Workgroup: WG Working Group
Internet-Draft:
draft-irtf-cfrg-signature-key-blinding-02
Published: 2 August 2022
Intended Status: Informational
Expires: 3 February 2023
Authors: F. Denis, Fastly Inc.
E. Eaton, University of Waterloo
T. Lepoint
C. A. Wood, Cloudflare, Inc.

Key Blinding for Signature Schemes

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/.

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

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)) =
```

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
```

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.

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]. For EdDSA, further analysis is needed to ensure this is compliant with the signature algorithm described in [RFC8032].

The constructions in this document 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 private key and blind seeds, denoted skS and bk and encoded as hexadecimal strings, along with the public key pkS corresponding to skS encoded has hexadecimal strings according to [RFC8032], Section 5.1.2. Each test vector also includes the blinded public key pkR computed from skS and bk, denoted pkR and encoded has a hexadecimal string. Finally, each vector includes the message and signature values, each encoded as hexadecimal strings.

```
// Randomly generated private key and blind seed, empty context
skS: 63ac6c411cf72d9006b853db3458940fb1b5d690747abd8b1ccb73f0f5269837
pkS: 963d13e180030cfcf1891c10d3143b5cd3613780b943dfd9100f7d9bb31af2cd
pkR: 4ed06c22a58ef8e65d280f0970fd02f839083026b6116b0d65c2cbf3f519368c
message: 68656c6c6f20776f726c64
context:
signature: 3a2ada316be0e7162ae8cdcc6b35dda7ab4159296fd1b060cc809fdb55e56
23cf0af5550140eaff2bb99516986d270bbb6737e5c8661731e016923e998315e04

// Randomly generated private key seed and zero blind seed, empty contex
skS: 056f2668895cda2f89e8ddc3138910979982ab3135ee22d358e80c85cec4cdc7
pkS: e211f518d9361dc1e39ca572a10e45c7372ac990465b17d62fde42247c367fb7
pkR: d73a40d8806f08936ef0c425482d2fdf6424242f008854db74c6230eeb44e19c
message: 68656c6c6f20776f726c64
context:
signature: 89d007f1215595b14612217ace71d3ce28688ebf55e5151e97861eceb5b60
6a32d37c15afc31a9ad7ad7101d15b9228edfe9b9b25ddd42f442475f4317f47405

// Randomly generated private key and blind seed, non-empty context
skS: d9fecb86876468c4c244e567662b4ad061c795ad03cbfdcf95fd67d1cba836d3
pkS: d018ed0a1c47f8d530c58afc30bcf141c0d4766429fd53a1b69287867e169827
pkR: cfd1458e1f81ba8c59446180cea170f5f2ecd721d68d02c625449c8ce4a8ab28
message: 68656c6c6f20776f726c64
context:
2b79c03dd60967c954d3263ac32834692d6938c75fbc9a089ec855ca3a15ad40
signature: 27afdabf12ebc768863df1ee10db0408362132b56fe7a7fa84cc8b191200d
8cd8d8cd39f3698798f1a7e1a89c477699e2450c65edfbf65bf354ae7de45aa6e0e

// Randomly generated private key seed and zero blind seed, non-empty co
skS: d5c4c2f3fc43b8cceb6083b1db97c4dd0b9fca0773b14ed73066ad64d7d276df
pkS: 8552d8d4ffe3c7f94ee0cdc1e52598de3425439ed6161f8037bcce99d84c7953
pkR: aa148c1e6ceb8557aa89d85fb8d71e24cd4d0bc958f6526f3336e357679b77df
message: 68656c6c6f20776f726c64
context:
a9df0f21630248d1753e4a21ee2edcaa78609386134548a22696dd409cf1c2ac
signature: fe138ac61c020db62bfcdf70d181a4c6ee7d8015d4d577e55868bd86676bd
ecccea8db0da501e877ab58ab17fe043979eec7e467c68a1e690932dd5552ae4705
```

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 signing and blinding keys, denoted skS and bk, each serialized as a big-endian integers and encoded as hexadecimal strings. Each test vector also blinded public key pkR, encoded as compressed elliptic curve points according to [ECDSA]. Finally, each vector lists message and signature values, where the message is encoded as a hexadecimal string, and the signature value is serialized as the concatenation of scalars (r, s) and encoded as a hexadecimal string.

```
// Randomly generated signing and blind private keys, empty context
skS: cc09c66952c416956f78b73c8fd984f8bd69fa894fb08dd197be0a97dae1d781083
d8bcc4cca0aa906450c6b5e1b5cf3
pkS: 02091444dfde7de0623d8b94ba9ef8010756baf982b12db755d130c16fda97c4f95
6dd0f7b346fc3ef7245dfc76e1cacc4
bk: e49afba496c06344afa224480f823457863ac71e5f67c359ca1fbc42411754cc893c
c0bc10ff6d95363ab2e1c4154092
pkR: 03ae6ef617427a15bdc9dcba8a482f5f25aa45af6916edc8b51254304f393ee18b7
2fab54aae380426984ebfb7ef4045c8
message: 68656c6c6f20776f726c64
context:
signature: 6e31a96d811b0a271640e5dead87c8a5a0e9aaece4145464818bfcaa0ee9e
ea09c9178a59a4003800ae0a88cf2d3ae2811303f0acb0f77ce14ed8a2ad82d612af0ebe
b87c23047b7129ffcae4c2dd0f187e671e2e05a85972cd7e53de1529c45

// Randomly generated signing and blind private keys, non-empty context
skS: 90792d09edfc4afcb3a770b1d8582ed4bf3f3b3f751b90e5c8ac8ce1671c60475ee
0d390d0505e4b5bfb678ba9e665c9
pkS: 0275741cc339a46fc7ce24a0553eb3c2f2e83cf50dbb856ff3ae445d3a511f42749
c2f8510b0de9ac1b5deca9161ded522
bk: 9ff66325364badbef3d29bf4c955dbbe2be8cdaa2cc777f8badc066e171fb7b2df53
349278028da700eeabb745c045ef
pkR: 03782eed6bd50a9554989998850f88d91279ee865153d4be922488df39d614588df
94c70ed3494a7fcbd3fc057ea6ccf29
message: 68656c6c6f20776f726c64
context:
ff47734c8ccb40a369fdd2bdc34749e1d06feee27170b6452048594365e1c853
signature: fdc8e03668cbe9ed0146d0e2fbdcc5d494860f8017217c5044a0ff773af72
205eb4a20ffa5bbc9076cd5a43fd99d9b42a79a483a6be3a1b9f85b3180fd0e6c371ef06
6e2557ea5cf752eecc1f2a0ff67777eb01604039a92fe3d48d6991ebdc6
```

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>.

[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", Information Security and Cryptology - ICISC 2015 pp. 20-35, DOI 10.1007/978-3-319-30840-1_2, 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>.

[TORBLINDING] Hopper, N., "Proving Security of Tor’s 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 for helpful discussions that informed 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

Tancrède 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