

DRIP
Internet-Draft
Updates: 7401, 7343 (if approved)
Intended status: Standards Track
Expires: 14 November 2022

R. Moskowitz
HTT Consulting
S. Card
A. Wiethuechter
AX Enterprize, LLC
A. Gurtov
Linköping University
13 May 2022

DRIP Entity Tag (DET) for Unmanned Aircraft System Remote ID (UAS RID)
draft-ietf-drip-rid-26

Abstract

This document describes the use of Hierarchical Host Identity Tags (HHITs) as self-asserting IPv6 addresses and thereby a trustable identifier for use as the Unmanned Aircraft System Remote Identification and tracking (UAS RID).

This document updates RFC7401 and RFC7343.

Within the context of RID, HHITs will be called DRIP Entity Tags (DETs). HHITs self-attest to the included explicit hierarchy that provides registry (via, e.g., DNS, EPP) discovery for 3rd-party identifier attestation.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 14 November 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	3
1.1. HHIT Statistical Uniqueness different from UUID or X.509 Subject	4
2. Terms and Definitions	4
2.1. Requirements Terminology	4
2.2. Notations	4
2.3. Definitions	4
3. The Hierarchical Host Identity Tag (HHIT)	6
3.1. HHIT Prefix for RID Purposes	7
3.2. HHIT Suite IDs	7
3.2.1. HDA custom HIT Suite IDs	8
3.3. The Hierarchy ID (HID)	8
3.3.1. The Registered Assigning Authority (RAA)	8
3.3.2. The Hierarchical HIT Domain Authority (HDA)	9
3.4. Edward-Curve Digital Signature Algorithm for HHITs	9
3.4.1. HOST_ID	10
3.4.2. HIT_SUITE_LIST	11
3.5. ORCHIDs for Hierarchical HITs	11
3.5.1. Adding Additional Information to the ORCHID	12
3.5.2. ORCHID Encoding	13
3.5.3. ORCHID Decoding	15
3.5.4. Decoding ORCHIDs for HIPv2	15
4. Hierarchical HITs as DRIP Entity Tags	15
4.1. Nontransferability of DETs	16
4.2. Encoding HHITs in CTA 2063-A Serial Numbers	16
4.3. Remote ID DET as one Class of Hierarchical HITs	17
4.4. Hierarchy in ORCHID Generation	17
4.5. DRIP Entity Tag (DET) Registry	18
4.6. Remote ID Authentication using DETs	18
5. DRIP Entity Tags (DETs) in DNS	18
6. Other UTM Uses of HHITs Beyond DET	20
7. Summary of Addressed DRIP Requirements	20
8. IANA Considerations	20
8.1. New Well-Known IPv6 prefix for DETs	20
8.2. New IANA DRIP Registry	21
8.3. IANA CGA Registry Update	22
8.4. IANA HIP Registry Updates	22

8.5.	IANA IPSECKEY Registry Update	23
9.	Security Considerations	23
9.1.	DET Trust in ASTM messaging	25
9.2.	DET Revocation	25
9.3.	Privacy Considerations	26
9.4.	Collision Risks with DETs	27
10.	References	27
10.1.	Normative References	27
10.2.	Informative References	28
Appendix A.	EU U-Space RID Privacy Considerations	31
Appendix B.	The 14/14 HID split	31
Appendix C.	Calculating Collision Probabilities	33
Acknowledgments	33
Authors' Addresses	34

1. Introduction

DRIP Requirements [RFC9153] describe an Unmanned Aircraft System Remote ID (UAS ID) as unique (ID-4), non-spoofable (ID-5), and identify a registry where the ID is listed (ID-2); all within a 19-character identifier (ID-1).

This document describes (per Section 3 of [drip-architecture]) the use of Hierarchical Host Identity Tags (HHITs) (Section 3) as self-asserting IPv6 addresses and thereby a trustable identifier for use as the UAS Remote ID. HHITs add explicit hierarchy to the 128-bit HITs, enabling DNS HHIT queries (Host ID for authentication, e.g., [drip-authentication]) and for Extensible Provisioning Protocol (EPP) Registrar discovery [RFC9224] for 3rd-party identification attestation (e.g., [drip-authentication]).

This addition of hierarchy to HITs is an extension to [RFC7401] and requires an update to [RFC7343]. As this document also adds EdDSA (Section 3.4) for Host Identities (HIs), a number of Host Identity Protocol (HIP) parameters in [RFC7401] are updated, but these should not be needed in a DRIP implementation that does not use HIP.

HHITs as used within the context of Unmanned Aircraft System (UAS) are labeled as DRIP Entity Tags (DETs). Throughout this document HHIT and DET will be used appropriately. HHIT will be used when covering the technology, and DET for their context within UAS RID.

Hierarchical HITs provide self-attestation of the HHIT registry. A HHIT can only be in a single registry within a registry system (e.g., EPP and DNS).

Hierarchical HITs are valid, though non-routable, IPv6 addresses [RFC8200]. As such, they fit in many ways within various IETF technologies.

1.1. HHIT Statistical Uniqueness different from UUID or X.509 Subject

HHITs are statistically unique through the cryptographic hash feature of second-preimage resistance. The cryptographically-bound addition of the hierarchy and a HHIT registration process [drip-registries] provide complete, global HHIT uniqueness. This contrasts with using general identifiers (e.g., a Universally Unique IDentifiers (UUID) [RFC4122] or device serial numbers) as the subject in an X.509 [RFC5280] certificate.

In a multi-Certificate Authority (multi-CA) PKI alternative to HHITs, a Remote ID as the Subject (Section 4.1.2.6 of [RFC5280]) can occur in multiple CAs, possibly fraudulently. CAs within the PKI would need to implement an approach to enforce assurance of the uniqueness achieved with HHITs.

2. Terms and Definitions

2.1. Requirements Terminology

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.

2.2. Notations

| Signifies concatenation of information - e.g., X | Y is the concatenation of X and Y.

2.3. Definitions

This document uses the terms defined in Section 2.2 of [RFC9153]. The following new terms are used in the document:

cSHAKE (The customizable SHAKE function [NIST.SP.800-185]):
Extends the SHAKE [NIST.FIPS.202] scheme to allow users to customize their use of the SHAKE function.

HDA (HHIT Domain Authority):
The 14-bit field that identifies the HHIT Domain Authority under a Registered Assigning Authority (RAA).

HHIT

Hierarchical Host Identity Tag. A HIT with extra hierarchical information not found in a standard HIT [RFC7401].

HI

Host Identity. The public key portion of an asymmetric key pair as defined in [RFC9063].

HID (Hierarchy ID):

The 28-bit field providing the HIT Hierarchy ID.

HIP (Host Identity Protocol)

The origin [RFC7401] of HI, HIT, and HHIT.

HIT

Host Identity Tag. A 128-bit handle on the HI. HITs are valid IPv6 addresses.

Keccak (KECCAK Message Authentication Code):

The family of all sponge functions with a KECCAK-f permutation as the underlying function and multi-rate padding as the padding rule. It refers in particular to all the functions referenced from [NIST.FIPS.202] and [NIST.SP.800-185].

KMAC (KECCAK Message Authentication Code [NIST.SP.800-185]):

A Pseudo Random Function (PRF) and keyed hash function based on KECCAK.

RAA (Registered Assigning Authority):

The 14-bit field identifying the business or organization that manages a registry of HDAs.

RVS (Rendezvous Server):

A Rendezvous Server such as the HIP Rendezvous Server for enabling mobility, as defined in [RFC8004].

SHAKE (Secure Hash Algorithm KECCAK [NIST.FIPS.202]):

A secure hash that allows for an arbitrary output length.

XOF (eXtendable-Output Function [NIST.FIPS.202]):

A function on bit strings (also called messages) in which the output can be extended to any desired length.

3. The Hierarchical Host Identity Tag (HHIT)

The Hierarchical HIT (HHIT) is a small but important enhancement over the flat Host Identity Tag (HIT) space, constructed as an Overlay Routable Cryptographic Hash Identifier (ORCHID) [RFC7343]. By adding two levels of hierarchical administration control, the HHIT provides for device registration/ownership, thereby enhancing the trust framework for HITs.

The 128-bit HHITs represent the HI in only a 64-bit hash, rather than the 96 bits in HITs. 4 of these 32 freed up bits expand the Suite ID to 8 bits, and the other 28 bits are used to create a hierarchical administration organization for HIT domains. Hierarchical HIT construction is defined in Section 3.5. The input values for the Encoding rules are described in Section 3.5.1.

A HHIT is built from the following fields (Figure 1):

- * p = an IPV6 prefix (max 28 bit)
- * 28-bit Hierarchy ID (HID) which provides the structure to organize HITs into administrative domains. HIDs are further divided into two fields:
 - 14-bit Registered Assigning Authority (RAA) (Section 3.3.1)
 - 14-bit Hierarchical HIT Domain Authority (HDA) (Section 3.3.2)
- * 8-bit HHIT Suite ID (HHSI)
- * ORCHID hash (96 - prefix length - 8 for HHIT Suite ID, e.g., 64)
See Section 3.5 for more details.

