.2. BlindKeySign

BlindKeySign for Ed448ph and Ed448 is implemented just as this routine for Ed25519ph, Ed25519ctx, and Ed25519, except in how the scalars (s1, s2), public keys (A1, A2), and message strings (prefix1, prefix2) are computed. More specifically, BlindKeySign(skS, bk, msg) works as follows:

1. Hash the private key skS, 57 octets, using SHAKE256(skS, 117). Let h1 denote the resulting digest. Construct the secret scalar s1 from the first half of h1, and the corresponding public key A1, as described in [RFC8032], Section 5.2.5. Let prefix1 denote the second half of the hash digest, h1[57],...,h1[113].

2. Construct the blind_ctx as concat(bk, 0x00, ctx), where bk is a 57-byte octet string, hash the result using SHAKE256(blind_ctx, 117), and store the digest in a 117-octet digest, denoted h2. Interpret the lower 57 bytes buffer as a little-endian integer, forming a secret scalar s2. Note that this explicitly skips the buffer pruning step in [RFC8032], Section 5.2. Let prefix2 denote the second half of the hash digest, h2[57],...,h2[113].

3. Compute the signing scalar s = s1 * s2 (mod L) and the signing public key A = ScalarMult(A1, s2).

4. Compute the signing prefix as concat(prefix1, prefix2).

5. Run the rest of the Sign procedure in [RFC8032], Section 5.2.6 from step (2) onwards using the modified scalar s, public key A, and string prefix.

6. ECDSA

[[DISCLAIMER: Multiplicative blinding for ECDSA is known to be NOT be SUF-CMA-secure in the presence of an adversary that controls the blinding value. [MSMHI15] describes this in the context of related-key attacks. This variant may likely be removed in followup versions of this document based on further analysis.]]

This section describes implementations of BlindPublicKey, UnblindPublicKey, and BlindKeySign as functions implemented on top of an existing [ECDSA] implementation. BlindKeyGen invokes the key generation routine specified in [ECDSA] and outputs only the private key. In the descriptions below, let p be the order of the corresponding elliptic curve group used for ECDSA. For example, for P-256, p = 115792089210356248762697446949407573529996955224135760342422259061068512044369.

This section assumes a context value ctx has been configured or otherwise chosen by the application.

6.1. BlindPublicKey and UnblindPublicKey

BlindPublicKey multiplies the public key pkS by an augmented private key bk yielding a new public key pkR. UnblindPublicKey inverts this process by multiplying the input public key by the multiplicative inverse of the augmented bk. Augmentation here maps the private key bk to another scalar using hash_to_field as defined in Section 5 of [H2C], with DST set to "ECDSA Key Blind", L set to the value corresponding to the target curve, e.g., 48 for P-256 and 72 for P-384, expand_message_xmd with a hash function matching that used for the corresponding digital signature algorithm, and prime modulus equal to the order p of the corresponding curve. Letting HashToScalar denote this augmentation process, and blind_ctx = concat(bk, 0x00, ctx), BlindPublicKey and UnblindPublicKey are then implemented as follows:

```
BlindPublicKey(pk, bk, ctx)   = ScalarMult(pk, HashToScalar(blind_ctx))
UnblindPublicKey(pkR, bk, ctx) = ScalarMult(pkR, ModInverse(HashToScalar(blind_ctx), p))
```

6.2. BlindKeySign

BlindKeySign transforms the signing key skS by the private key bk along with context ctx into a new signing key, skR, and then invokes the existing ECDSA signing procedure. More specifically, skR = skS * HashToScalar(blind_ctx) (mod p), where blind_ctx = concat(bk, 0x00, ctx).

7. Application Considerations

Choice of the context string ctx is application-specific. For example, in Tor [TORDIRECTORY], the context string is set to the concatenation of the long-term signer public key and an integer epoch. This makes it so that unblinding a blinded public key requires knowledge of the long-term public key as well as the blinding key. Similarly, in a rate-limited version of Privacy Pass [RATELIMITED], the context is empty, thereby allowing unblinding by anyone in possession of the blinding key.