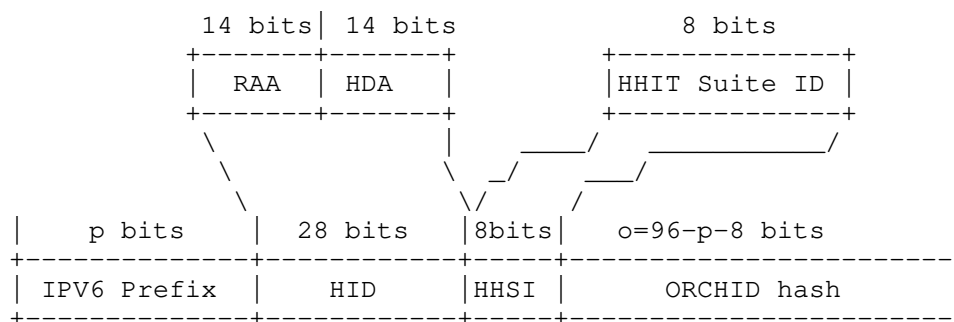


Figure 1: HHIT Format

The Context ID (generated with openssl rand) for the ORCHID hash is:

Context ID := 0x00B5 A69C 795D F5D5 F008 7F56 843F 2C40

Context IDs are allocated out of the namespace introduced for Cryptographically Generated Addresses (CGA) Type Tags [RFC3972].

3.1. HHIT Prefix for RID Purposes

The IPv6 HHIT prefix MUST be distinct from that used in the flat-space HIT as allocated in [RFC7343]. Without this distinct prefix, the first 4 bits of the RAA would be interpreted as the HIT Suite ID per HIPv2 [RFC7401].

Initially, for DET use, one 28-bit prefix should be assigned out of the IANA IPv6 Special Purpose Address Block ([RFC6890]).

HHIT Use	Bits	Value
DET	28	TBD6 (suggested value 2001:30::/28)

Other prefixes may be added in the future either for DET use or other applications of HHITs. For a prefix to be added to the registry in Section 8.2, its usage and HID allocation process have to be publicly available.

3.2. HHIT Suite IDs

The HHIT Suite IDs specify the HI and hash algorithms. These are a superset of the 4/8-bit HIT Suite ID as defined in Section 5.2.10 of [RFC7401].

The HHIT values of 1 - 15 map to the basic 4-bit HIT Suite IDs. HHIT values of 17 - 31 map to the extended 8-bit HIT Suite IDs. HHIT values unique to HHIT will start with value 32.

As HHIT introduces a new Suite ID, EdDSA/cSHAKE128, and since this is of value to HIPv2, it will be allocated out of the 4-bit HIT space and result in an update to HIT Suite IDs. Future HHIT Suite IDs may be allocated similarly, or may come out of the additional space made available by going to 8 bits.

The following HHIT Suite IDs are defined:

HHIT Suite	Value
RESERVED	0
RSA,DSA/SHA-256	1 [RFC7401]
ECDSA/SHA-384	2 [RFC7401]
ECDSA_LOW/SHA-1	3 [RFC7401]
EdDSA/cSHAKE128	TBD3 (suggested value 5) (RECOMMENDED)

3.2.1. HDA custom HIT Suite IDs

Support for 8-bit HHIT Suite IDs allows for HDA custom HIT Suite IDs. These will be assigned values greater than 15 as follows:

HHIT Suite	Value
HDA Private Use 1	TBD4 (suggested value 254)
HDA Private Use 2	TBD5 (suggested value 255)

These custom HIT Suite IDs, for example, may be used for large-scale experimenting with post quantum computing hashes or similar domain specific needs. Note that currently there is no support for domain-specific HI algorithms.

They should not be used to create a "de facto standardization". Section 8.2 states that additional Suite IDs can be made through IETF Review.

3.3. The Hierarchy ID (HID)

The Hierarchy ID (HID) provides the structure to organize HITs into administrative domains. HIDs are further divided into two fields:

- * 14-bit Registered Assigning Authority (RAA)
- * 14-bit Hierarchical HIT Domain Authority (HDA)

The rationale for the 14/14 HID split is described in Appendix B.

The two levels of hierarchy allows for CAAs to have at least one RAA for their National Air Space (NAS). Within its RAA(s), the CAAs can delegate HDAs as needed. There may be other RAAs allowed to operate within a given NAS; this is a policy decision of each CAA.

3.3.1. The Registered Assigning Authority (RAA)

An RAA is a business or organization that manages a registry of HDAs. For example, the Federal Aviation Authority (FAA) or Japan Civil Aviation Bureau (JCAB) could be an RAA.

The RAA is a 14-bit field (16,384 RAAs). The management of this space is further elaborated in [drip-registries]. An RAA MUST provide a set of services to allocate HDAs to organizations. It SHOULD have a public policy on what is necessary to obtain an HDA. The RAA need not maintain any HIP related services. It MUST maintain a DNS zone minimally for discovering HIP RVS servers for the HID. The zone delegation is also covered in [drip-registries].

As DETs under an administrative control may be used in many different domains (e.g., commercial, recreation, military), RAAs should be allocated in blocks (e.g. 16-19) with consideration on the likely size of a particular usage. Alternatively, different prefixes can be used to separate different domains of use of HHITs.

The RAA DNS zone within the UAS DNS tree may be a PTR for its RAA. It may be a zone in an HHIT specific DNS zone. Assume that the RAA is decimal 100. The PTR record could be constructed as follows:

```
100.hhit.arpa    IN PTR      raa.example.com.
```

Note that if the zone `hhit.arpa` is ultimately used, some registrar will need to manage this for all HHIT applications. Thus further thought will be needed in the actual zone tree and registration process [drip-registries].

3.3.2. The Hierarchical HIT Domain Authority (HDA)

An HDA may be an Internet Service Provider (ISP), UAS Service Supplier (USS), or any third party that takes on the business to provide UAS services management, HIP RVSSs or other needed services such as those required for HHIT and/or HIP-enabled devices.

The HDA is a 14-bit field (16,384 HDAs per RAA) assigned by an RAA is further elaborated in [drip-registries]. An HDA must maintain public and private UAS registration information and should maintain a set of RVS servers for UAS clients that may use HIP. How this is done and scales to the potentially millions of customers are outside the scope of this document, though covered in [drip-registries]. This service should be discoverable through the DNS zone maintained by the HDA's RAA.

An RAA may assign a block of values to an individual organization. This is completely up to the individual RAA's published policy for delegation. Such policy is out of scope.

3.4. Edwards-Curve Digital Signature Algorithm for HHITs

The Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032] is specified here for use as HIs per HIPv2 [RFC7401].

The intent in this document is to add EdDSA as a HI algorithm for DETs, but doing so impacts the HIP parameters used in a HIP exchange. The subsections of this section document the required updates of HIP parameters. Other than the HIP DNS RR (Resource Record), these should not be needed in a DRIP implementation that does not use HIP.

See Section 3.2 for use of the HIT Suite in the context of DRIP.

3.4.1. HOST_ID

The HOST_ID parameter specifies the public key algorithm, and for elliptic curves, a name. The HOST_ID parameter is defined in Section 5.2.9 of [RFC7401].

Algorithm profiles	Values
EdDSA	TBD1 (suggested value 13) [RFC8032] (RECOMMENDED)

3.4.1.1. HIP Parameter support for EdDSA

The addition of EdDSA as a HI algorithm requires a subfield in the HIP HOST_ID parameter (Section 5.2.9 of [RFC7401]) as was done for ECDSA when used in a HIP exchange.

For HIP hosts that implement EdDSA as the algorithm, the following EdDSA curves are represented by the following fields:

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
EdDSA Curve		NULL	/
/	Public Key		

EdDSA Curve	Curve label
Public Key	Represented in Octet-string format [RFC8032]

Figure 2

For hosts that implement EdDSA as a HIP algorithm the following EdDSA curves are required:

Algorithm	Curve	Values
EdDSA	RESERVED	0
EdDSA	EdDSA25519	1 [RFC8032] (RECOMMENDED)
EdDSA	EdDSA25519ph	2 [RFC8032]
EdDSA	EdDSA448	3 [RFC8032] (RECOMMENDED)
EdDSA	EdDSA448ph	4 [RFC8032]

3.4.1.2. HIP DNS RR support for EdDSA

The HIP DNS RR is defined in [RFC8005]. It uses the values defined for the 'Algorithm Type' of the IPSECKEY RR [RFC4025] for its PK Algorithm field.

The new EdDSA HI uses [RFC8080] for the IPSECKEY RR encoding:

Value	Description
-------	-------------

TBD2 (suggested value 4)	
--------------------------	--

An EdDSA key is present, in the format defined in [RFC8080]	
---	--

3.4.2. HIT_SUITE_LIST

The HIT_SUITE_LIST parameter contains a list of the supported HIT suite IDs of the HIP Responder. Based on the HIT_SUITE_LIST, the HIP Initiator can determine which source HIT Suite IDs are supported by the Responder. The HIT_SUITE_LIST parameter is defined in Section 5.2.10 of [RFC7401].

The following HIT Suite ID is defined:

HIT Suite	Value	
EdDSA/cSHAKE128	TBD3 (suggested value 5)	(RECOMMENDED)

Table 1 provides more detail on the above HIT Suite combination.

The output of cSHAKE128 is variable per the needs of a specific ORCHID construction. It is at most 96 bits long and is directly used in the ORCHID (without truncation).