Applications are RECOMMENDED to choose context strings that are distinct from other protocols as a way of enforcing domain separation. See Section 2.2.5 of [HASH-TO-CURVE] for additional discussion around the construction of suitable domain separation values.

8. Security Considerations

The signature scheme extensions in this document aim to achieve unforgeability and unlinkability. Informally, unforgeability means that one cannot produce a valid (message, signature) pair for any blinding key without access to the private signing key. Similarly, unlinkability means that one cannot distinguish between two signatures produced from two independent key signing keys, and two signatures produced from the same signing key but with different blinds. Security analysis of the extensions in this document with respect to these two properties is currently underway. See [CGHKS23] for more detailed discussion of signature extensions with these properties.

Preliminary analysis has been done for a variant of these extensions used for identity key blinding routine used in Tor’s Hidden Service feature [TORBLINDING]. Further analysis exists in [ELW23], which demonstrates that the extensions in this specification for EdDSA and ECDSA both achieve the desired security properties.

The constructions in this document, as well as the analysis in [ELW23], assume that both the signing and blinding keys are private, and, as such, not controlled by an attacker. [MSMHI15] demonstrate

that ECDSA with attacker-controlled multiplicative blinding for producing related keys can be abused to produce forgeries. In particular, if an attacker can control the private blinding key used in BlindKeySign, they can construct a forgery over a different message that validates under a different public key. One mitigation to this problem is to change BlindKeySign such that the signature is computed over the input message as well as the blind public key. However, this would require verifiers to treat both the blind public key and message as input to their verification interface. The construction in Section 6 does not require this change. However, further analysis is needed to determine whether or not this construction is safe.

9. IANA Considerations

This document has no IANA actions.

10. Test Vectors

This section contains test vectors for a subset of the signature schemes covered in this document.

10.1. Ed25519 Test Vectors

This section contains test vectors for Ed25519 as described in [RFC8032]. Each test vector lists the serialized signing key (skS), blind key (bk), and public key (pkS) encoded has hexadecimal strings; skS and bk are serialized as little-endian 32-byte encoding of the scalar value with the top three bits set to zero, whereas pkS is serialized as described in Section 5.1.2 of [RFC8032]. Each test vector also includes the blinded public key (pkR) computed from skS and bk, serialized similarly to pkS and encoded as a hexadecimal string. Finally, each vector includes the message and signature values, each encoded as hexadecimal strings. The signature is encoded as specified in Section 5.1.6 of [RFC8032].

```
// Randomly generated private key and blind seed, empty context

skS: d142b3b1d532b0a516353a0746a6d43a86cee8efaf6b14ae85c2199072f47d93
pkS: cd875d3f46a8e8742cf4a6a9f9645d4153a394a5a0a8028c9041cd455d093cd5
bk: bb58c768d9b16571f553efd48207e64391e16439b79fe9409e70b38040c81302
pkR: 666443ce8f03fa09240db73a584efad5462ffe346b14fd78fb666b25db29902f
message: 68656c6c6f20776f726c64
context:
signature: 5458111c708ce05cb0a1608b08dc649937dc22cf1da045eb866f2face50be
930e79b44d57e5215a82ac227bdccccca52bfe509b96efe8e723cb42b5f14be5f0e
```

```
// Randomly generated private key seed and zero blind seed, empty context

skS: aa69e9cb50abf39b05ebc823242c4fd13ccadd0dadc1b45f6fcbf7be4f30db5d
pkS: 5c9a9e271f204c931646aa079e2e66f0783ab3d29946eff37bd3b569e9c8e009
bk: 0000000000000000000000000000000000000000000000000000000000000000
pkR: 23eb5eccb9448ee8403c36595ccfd5edd7257ae70da69aa22282a0a7cd97e443
message: 68656c6c6f20776f726c64
context:
signature: 4e9f3ad2b14cf2f9bbf4b88a8832358a568bd69368b471dfabac594e8a8b3
3ab54978ecf902560ed754f011186c4c4dda65d158b96c1e6b99a8e150a26e51e03

// Randomly generated private key and blind seed, non-empty context

skS: d1e5a0f806eb3c491566cef6d2d195e6bbf0a54c9de0e291a7ced050c63ea91c
pkS: 8b37c949d39cddf4d2a0fc0da781ea7f85c7bfbdfeb94a3c9ecb5e8a3c24d65f
bk: 05b235297dff87c492835d562c6e03c0f36b9c306f2dcb3b5038c2744d4e8a70
pkR: 019b0a06107e01361facdad39ec16a9647c86c0086bc38825eb664b97d9c514d
message: 68656c6c6f20776f726c64
context:
d6bbaa0646f5617d3cbd1e22ef05e714d1ec7812efff793999667648b2cc54bc
signature: f54214acb3c695c46b1e7aa2da947273cb19ec33d8215dde0f43a8f7250fe
bb508f4a5007e3c96be6402074ec843d40358a281ff969c66c1724016208650dd09

// Randomly generated private key seed and zero blind seed, non-empty context

skS: 89e3e3acef6a6c2d9b7c062199bf996f9ae96b662c73e2b445636f9f22d5012e
pkS: 3f667a2305a8baf328a1d8e9ed726f278229607d28fb32d9933da7379947ac44
bk: 0000000000000000000000000000000000000000000000000000000000000000
pkR: 90a543dd29c6e6cd08ef85c43618f2d314139db5baed802383cf674310294e40
message: 68656c6c6f20776f726c64
context:
802def4d21c7c7d0fa4b48af5e85f8ebfc4119a04117c14d961567eaef2859f2
signature: ce305a0f40a3270a84d2d9403617cdb89b7b4edf779b4de27f9acaadf1716
84b162e752c95f17b16aaca7c2662e69ba9696bdd230a107ecab973886e8d5bf00e
```