Index	Hash function	HMAC	Signature algorithm family	Description
5	cSHAKE128	KMAC128	EdDSA	EdDSA HI hashed with cSHAKE128, output is variable

Table 1: HIT Suites

3.5. ORCHIDs for Hierarchical HITs

This section improves on ORCHIDv2 [RFC7343] with three enhancements:

- * Optional "Info" field between the Prefix and OGA ID.
- * Increased flexibility on the length of each component in the ORCHID construction, provided the resulting ORCHID is 128 bits.
- * Use of cSHAKE, NIST SP 800-185 [NIST.SP.800-185], for the hashing function.

The Keccak [Keccak] based cSHAKE XOF hash function is a variable output length hash function. As such it does not use the truncation operation that other hashes need. The invocation of cSHAKE specifies the desired number of bits in the hash output. Further, cSHAKE has a parameter 'S' as a customization bit string. This parameter will be used for including the ORCHID Context Identifier in a standard fashion.

This ORCHID construction includes the fields in the ORCHID in the hash to protect them against substitution attacks. It also provides for inclusion of additional information, in particular the hierarchical bits of the Hierarchical HIT, in the ORCHID generation. This should be viewed as an update to ORCHIDv2 [RFC7343], as it can produce ORCHIDv2 output.

3.5.1. Adding Additional Information to the ORCHID

ORCHIDv2 [RFC7343] is defined as consisting of three components:

ORCHID := Prefix | OGA ID | Encode_96(Hash)

where:

Prefix	: A constant 28-bit-long bitstring value (IPv6 prefix)
OGA ID	: A 4-bit long identifier for the Hash_function in use within the specific usage context. When used for HIT generation this is the HIT Suite ID.
Encode_96()	: An extraction function in which output is obtained by extracting the middle 96-bit-long bitstring from the argument bitstring.

The new ORCHID function is as follows:

ORCHID := Prefix (p) | Info (n) | OGA ID (o) | Hash (m)

where:

Prefix (p) : An IPv6 prefix of length p (max 28-bit-long).

Info (n) : n bits of information that define a use of the ORCHID. 'n' can be zero, that is no additional information.

OGA ID (o) : A 4- or 8-bit long identifier for the Hash_function in use within the specific usage context. When used for HIT generation this is the HIT Suite ID. When used for HHIT generation this is the HHIT Suite ID.

Hash (m) : An extraction function in which output is 'm' bits.

$p + n + o + m = 128$ bits

The ORCHID length MUST be 128 bits. With a 28-bit IPv6 prefix, the remaining 100 bits can be divided in any manner between the additional information ("Info"), OGA ID, and the hash output. Care must be considering the size of the hash portion, taking into account risks like pre-image attacks. 64 bits, as used in Hierarchical HITs may be as small as is acceptable. The size of 'n' is determined as what is left; in the case of the 8-bit OGA used for HHIT, this is 28 bits.

3.5.2. ORCHID Encoding

This update adds a different encoding process to that currently used in ORCHIDv2. The input to the hash function explicitly includes all the header content plus the Context ID. The header content consists of the Prefix, the Additional Information ("Info"), and OGA ID (HIT Suite ID). Secondly, the length of the resulting hash is set by sum of the length of the ORCHID header fields. For example, a 28-bit prefix with 28 bits for the HID and 8 bits for the OGA ID leaves 64 bits for the hash length.

To achieve the variable length output in a consistent manner, the cSHAKE hash is used. For this purpose, cSHAKE128 is appropriate. The cSHAKE function call for this update is:

cSHAKE128(Input, L, "", Context ID)

Input := Prefix | Additional Information | OGA ID | HOST_ID
L := Length in bits of hash portion of ORCHID

For full Suite ID support (those that use fixed length hashes like SHA256), the following hashing can be used (Note: this does not produce output Identical to ORCHIDv2 for a /28 prefix and Additional Information of zero-length):

```
Hash[L] (Context ID | Input)

Input      := Prefix | Additional Information | OGA ID | HOST_ID
L          := Length in bits of hash portion of ORCHID

Hash[L]    := An extraction function in which output is obtained
               by extracting the middle L-bit-long bitstring
               from the argument bitstring.
```

Hierarchical HITs use the Context ID defined in Section 3.

3.5.2.1. Encoding ORCHIDs for HIPv2

This section discusses how to provide backwards compatibility for ORCHIDv2 [RFC7343] as used in HIPv2 [RFC7401].

For HIPv2, the Prefix is 2001:20::/28 (Section 6 of [RFC7343]). 'Info' is zero-length (i.e., not included), and OGA ID is 4-bit. Thus, the HI Hash is 96-bit length. Further, the Prefix and OGA ID are not included in the hash calculation. Thus, the following ORCHID calculations for fixed output length hashes are used:

```
Hash[L] (Context ID | Input)

Input      := HOST_ID
L          := 96
Context ID := 0xF0EF F02F BFF4 3D0F E793 0C3C 6E61 74EA

Hash[L]    := An extraction function in which output is obtained
               by extracting the middle L-bit-long bitstring
               from the argument bitstring.
```

For variable output length hashes use:

```
Hash[L] (Context ID | Input)

Input      := HOST_ID
L          := 96
Context ID := 0xF0EF F02F BFF4 3D0F E793 0C3C 6E61 74EA

Hash[L]    := The L-bit output from the hash function
```

Then, the ORCHID is constructed as follows:

Prefix | OGA ID | Hash Output

3.5.3. ORCHID Decoding

With this update, the decoding of an ORCHID is determined by the Prefix and OGA ID. ORCHIDv2 [RFC7343] decoding is selected when the Prefix is: 2001:20::/28.

For Hierarchical HITs, the decoding is determined by the presence of the HHIT Prefix as specified in Section 8.2.

3.5.4. Decoding ORCHIDs for HIPv2

This section is included to provide backwards compatibility for ORCHIDv2 [RFC7343] as used for HIPv2 [RFC7401].

HITs are identified by a Prefix of 2001:20::/28. The next 4 bits are the OGA ID. The remaining 96 bits are the HI Hash.

4. Hierarchical HITs as DRIP Entity Tags

HHITs for UAS ID (called, DETs) use the new EdDSA/SHAKE128 HIT suite defined in Section 3.4 (GEN-2 in [RFC9153]). This hierarchy, cryptographically bound within the HHIT, provides the information for finding the UA's HHIT registry (ID-3 in [RFC9153]).

The 2022 forthcoming updated release of ASTM Standard Specification for Remote ID and Tracking [F3411] adds support for DETs. This is within the UAS ID type 4, "Specific Session ID (SSI)".

Note to RFC Editor: This, and all references to F3411 need to be updated to this new version which is in final ASTM editing. A new link and replacement text will be provided when it is published.

The original UAS ID Types 1 - 3 allow for an UAS ID with a maximum length of 20 bytes, this new SSI (Type 4) uses the first byte of the ID for the SSI Type, thus restricting the UAS ID of this type to a maximum of 19 bytes. The SSI Types initially assigned are:

ID 1 IETF - DRIP Drone Remote ID Protocol (DRIP) entity ID.

ID 2 3GPP - IEEE 1609.2-2016 HashedID8

4.1. Nontransferability of DETs

A HI and its HHIT SHOULD NOT be transferable between UA or even between replacement electronics (e.g., replacement of damaged controller CPU) for a UA. The private key for the HI SHOULD be held in a cryptographically secure component.

4.2. Encoding HHITs in CTA 2063-A Serial Numbers

In some cases, it is advantageous to encode HHITs as a CTA 2063-A Serial Number [CTA2063A]. For example, the FAA Remote ID Rules [FAA_RID] state that a Remote ID Module (i.e., not integrated with UA controller) must only use "the serial number of the unmanned aircraft"; CTA 2063-A meets this requirement.

Encoding an HHIT within the CTA 2063-A format is not simple. The CTA 2063-A format is defined as follows:

Serial Number := MFR Code | Length Code | MFR SN

where:

MFR Code : 4 character code assigned by ICAO
(International Civil Aviation Organization,
a UN Agency).

Length Code : 1 character Hex encoding of MFR SN length (1-F).

MFR SN : Alphanumeric code (0-9, A-Z except O and I).
Maximum length of 15 characters.

There is no place for the HID; there will need to be a mapping service from Manufacturer Code to HID. The HHIT Suite ID and ORCHID hash will take the full 15 characters (as described below) of the MFR SN field.

A character in a CTA 2063-A Serial Number "shall include any combination of digits and uppercase letters, except the letters O and I, but may include all digits". This would allow for a Base34 encoding of the binary HHIT Suite ID and ORCHID hash in 15 characters. Although, programmatically, such a conversion is not hard, other technologies (e.g., credit card payment systems) that have used such odd base encoding have had performance challenges. Thus, here a Base32 encoding will be used by also excluding the letters Z and S (too similar to the digits 2 and 5).

The low-order 72 bits (HHIT Suite ID | ORCHID hash) of the HHIT SHALL be left-padded with 3 bits of zeros. This 75-bit number will be encoded into the 15-character MFR SN field using the digit/letters above. The manufacturer MUST use a Length Code of F (15).

Using the sample DET from Section 5 that is for HDA=20 under RAA=10 and having the ICAO CTA MFR Code of 8653, the 20-character CTA 2063-A Serial Number would be:

8653F02T7B8RA85D19LX

A mapping service (e.g., DNS) MUST provide a trusted (e.g., via DNSSEC [RFC4034]) conversion of the 4-character Manufacturer Code to high-order 58 bits (Prefix | HID) of the HHIT. Definition of this mapping service is currently out of scope of this document.

It should be noted that this encoding would only be used in the Basic ID Message (Section 2.2 of [RFC9153]). The DET is used in the Authentication Messages (i.e., the messages that provide framing for authentication data only).

4.3. Remote ID DET as one Class of Hierarchical HITs

UAS Remote ID DET may be one of a number of uses of HHITs. However, it is out of the scope of the document to elaborate on other uses of HHITs. As such these follow-on uses need to be considered in allocating the RAAs (Section 3.3.1) or HHIT prefix assignments (Section 8).

4.4. Hierarchy in ORCHID Generation

ORCHIDS, as defined in [RFC7343], do not cryptographically bind an IPv6 prefix nor the ORCHID Generation Algorithm (OGA) ID (the HIT Suite ID) to the hash of the HI. The rationale at the time of developing ORCHID was attacks against these fields are Denial-of-Service (DoS) attacks against protocols using ORCHIDs and thus up to those protocols to address the issue.

HHITs, as defined in Section 3.5, cryptographically bind all content in the ORCHID through the hashing function. A recipient of a DET that has the underlying HI can directly trust and act on all content in the HHIT. This provides a strong, self-attestation for using the hierarchy to find the DET Registry based on the HID (Section 4.5).

4.5. DRIP Entity Tag (DET) Registry

DETs are registered to HDAs. A registration process, [drip-registries], ensures DET global uniqueness (ID-4 in [RFC9153]). It also provides the mechanism to create UAS public/private data that are associated with the DET (REG-1 and REG-2 in [RFC9153]).

4.6. Remote ID Authentication using DETs

The EdDSA25519 HI (Section 3.4) underlying the DET can be used in an 84-byte self-proof attestation (timestamp, HHIT, and signature of these) to provide proof of Remote ID ownership (GEN-1 in [RFC9153]). In practice, the Wrapper and Manifest authentication formats (Sections 6.3.3 and 6.3.4 of [drip-authentication]) implicitly provide this self-attestation. A lookup service like DNS can provide the HI and registration proof (GEN-3 in [RFC9153]).

Similarly, for Observers without Internet access, a 200-byte offline self-attestation could provide the same Remote ID ownership proof. This attestation would contain the HDA's signing of the UA's HHIT, itself signed by the UA's HI. Only a small cache that contains the HDA's HI/HHIT and HDA meta-data is needed by the Observer. However, such an object would just fit in the ASTM Authentication Message (Section 2.2 of [RFC9153]) with no room for growth. In practice, [drip-authentication] provides this offline self-attestation in two authentication messages: the HDA's certification of the UA's HHIT registration in a Link authentication message whose hash is sent in a Manifest authentication message.

Hashes of any previously sent ASTM messages can be placed in a Manifest authentication message (GEN-2 in [RFC9153]). When a Location/Vector Message (i.e., a message that provides UA location, altitude, heading, speed, and status) hash along with the hash of the HDA's UA HHIT attestation are sent in a Manifest authentication message and the Observer can visually see a UA at the claimed location, the Observer has a very strong proof of the UA's Remote ID.

All this behavior and how to mix these authentication messages into the flow of UA operation messages are detailed in [drip-authentication].

5. DRIP Entity Tags (DETs) in DNS

There are two approaches for storing and retrieving DETs using DNS. The following are examples of how this may be done. This will serve as guidance to the actual deployment of DETs in DNS. However, this document does not intend to provide a recommendation. Further DNS-related considerations are covered in [drip-registries].

* As FQDNs, for example, ".icao.int".

* Reverse DNS lookups as IPv6 addresses per [RFC8005].

A DET can be used to construct an FQDN that points to the USS that has the public/private information for the UA (REG-1 and REG-2 in [RFC9153]). For example, the USS for the HHIT could be found via the following: assume the RAA is decimal 100 and the HDA is decimal 50. The PTR record is constructed as follows:

```
100.50.det.uas.icao.int    IN PTR        foo.uss.icao.int.
```

The individual DETs may be potentially too numerous (e.g., 60 - 600M) and dynamic (e.g., new DETs every minute for some HDAs) to store in a signed, DNS zone. The HDA SHOULD provide DNS service for its zone and provide the HHIT detail response.

The DET reverse lookup can be a standard IPv6 reverse look up, or it can leverage off the HHIT structure. Using the allocated prefix for HHITs TBD6 [suggested value 2001:30::/28] (See Section 3.1), the RAA is 10 and the HDA is 20, the DET is:

```
2001:30:280:1405:a3ad:1952:ad0:a69e
```

A DET reverse lookup could be to:

```
a69e.ad0.1952.a3ad.1405.280.30.2001.20.10.det.arpa.
```

or:

```
a3ad1952ad0a69e.5.20.10.30.2001.det.remoteid.icao.int.
```

A 'standard' ip6.arpa RR has the advantage of only one Registry service supported.

```
$ORIGIN 5.0.4.1.0.8.2.0.0.3.0.0.1.0.0.2.ip6.arpa.
e.9.6.a.0.d.a.0.2.5.9.1.d.a.3.a    IN    PTR
a3ad1952ad0a69e.20.10.det.rid.icao.int.
```

This DNS entry for the DET can also provide a revocation service. For example, instead of returning the HI RR it may return some record showing that the HI (and thus DET) has been revoked. Guidance on revocation service will be provided in [drip-registries].

6. Other UTM Uses of HHITs Beyond DET

HHITs will be used within the UTM architecture beyond DET (and USS in UA ID registration and authentication), for example, as a Ground Control Station (GCS) HHIT ID. Some GCS will use its HHIT for securing its Network Remote ID (to USS HHIT) and Command and Control (C2, Section 2.2.2 of [RFC9153]) transports.

Observers may have their own HHITs to facilitate UAS information retrieval (e.g., for authorization to private UAS data). They could also use their HHIT for establishing a HIP connection with the UA Pilot for direct communications per authorization. Details about such issues are out of the scope of this document).

7. Summary of Addressed DRIP Requirements

This document provides the details to solutions for GEN 1 - 3, ID 1 - 5, and REG 1 - 2 requirements that are described in [RFC9153].

8. IANA Considerations

8.1. New Well-Known IPv6 prefix for DETs

Since the DET format is not compatible with [RFC7343], IANA is requested to allocate a new prefix following this template for the IPv6 Special-Purpose Address Registry.

Address Block:

IANA is requested to allocate a new 28-bit prefix out of the IANA IPv6 Special Purpose Address Block, namely 2001::/23, as per [RFC6890] (TBD6, suggested: 2001:30::/28).

Name:

This block should be named "DRIP Entity Tags (DETs) Prefix".

RFC:

This document.

Allocation Date:

Date this document published.

Termination Date:

Forever.

Source:

False.

Destination:
False.

Forwardable:
False.

Globally Reachable:
False.

Reserved-by-Protocol:
False.

8.2. New IANA DRIP Registry

This document requests IANA to create a new registry titled "Drone Remote ID Protocol" registry. The following two subregistries should be created under that registry.

Hierarchical HIT (HHIT) Prefixes:

Initially, for DET use, one 28-bit prefix should be assigned out of the IANA IPv6 Special Purpose Address Block, namely 2001::/23, as per [RFC6890]. Future additions to this subregistry are to be made through Expert Review (Section 4.5 of [RFC8126]). Entries with network-specific prefixes may be present in the registry.

HHIT Use	Bits	Value
DET	28	TBD6 (suggested value 2001:30::/28)

Hierarchical HIT (HHIT) Suite ID:

This 8-bit valued subregistry is a superset of the 4/8-bit "HIT Suite ID" subregistry of the "Host Identity Protocol (HIP) Parameters" registry in [IANA-HIP]. Future additions to this subregistry are to be made through IETF Review (Section 4.8 of [RFC8126]). The following HHIT Suite IDs are defined:

HHIT Suite	Value
RESERVED	0
RSA,DSA/SHA-256	1 [RFC7401]
ECDSA/SHA-384	2 [RFC7401]
ECDSA_LOW/SHA-1	3 [RFC7401]
EdDSA/cSHAKE128	TBD3 (suggested value 5) (RECOMMENDED)
HDA Private Use 1	TBD4 (suggested value 254)
HDA Private Use 2	TBD5 (suggested value 255)

The HHIT Suite ID values 1 - 31 are reserved for IDs that MUST be replicated as HIT Suite IDs (Section 8.4) as is TBD3 here. Higher values (32 - 255) are for those Suite IDs that need not or cannot be accommodated as a HIT Suite ID.

8.3. IANA CGA Registry Update

This document requests that this document be added to the reference field for the "CGA Extension Type Tags" registry [IANA-CGA], where IANA registers the following Context ID:

Context ID:

The Context ID (Section 3) shares the namespace introduced for CGA Type Tags. Defining new Context IDs follow the rules in Section 8 of [RFC3972]:

Context ID := 0x00B5 A69C 795D F5D5 F008 7F56 843F 2C40

8.4. IANA HIP Registry Updates

This document requests IANA to make the following changes to the IANA "Host Identity Protocol (HIP) Parameters" [IANA-HIP] registry:

Host ID:

This document defines the new EdDSA Host ID with value TBD1 (suggested: 13) (Section 3.4.1) in the "HI Algorithm" subregistry of the "Host Identity Protocol (HIP) Parameters" registry.