10.2. ECDSA(P-384, SHA-384) Test Vectors

This section contains test vectors for ECDSA with P-384 and SHA-384, as described in [ECDSA]. Each test vector lists the serialized signing key (skS), blind key (bk), and public key (pkS) encoded has hexadecimal strings; skS and bk are serialized using the Field-Element-to-Octet-String conversion according to [SEC1], whereas pkS is serialized using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SEC1]. Each test vector also includes the blinded public key (pkR) computed from skS and bk, serialized similarly to pkS and encoded as a hexadecimal string. Finally, each vector includes the message and signature values, each encoded as hexadecimal strings. The signature value is serialized as the

concatenation of scalars (r, s), each serialized as skS and bk, and encoded as a hexadecimal string.

```
// Randomly generated signing and blind private keys, empty context

skS: fcc8217ec4c89862d069a6679026c8042a74a513ba5b4a63da58488643132afaf35
9c3645dcc99c11862d9606370b9b7
pkS: 02582e4108018f9657f8bb55192838ff057442c8f7dc265f195dc1e4aa2cff2ec10
e2f2220dbeb300125d46b00dff747f1
bk: 1d3b48eec849b9d0e7376be1eca90369663939d140a8f3418ebc2221159402647a9e
283a78694377915b2894bc38cfe5
pkR: 03031c9914e4aa550605ded5c8b2604a2910c7c4d7e1e8608d81152a2ed3b8eb85a
c8c7896107c91875090b651f43d2f31
message: 68656c6c6f20776f726c64
context:
signature: 0ca279fba24a47ef2dded3f3171f805779d41ff0c3b13af260977d26f9df8
a0993591b34e84f954149a478408abc685cb88ca32e482ffb9ea2f377ac949cb37468f18
4b8f03ce4c7da06c024a38e3d8f2a9eea84493288627a13f317cc6d8457

// Randomly generated signing and blind private keys, non-empty context

skS: 5f9ed9f16ac74cb510689321cbd6a0a9602f50a96cb17ff479ec46fff130afcd9fe
d3766c6d98fe4b4f1c2fa275f58ed
pkS: 03e690b68b39c0bfb0be6a7f7f0ab49a930437b427dbf588c7acbf3fc8e3e221c83
03e2d38c7bfe735d2d8afaecfacec8c
bk: 7c65bba8e98f1f75eb9748ccc4a85b7d5d9523522d02909958e0e2fc81693dbb4d10
460355eec3a3af54184ced97697a
pkR: 0280a5180793a1c8155face304fea93783514124cdf7f0fedab11da05289e192da3
6a9f0e3ab4544d75f8eaa8ef9987554
message: 68656c6c6f20776f726c64
context:
327a0a52fa1c01d376cfc259925555920d89f15b509bb84e7385ff7207dcb93d
signature: 240e49a4dc681e3cedb241f2cf97f7c86f215902c03e38838e1d23d127c61
debca8af590ebb0fd7f1dd58a51a63aa45e5991fda32da0e7e9bb56b9374be6fed60c672
2de2689f6a969af5c78b78e5dcc353d8a47a71f337586f737b020e541c1
```