Algorithm profiles	Values
EdDSA	TBD1 (suggested value 13) [RFC8032] (RECOMMENDED)

EdDSA Curve Label:

This document specifies a new algorithm-specific subregistry named "EdDSA Curve Label". The values for this subregistry are defined in Section 3.4.1.1. Future additions to this subregistry are to be made through IETF Review (Section 4.8 of [RFC8126]).

Algorithm	Curve	Values	
EdDSA	RESERVED	0	
EdDSA	EdDSA25519	1 [RFC8032]	(RECOMMENDED)
EdDSA	EdDSA25519ph	2 [RFC8032]	
EdDSA	EdDSA448	3 [RFC8032]	(RECOMMENDED)
EdDSA	EdDSA448ph	4 [RFC8032]	
		5-65535	Unassigned

HIT Suite ID:

This document defines the new HIT Suite of EdDSA/cSHAKE with value TBD3 (suggested: 5) (Section 3.4.2) in the "HIT Suite ID" subregistry of the "Host Identity Protocol (HIP) Parameters" registry.

HIT Suite	Value	
EdDSA/cSHAKE128	TBD3 (suggested value 5)	(RECOMMENDED)

The HIT Suite ID 4-bit values 1 - 15 and 8-bit values 0x00 - 0x0F MUST be replicated as HHIT Suite IDs (Section 8.2) as is TBD3 here.

8.5. IANA IPSECKEY Registry Update

This document requests IANA to make the following change to the "IPSECKEY Resource Record Parameters" [IANA-IPSECKEY] registry:

IPSECKEY:

This document defines the new IPSECKEY value TBD2 (suggested: 4) (Section 3.4.1.2) in the "Algorithm Type Field" subregistry of the "IPSECKEY Resource Record Parameters" registry.

Value Description

TBD2 (suggested value 4)
An EdDSA key is present, in the format defined in [RFC8080]

9. Security Considerations

The 64-bit hash in HHITs presents a real risk of second pre-image cryptographic hash attack Section 9.4. There are no known (to the authors) studies of hash size to cryptographic hash attacks. A Python script is available to randomly generate 1M HHITs that did not produce a hash collision which is a simpler attack than a first or second pre-image attack.

However, with today's computing power, producing 2^{64} EdDSA keypairs and then generating the corresponding HHIT is economically feasible. Consider that a *single* bitcoin mining ASIC can do on the order of 2^{46} sha256 hashes a second or about 2^{62} hashes in a single day. The point being, 2^{64} is not prohibitive, especially as this can be done in parallel.

Now it should be noted that the 2^{64} attempts is for stealing a specific HHIT. Consider a scenario of a street photography company with 1,024 UAs (each with its own HHIT); you'd be happy stealing any one of them. Then rather than needing to satisfy a 64-bit condition on the cSHAKE128 output, you need only satisfy what is equivalent to a 54-bit condition (since there are 2^{10} more opportunities for success).

Thus, although the probability of a collision or pre-image attack is low in a collection of 1,024 HHITs out of a total population of 2^{64} , per Section 9.4, it is computationally and economically feasible. Therefore, the HHIT registration and HHIT/HI registration validation is strongly recommended.

The DET Registry services effectively block attempts to "take over" or "hijack" a DET. It does not stop a rogue attempting to impersonate a known DET. This attack can be mitigated by the receiver of messages containing DETs using DNS to find the HI for the DET. As such, use of DNSSEC by the DET registries is recommended to provide trust in HI retrieval.

The 60-bit hash for DETs with 8-bit OGAs have a greater hash attack risk. As such its use should be restricted to testing and to small, well managed UAS/USS.

Another mitigation of HHIT hijacking is if the HI owner (UA) supplies an object containing the HHIT and signed by the HI private key of the HDA such as detailed in [drip-authentication].

The two risks with hierarchical HITs are the use of an invalid HID and forced HIT collisions. The use of a DNS zone (e.g., "det.arpa.") is a strong protection against invalid HIDs. Querying an HDA's RVS for a HIT under the HDA protects against talking to unregistered clients. The Registry service [drip-registries], through its HHIT uniqueness enforcement, provides against forced or accidental HHIT hash collisions.

Cryptographically Generated Addresses (CGAs) provide an assurance of uniqueness. This is two-fold. The address (in this case the UAS ID) is a hash of a public key and a Registry hierarchy naming. Collision resistance (more important than it implied second-preimage

resistance) makes it statistically challenging to attacks. A registration process [drip-registries] within the HDA provides a level of assured uniqueness unattainable without mirroring this approach.

The second aspect of assured uniqueness is the digital signing (attestation) process of the DET by the HI private key and the further signing (attestation) of the HI public key by the Registry's key. This completes the ownership process. The observer at this point does not know what owns the DET, but is assured, other than the risk of theft of the HI private key, that this UAS ID is owned by something and is properly registered.

9.1. DET Trust in ASTM messaging

The DET in the ASTM Basic ID Message (Msg Type 0x0, the actual Remote ID message) does not provide any assertion of trust. The best that might be done within this Basic ID Message is 4 bytes truncated from a HI signing of the HHIT (the UA ID field is 20 bytes and a HHIT is 16). This is not trustable; that is, too open to a hash attack. Minimally, it takes 84 bytes (Section 4.6) to prove ownership of a DET with a full EdDSA signature. Thus, no attempt has been made to add DET trust directly within the very small Basic ID Message.

The ASTM Authentication Message (Msg Type 0x2) as shown in Section 4.6 can provide practical actual ownership proofs. These attestations include timestamps to defend against replay attacks. But in themselves, they do not prove which UA sent the message. They could have been sent by a dog running down the street with a Broadcast Remote ID module strapped to its back.

Proof of UA transmission comes when the Authentication Message includes proofs for the ASTM Location/Vector Message (Msg Type 0x1) and the observer can see the UA or that information is validated by ground multilateration. Only then does an observer gain full trust in the DET of the UA.

DETs obtained via the Network RID path provides a different approach to trust. Here the UAS SHOULD be securely communicating to the USS, thus asserting DET trust.

9.2. DET Revocation

The DNS entry for the DET can also provide a revocation service. For example, instead of returning the HI RR it may return some record showing that the HI (and thus DET) has been revoked. Guidance on revocation service will be provided in [drip-registries].

9.3. Privacy Considerations

There is no expectation of privacy for DETs; it is not part of the privacy normative requirements listed in, Section 4.3.1, of [RFC9153]. DETs are broadcast in the clear over the open air via Bluetooth and Wi-Fi. They will be collected and collated with other public information about the UAS. This will include DET registration information and location and times of operations for a DET. A DET can be for the life of a UA if there is no concern about DET/UA activity harvesting.

Further, the MAC address of the wireless interface used for Remote ID broadcasts are a target for UA operation aggregation that may not be mitigated through MAC address randomization. For Bluetooth 4 Remote ID messaging, the MAC address is used by observers to link the Basic ID Message that contains the RID with other Remote ID messages, thus must be constant for a UA operation. This message linkage use of MAC addresses may not be needed with the Bluetooth 5 or Wi-Fi PHYs. These PHYs provide for a larger message payload and can use the Message Pack (Msg Type 0xF) and the Authentication Message to transmit the RID with other Remote ID messages. However, it is not mandatory to send the RID in a Message Pack or Authentication Message, so allowance for using the MAC address for UA message linking must be maintained. That is, the MAC address should be stable for at least a UA operation.

Finally, it is not adequate to simply change the DET and MAC for a UA per operation to defeat historically tracking a UA's activity.

Any changes to the UA MAC may have impacts to C2 setup and use. A constant GCS MAC may well defeat any privacy gains in UA MAC and RID changes. UA/GCS binding is complicated with changing MAC addresses; historically UAS design assumed these to be "forever" and made setup a one-time process. Additionally, if IP is used for C2, a changing MAC may mean a changing IP address to further impact the UAS bindings. Finally, an encryption wrapper's identifier (such as ESP [RFC4303] SPI) would need to change per operation to insure operation tracking separation.

Creating and maintaining UAS operational privacy is a multifaceted problem. Many communication pieces need to be considered to truly create a separation between UA operations. Simply changing the DET only starts the changes that need to be implemented.

These privacy realities may present challenges for the EU U-space (Appendix A) program.

9.4. Collision Risks with DETs

The 64-bit hash size does have an increased risk of collisions over the 96-bit hash size used for the other HIT Suites. There is a 0.01% probability of a collision in a population of 66 million. The probability goes up to 1% for a population of 663 million. See Appendix C for the collision probability formula.

However, this risk of collision is within a single "Additional Information" value, i.e., a RAA/HDA domain. The UAS/USS registration process should include registering the DET and MUST reject a collision, forcing the UAS to generate a new HI and thus HHIT and reapplying to the DET registration process.

10. References

10.1. Normative References

[NIST.FIPS.202]

Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.202, July 2015, <<https://doi.org/10.6028/nist.fips.202>>.

[NIST.SP.800-185]

Kelsey, J., Change, S., and R. Perlner, "SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-185, December 2016, <<https://doi.org/10.6028/nist.sp.800-185>>.

[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/info/rfc2119>>.

[RFC6890] Cotton, M., Vegoda, L., Bonica, R., Ed., and B. Haberman, "Special-Purpose IP Address Registries", BCP 153, RFC 6890, DOI 10.17487/RFC6890, April 2013, <<https://www.rfc-editor.org/info/rfc6890>>.

[RFC7343] Laganier, J. and F. Dupont, "An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers Version 2 (ORCHIDv2)", RFC 7343, DOI 10.17487/RFC7343, September 2014, <<https://www.rfc-editor.org/info/rfc7343>>.

- [RFC7401] Moskowitz, R., Ed., Heer, T., Jokela, P., and T. Henderson, "Host Identity Protocol Version 2 (HIPv2)", RFC 7401, DOI 10.17487/RFC7401, April 2015, <<https://www.rfc-editor.org/info/rfc7401>>.
- [RFC8005] Laganier, J., "Host Identity Protocol (HIP) Domain Name System (DNS) Extension", RFC 8005, DOI 10.17487/RFC8005, October 2016, <<https://www.rfc-editor.org/info/rfc8005>>.
- [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/info/rfc8032>>.
- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [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/info/rfc8174>>.

10.2. Informative References

- [cfrg-comment] "A CFRG review of draft-ietf-drip-rid", September 2021, <https://mailarchive.ietf.org/arch/msg/cfrg/tAJJq60W6TlUv7_pde5cw5TDTCU/>.
- [corus] CORUS, "U-space Concept of Operations", September 2019, <<https://www.sesarju.eu/node/3411>>.
- [CTA2063A] ANSI/CTA, "Small Unmanned Aerial Systems Serial Numbers", September 2019, <<https://shop.cta.tech/products/small-unmanned-aerial-systems-serial-numbers>>.
- [drip-architecture] Card, S. W., Wiethuechter, A., Moskowitz, R., Zhao, S., and A. Gurtov, "Drone Remote Identification Protocol (DRIP) Architecture", Work in Progress, Internet-Draft, draft-ietf-drip-arch-22, 21 March 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-drip-arch-22>>.
- [drip-authentication] Wiethuechter, A., Card, S., and R. Moskowitz, "DRIP Entity Tag Authentication Formats & Protocols for Broadcast

Remote ID", Work in Progress, Internet-Draft, draft-ietf-drip-auth-10, 11 May 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-drip-auth-10>>.

[drip-registries]

Wiethuechter, A., Card, S., Moskowitz, R., and J. Reid, "DRIP Entity Tag Registration & Lookup", Work in Progress, Internet-Draft, draft-ietf-drip-registries-02, 30 April 2022, <<https://datatracker.ietf.org/doc/html/draft-ietf-drip-registries-02>>.

[F3411] ASTM International, "Standard Specification for Remote ID and Tracking", <<http://www.astm.org/cgi-bin/resolver.cgi?F3411>>.

[FAA_RID] United States Federal Aviation Administration (FAA), "Remote Identification of Unmanned Aircraft", 2021, <<https://www.govinfo.gov/content/pkg/FR-2021-01-15/pdf/2020-28948.pdf>>.

[IANA-CGA] IANA, "Cryptographically Generated Addresses (CGA) Message Type Name Space", <<https://www.iana.org/assignments/cga-message-types/cga-message-types.xhtml>>.

[IANA-HIP] IANA, "Host Identity Protocol (HIP) Parameters", <<https://www.iana.org/assignments/hip-parameters/hip-parameters.xhtml>>.

[IANA-IPSECKEY]

IANA, "IPSECKEY Resource Record Parameters", <<https://www.iana.org/assignments/ipseckey-rr-parameters/ipseckey-rr-parameters.xhtml>>.

[Keccak] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "The Keccak Function", <<https://keccak.team/index.html>>.

[RFC3972] Aura, T., "Cryptographically Generated Addresses (CGA)", RFC 3972, DOI 10.17487/RFC3972, March 2005, <<https://www.rfc-editor.org/info/rfc3972>>.

[RFC4025] Richardson, M., "A Method for Storing IPsec Keying Material in DNS", RFC 4025, DOI 10.17487/RFC4025, March 2005, <<https://www.rfc-editor.org/info/rfc4025>>.

- [RFC4034] Arends, R., Austein, R., Larson, M., Massey, D., and S. Rose, "Resource Records for the DNS Security Extensions", RFC 4034, DOI 10.17487/RFC4034, March 2005, <<https://www.rfc-editor.org/info/rfc4034>>.
- [RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <<https://www.rfc-editor.org/info/rfc4122>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [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/info/rfc5280>>.
- [RFC8004] Laganier, J. and L. Eggert, "Host Identity Protocol (HIP) Rendezvous Extension", RFC 8004, DOI 10.17487/RFC8004, October 2016, <<https://www.rfc-editor.org/info/rfc8004>>.
- [RFC8080] Sury, O. and R. Edmonds, "Edwards-Curve Digital Security Algorithm (EdDSA) for DNSSEC", RFC 8080, DOI 10.17487/RFC8080, February 2017, <<https://www.rfc-editor.org/info/rfc8080>>.
- [RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <<https://www.rfc-editor.org/info/rfc8200>>.
- [RFC9063] Moskowitz, R., Ed. and M. Komu, "Host Identity Protocol Architecture", RFC 9063, DOI 10.17487/RFC9063, July 2021, <<https://www.rfc-editor.org/info/rfc9063>>.
- [RFC9153] Card, S., Ed., Wiethuechter, A., Moskowitz, R., and A. Gurtov, "Drone Remote Identification Protocol (DRIP) Requirements and Terminology", RFC 9153, DOI 10.17487/RFC9153, February 2022, <<https://www.rfc-editor.org/info/rfc9153>>.
- [RFC9224] Blanchet, M., "Finding the Authoritative Registration Data Access Protocol (RDAP) Service", STD 95, RFC 9224, DOI 10.17487/RFC9224, March 2022, <<https://www.rfc-editor.org/info/rfc9224>>.

Appendix A. EU U-Space RID Privacy Considerations

The EU is defining a future of airspace management known as U-space within the Single European Sky ATM Research (SESAR) undertaking. Concept of Operation for European UTM Systems (CORUS) project proposed low-level Concept of Operations [corus] for UAS in the EU. It introduces strong requirements for UAS privacy based on European GDPR regulations. It suggests that UAs are identified with agnostic IDs, with no information about UA type, the operators or flight trajectory. Only authorized persons should be able to query the details of the flight with a record of access.

Due to the high privacy requirements, a casual observer can only query U-space if it is aware of a UA seen in a certain area. A general observer can use a public U-space portal to query UA details based on the UA transmitted "Remote identification" signal. Direct remote identification (DRID) is based on a signal transmitted by the UA directly. Network remote identification (NRID) is only possible for UAs being tracked by U-Space and is based on the matching the current UA position to one of the tracks.

This is potentially a contrary expectation as that presented in Section 9.3. U-space will have to deal with this reality within the GDPR regulations. Still, DETs as defined here present a large step in the right direction for agnostic IDs.

The project lists "E-Identification" and "E-Registrations" services as to be developed. These services can use DETs and follow the privacy considerations outlined in this document for DETs.

If an "agnostic ID" above refers to a completely random identifier, it creates a problem with identity resolution and detection of misuse. On the other hand, a classical HIT has a flat structure which makes its resolution difficult. The DET (Hierarchical HIT) provides a balanced solution by associating a registry with the UA identifier. This is not likely to cause a major conflict with U-space privacy requirements, as the registries are typically few at a country level (e.g., civil personal, military, law enforcement, or commercial).

Appendix B. The 14/14 HID split

The following explains the logic behind selecting to divide the 28 bits of the HID into 2 14-bit components.

At this writing ICAO has 273 member "States", each may want to control RID assignment within its National Air Space (NAS). Some members may want separate RAAs to use for Civil, general Government,

and Military use. They may also want allowances for competing Civil RAA operations. It is reasonable to plan for 8 RAAs per ICAO member (plus regional aviation organizations like in the European Union). Thus at a start a 4,096 RAA space is advised.

There will be requests by commercial entities for their own, RAA allotments. Examples could include international organizations that will be using UAS and international delivery service associations. These may be smaller than the RAA space needed by ICAO member States and could be met with a 2,048 space allotment, but as will be seen, might as well be 4,096 as well.

This may well cover currently understood RAA entities. There will be future new applications, branching off into new areas. So yet another space allocation should be set aside. If this is equal to all that has been reserved, we should allow for 16,384 (2^{14}) RAAs.

The HDA allocation follows a different logic from that of RAAs. Per Appendix C, an HDA should be able to easily assign 63M RIDs and even manage 663M with a "first come, first assigned" registration process. For most HDAs this is more than enough, and a single HDA assignment within their RAA will suffice. Most RAAs will only delegate to a couple HDAs for their operational needs. But there are major exceptions that point to some RAAs needing large numbers of HDA assignments.

Delivery service operators like Amazon (est. 30K delivery vans) and UPS (est. 500K delivery vans) may choose, for anti-tracking reasons, to use unique RIDs per day or even per operation. 30K delivery UA could need 11M upwards to 44M RIDs. Anti-tracking would be hard to provide if the HID were the same for a delivery service fleet, so such a company may turn to an HDA that provides this service to multiple companies so that who's UA is who's is not evident in the HID. A USS providing this service could well use multiple HDA assignments per year, depending on strategy.