11. References

11.1. Normative References

[ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry - The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.

[HASH-TO-CURVE] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in

Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[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>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/rfc/rfc8032>.

[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>.

11.2. Informative References

[AIRDROP] Wahby, R. S., Boneh, D., Jeffrey, C., and J. Poon, "An airdrop that preserves recipient privacy", n.d., <https://eprint.iacr.org/2020/676.pdf>.

[CGHKS23] Celi, S., Griffy, S., Hanzlik, L., Perez Kempner, O., and D. Slamanig, "SoK: Signatures With Randomizable Keys", <https://eprint.iacr.org/2023/1524>.

[ELW23] Eaton, E., Lepoint, T., and C. A. Wood, "Security Analysis of Signature Schemes with Key Blinding", <https://eprint.iacr.org/2023/380>.

[ESS21] Eaton, E., Stebila, D., and R. Stracovsky, "Post-Quantum Key-Blinding for Authentication in Anonymity Networks", 2021, <https://eprint.iacr.org/2021/963>.

[H2C] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[MSMHI15] Morita, H., Schuldt, J., Matsuda, T., Hanaoka, G., and T. Iwata, "On the Security of the Schnorr Signature Scheme and DSA Against Related-Key Attacks", Springer International Publishing, Lecture Notes in Computer Science pp. 20-35, DOI 10.1007/978-3-319-30840-1_2, ISBN ["9783319308395", "9783319308401"], 2016, <https://doi.org/10.1007/978-3-319-30840-1_2>.

[RATELIMITED] Hendrickson, S., Iyengar, J., Pauly, T., Valdez, S., and C. A. Wood, "Rate-Limited Token Issuance Protocol", Work in Progress, Internet-Draft, draft-privacypass-rate-limit-tokens-03, 6 July 2022, <https://datatracker.ietf.org/doc/html/draft-privacypass-rate-limit-tokens-03>.

[SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", <https://www.secg.org/sec1-v2.pdf>.

[TORBLINDING] Hopper, N., "Proving Security of Tor寰洰 Hidden Service Identity Blinding Protocol", 2013, <https://www-users.cse.umn.edu/~hoppernj/basic-proof.pdf>.

[TORDIRECTORY] "Tor directory protocol, version 3", n.d., <https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt>.

Acknowledgments

The authors would like to thank Dennis Jackson and Cathie Yun for helpful discussions and input that informed and improved the development of this draft.

Authors' Addresses

Frank Denis
Fastly Inc.
475 Brannan St
San Francisco,
United States of America
Email: fde@00f.net

Edward Eaton
University of Waterloo
200 University Av West
Waterloo
Canada
Email: ted@eeaton.ca

Tancrティde Lepoint
New York,
United States of America
Email: cfrg@tancre.de

Christopher A. Wood
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America
Email: caw@heapingbits.net