Perhaps a single RAA providing HDAs for delivery service (or similar behaving) UAS could 'get by' with a 2048 HDA space (11-bits). So the HDA space could well be served with only 12 bits allocated out of the 28-bit HID space. But as this is speculation, and it will take years of deployment experience, a 14-bit HDA space has been selected.

There may also be 'small' ICAO member States that opt for a single RAA and allocate their HDAs for all UA that are permitted in their NAS. The HDA space is large enough that some to use part for government needs as stated above and for small commercial needs. Or the State may use a separate, consecutive RAA for commercial users. Thus it would be 'easy' to recognize State-approved UA by HID high-order bits.

Appendix C. Calculating Collision Probabilities

The accepted formula for calculating the probability of a collision is:

$$p = 1 - e^{\{-k^2/(2n)\}}$$

P Collision Probability
 n Total possible population
 k Actual population

The following table provides the approximate population size for a collision for a given total population.

Total Population	Deployed Population With Collision Risk of	
	.01%	1%
2 ⁹⁶	4T	42T
2 ⁷²	1B	10B
2 ⁶⁸	250M	2.5B
2 ⁶⁴	66M	663M
2 ⁶⁰	16M	160M

Acknowledgments

Dr. Gurtoov is an adviser on Cybersecurity to the Swedish Civil Aviation Administration.

Quynh Dang of NIST gave considerable guidance on using Keccak and the NIST supporting documents. Joan Deamen of the Keccak team was especially helpful in many aspects of using Keccak. Nicholas Gajcowski [cfrg-comment] provided a concise hash pre-image security assessment via the CFRG list.

Many thanks to Michael Richardson and Brian Haberman for the iotdir review, Magnus Nystrom for the secdir review, Elwyn Davies for genart review and DRIP co-chair and draft shepherd, Mohamed Boucadair for his extensive comments and help on document clarity.

Authors' Addresses

Robert Moskowitz
HTT Consulting
Oak Park, MI 48237
United States of America
Email: rgm@labs.htt-consult.com

Stuart W. Card
AX Enterprize, LLC
4947 Commercial Drive
Yorkville, NY 13495
United States of America
Email: stu.card@axenterprize.com

Adam Wiethuechter
AX Enterprize, LLC
4947 Commercial Drive
Yorkville, NY 13495
United States of America
Email: adam.wiethuechter@axenterprize.com

Andrei Gurtov
Linköping University
IDA
SE-58183 Linköping
Sweden
Email: gurtov@acm.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 18 July 2022

M. Abdalla
DFINITY - Zurich
B. Haase
J. Hesse
Endress + Hauser Liquid Analysis - Gerlingen
IBM Research Europe - Zurich
14 January 2022

CPlace, a balanced composable PAKE
draft-irtf-cfrg-cplace-05

Abstract

This document describes CPlace which is a protocol for two parties that share a low-entropy secret (password) to derive a strong shared key without disclosing the secret to offline dictionary attacks. This method was tailored for constrained devices, is compatible with any group of both prime- and non-prime order, and comes with a security proof providing composability guarantees.

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/cfrg/draft-irtf-cfrg-cplace>.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 18 July 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	5
2. Requirements Notation	5
3. Definition CPace	6
3.1. Setup	6
3.1.1. Hash function H	6
3.1.2. Group environment G	7
3.2. Inputs	7
3.3. Notation	8
3.3.1. Notation for group operations	9
4. The CPace protocol	9
4.1. Session identifier establishment	10
4.2. Protocol flow	10
4.3. CPace protocol instructions	10
5. CPace cipher suites	11
6. Implementation of recommended CPace cipher suites	12
6.1. Common function for computing generators	12
6.2. CPace group objects G_X25519 and G_X448 for single-coordinate Ladders on Montgomery curves	13
6.2.1. Verification tests	14
6.3. CPace group objects G_Ristretto255 and G_Decaf448 for prime-order group abstractions	15
6.3.1. Verification tests	17
6.4. CPace group objects for curves in Short-Weierstrass representation	17
6.4.1. Curves and associated functions	17
6.4.2. Suitable encode_to_curve methods	18
6.4.3. Definition of the group environment G for Short-Weierstrass curves	18
6.4.4. Verification tests	20
7. Implementation verification	20
8. Security Considerations	20
8.1. Party identifiers and relay attacks	20

8.2.	Hashing and key derivation	21
8.3.	Key confirmation	21
8.4.	Sampling of scalars	22
8.5.	Single-coordinate CPace on Montgomery curves	22
8.6.	Nonce values	23
8.7.	Side channel attacks	23
8.8.	Quantum computers	23
9.	IANA Considerations	24
10.	Acknowledgements	24
11.	References	24
11.1.	Normative References	24
11.2.	Informative References	25
Appendix A.	CPace function definitions	26
A.1.	Definition and test vectors for string utility functions	26
A.1.1.	prepend_len function	26
A.1.2.	prepend_len test vectors	26
A.1.3.	prefix_free_cat function	27
A.1.4.	Testvector for prefix_free_cat()	27
A.1.5.	Examples for invalid encoded messages	27
A.2.	Definition of generator_string function.	28
A.3.	Definitions and test vector ordered concatenation	28
A.3.1.	Definitions for lexicographical ordering	28
A.3.2.	Definitions for ordered concatenation	28
A.3.3.	Test vectors ordered concatenation	28
A.4.	Decoding and Encoding functions according to RFC7748	29
A.5.	Elligator 2 reference implementation	29
Appendix B.	Test vectors	30
B.1.	Test vector for CPace using group X25519 and hash SHA-512	30
B.1.1.	Test vectors for calculate_generator with group X25519	30
B.1.2.	Test vector for MSGa	31
B.1.3.	Test vector for MSGb	31
B.1.4.	Test vector for secret points K	32
B.1.5.	Test vector for ISK calculation initiator/responder	32
B.1.6.	Test vector for ISK calculation parallel execution	32
B.1.7.	Corresponding ANSI-C initializers	33
B.1.8.	Test vectors for G_X25519.scalar_mult_vfy: low order points	34
B.2.	Test vector for CPace using group X448 and hash SHAKE-256	35
B.2.1.	Test vectors for calculate_generator with group X448	35
B.2.2.	Test vector for MSGa	36
B.2.3.	Test vector for MSGb	36
B.2.4.	Test vector for secret points K	37

B.2.5.	Test vector for ISK calculation initiator/ responder	37
B.2.6.	Test vector for ISK calculation parallel execution	38
B.2.7.	Corresponding ANSI-C initializers	38
B.2.8.	Test vectors for G_X448.scalar_mult_vfy: low order points	40
B.3.	Test vector for CPace using group ristretto255 and hash SHA-512	41
B.3.1.	Test vectors for calculate_generator with group ristretto255	41
B.3.2.	Test vector for MSGa	42
B.3.3.	Test vector for MSGb	42
B.3.4.	Test vector for secret points K	43
B.3.5.	Test vector for ISK calculation initiator/ responder	43
B.3.6.	Test vector for ISK calculation parallel execution	43
B.3.7.	Corresponding ANSI-C initializers	44
B.3.8.	Test case for scalar_mult with valid inputs	45
B.3.9.	Invalid inputs for scalar_mult_vfy	46
B.4.	Test vector for CPace using group decaf448 and hash SHAKE-256	46
B.4.1.	Test vectors for calculate_generator with group decaf448	46
B.4.2.	Test vector for MSGa	47
B.4.3.	Test vector for MSGb	47
B.4.4.	Test vector for secret points K	48
B.4.5.	Test vector for ISK calculation initiator/ responder	48
B.4.6.	Test vector for ISK calculation parallel execution	49
B.4.7.	Corresponding ANSI-C initializers	49
B.4.8.	Test case for scalar_mult with valid inputs	51
B.4.9.	Invalid inputs for scalar_mult_vfy	51
B.5.	Test vector for CPace using group NIST P-256 and hash SHA-256	51
B.5.1.	Test vectors for calculate_generator with group NIST P-256	51
B.5.2.	Test vector for MSGa	52
B.5.3.	Test vector for MSGb	52
B.5.4.	Test vector for secret points K	53
B.5.5.	Test vector for ISK calculation initiator/ responder	53
B.5.6.	Test vector for ISK calculation parallel execution	54
B.5.7.	Corresponding ANSI-C initializers	55
B.5.8.	Test case for scalar_mult_vfy with correct inputs	56
B.5.9.	Invalid inputs for scalar_mult_vfy	57
B.6.	Test vector for CPace using group NIST P-384 and hash SHA-384	57

B.6.1.	Test vectors for calculate_generator with group NIST P-384	57
B.6.2.	Test vector for MSGa	58
B.6.3.	Test vector for MSGb	59
B.6.4.	Test vector for secret points K	59
B.6.5.	Test vector for ISK calculation initiator/ responder	59
B.6.6.	Test vector for ISK calculation parallel execution .	60
B.6.7.	Corresponding ANSI-C initializers	61
B.6.8.	Test case for scalar_mult_vfy with correct inputs . .	63
B.6.9.	Invalid inputs for scalar_mult_vfy	63
B.7.	Test vector for CPace using group NIST P-521 and hash SHA-512	64
B.7.1.	Test vectors for calculate_generator with group NIST P-521	64
B.7.2.	Test vector for MSGa	64
B.7.3.	Test vector for MSGb	65
B.7.4.	Test vector for secret points K	66
B.7.5.	Test vector for ISK calculation initiator/ responder	66
B.7.6.	Test vector for ISK calculation parallel execution .	67
B.7.7.	Corresponding ANSI-C initializers	68
B.7.8.	Test case for scalar_mult_vfy with correct inputs . .	70
B.7.9.	Invalid inputs for scalar_mult_vfy	71
Authors'	Addresses	71

1. Introduction

This document describes CPace which is a protocol for two parties for deriving a strong shared secret from a shared low-entropy secret (password) without exposing the secret to offline dictionary attacks. The CPace design was tailored for efficiency on constrained devices such as secure-element chipsets and considers mitigations with respect to adversaries that might become capable of breaking the discrete logarithm problem on elliptic curves by quantum computers. CPace comes with both game-based and simulation-based proofs, where the latter provides composability guarantees that let CPace run securely in concurrent settings.

2. Requirements Notation

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. Definition CPace

3.1. Setup

For CPace both communication partners need to agree on a common cipher suite. Cipher suites consist of a combination of a hash function H and an elliptic curve environment G . We assume both G and H to come with associated constants and functions as detailed below. To access these we use an object-style notation such as, e.g., `H.b_in_bytes` and `G.sample_scalar()`.

3.1.1. Hash function H

With H we denote a hash function. Common choices for H are SHA-512 [RFC6234] or SHAKE-256 [FIPS202]. (I.e. the hash function outputs octet strings, and not group elements.) For considering both, variable-output-length hashes and fixed-length output hashes, we use the following convention. In case that the hash function is specified for a fixed-size output, we define `H.hash(m,l)` such that it returns the first l octets of the output.

We use the following notation for referring to the specific properties of a hash function H :

- * `H.hash(m,l)` is a function that operates on an input octet string m and returns a hashing result of l octets.
- * `H.b_in_bytes` denotes the default output size in bytes corresponding to the symmetric security level of the hash function. E.g. `H.b_in_bytes = 64` for SHA-512 and SHAKE-256 and `H.b_in_bytes = 32` for SHA-256 and SHAKE-128. We use the notation `H.hash(m) = H.hash(m, H.b_in_bytes)` and let the hash operation output the default length if no explicit length parameter is given.
- * `H.bmax_in_bytes` denotes the `_maximum_` output size in octets supported by the hash function. In case of fixed-size hashes such as SHA-256, this is the same as `H.b_in_bytes`, while there is no such limit for hash functions such as SHAKE-256.
- * `H.s_in_bytes` denotes the `_input block size_` used by H . For instance, for SHA-512 the input block size `s_in_bytes` is 128, while for SHAKE-256 the input block size amounts to 136 bytes.

3.1.2. Group environment G

The group environment G specifies an elliptic curve group (also denoted G for convenience) and associated constants and functions as detailed below. In this document we use multiplicative notation for the group operation.

- * `G.calculate_generator(H,PRS,CI,sid)` denotes a function that outputs a representation of a generator (referred to as "generator" from now on) of the group which is derived from input octet strings PRS, CI, and sid and with the help of the hash function H.
- * `G.sample_scalar()` is a function returning a representation of a scalar (referred to as "scalar" from now on) appropriate as a private Diffie-Hellman key for the group.
- * `G.scalar_mult(y,g)` is a function operating on a scalar y and a group element g. It returns an octet string representation of the group element $Y = g^y$.
- * `G.I` denotes a unique octet string representation of the neutral element of the group. `G.I` is used for detecting and signaling certain error conditions.
- * `G.scalar_mult_vfy(y,g)` is a function operating on a scalar y and a group element g. It returns an octet string representation of the group element g^y . Additionally, `scalar_mult_vfy` specifies validity conditions for y,g and g^y and outputs `G.I` in case they are not met.
- * `G.DSI` denotes a domain-separation identifier string which SHALL be uniquely identifying the group environment G.

3.2. Inputs

- * PRS denotes a password-related octet string which is a MANDATORY input for all CPace instantiations and needs to be available to both parties. Typically PRS is derived from a low-entropy secret such as a user-supplied password (pw) or a personal identification number, e.g. by use of a password-based key derivation function `PRS = PBKDF(pw)`.

- * CI denotes an OPTIONAL octet string identifying a communication channel that needs to be available to both parties. CI can be used for binding a CPace execution to one specific channel. Typically CI is obtained by concatenating strings that uniquely identify the protocol partner's identities, such as their networking addresses.
- * sid denotes an OPTIONAL octet string serving as session identifier that needs to be available to both parties. In application scenarios where a higher-level protocol has established a unique sid value, this parameter can be used to ensure strong composability guarantees of CPace, and to bind a CPace execution to the application.
- * ADa and ADb denote OPTIONAL octet strings containing arbitrary associated data, each available to one of the parties. They are not required to be equal, and are publicly transmitted as part of the protocol flow. ADa and ADb can for instance include party identifiers or protocol version information (to avoid, e.g., downgrade attacks). In a setting with initiator and responder roles, the information ADa sent by the initiator can be used by the responder for identifying which among possibly several different PRS to use for the CPace session.

3.3. Notation

- * bytes1 || bytes2 denotes concatenation of octet strings.
- * oCat(bytes1,bytes2) denotes ordered concatenation of octet strings, which places the lexicographically larger octet string first. (Explicit code for this function is given in the appendix.)
- * concat(MSGa,MSGb) denotes a concatenation method allows both parties to concatenate CPace's protocol messages in the same way. In applications where CPace is used without clear initiator and responder roles, i.e. where the ordering of messages is not enforced by the protocol flow, concat(MSGa,MSGb) = oCat(MSGa,MSGb) SHALL be used. In settings where the protocol flow enforces ordering, concat(MSGa,MSGb) SHOULD BE implemented such that the later message is appended to the earlier message, i.e., concat(MSGa,MSGb) = MSGa||MSGb if MSGa is sent first.
- * len(S) denotes the number of octets in a string S.
- * nil denotes an empty octet string, i.e., len(nil) = 0.

- * `prepend_len(octet_string)` denotes the octet sequence that is obtained from prepending the length of the octet string to the string itself. The length shall be prepended by using an LEB128 encoding of the length. This will result in a single-byte encoding for values below 128. (Test vectors and reference implementations are given in the appendix.)
- * `prefix_free_cat(a0,a1, ...)` denotes a function that outputs the prefix-free encoding of all input octet strings as the concatenation of the individual strings with their respective length prepended: `prepend_len(a0) || prepend_len(a1) || ...`. Such prefix-free encoding of multiple substrings allows for parsing individual subcomponents of a network message. (Test vectors and reference implementations are given in the appendix.)
- * `sample_random_bytes(n)` denotes a function that returns `n` octets uniformly distributed between 0 and 255.
- * `zero_bytes(n)` denotes a function that returns `n` octets with value 0.

3.3.1. Notation for group operations

We use multiplicative notation for the group, i.e., X^2 denotes the element that is obtained by computing $X * X$, for group element X and group operation $*$.

4. The CPace protocol

CPace is a one round protocol between two parties, A and B. At invocation, A and B are provisioned with PRS,G,H and OPTIONAL public CI,sid,ADa (for A) and CI,sid,ADb (for B). A sends a message MSGa to B. MSGa contains the public share Ya and OPTIONAL associated data ADa (i.e. an ADa field that MAY have a length of 0 bytes). Likewise, B sends a message MSGb to A. MSGb contains the public share Yb and OPTIONAL associated data ADb (i.e. an ADb field that MAY have a length of 0 bytes). Both A and B use the received messages for deriving a shared intermediate session key, ISK. Naming of this key as "intermediate" session key highlights the fact that it is RECOMMENDED to process ISK by use of a suitable strong key derivation function KDF (such as defined in [RFC5869]) first, before using the key in a higher-level protocol.

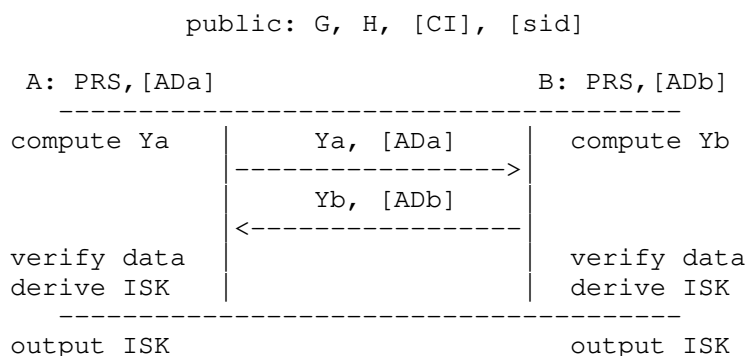
4.1. Session identifier establishment

It is RECOMMENDED to establish a unique session identifier `sid` in the course of the higher-level protocol that invokes CPace, by concatenating random bytes produced by A with random bytes produced by B. In settings where such establishment is not an option, we can let initiator A choose a fresh random `sid` and send it to B together with the first message. This method works whenever the message produced by party A comes first.

The `sid` string SHOULD HAVE a length of at least 8 bytes and it MAY also be the empty string, `nil`. I.e., use of the `sid` string is OPTIONAL.

4.2. Protocol flow

Optional parameters and messages are denoted with `[]`.



4.3. CPace protocol instructions

A computes a generator `g = G.calculate_generator(H, PRS, CI, sid)`, scalar `ya = G.sample_scalar()` and group element `Ya = G.scalar_mult(ya, g)`. A then transmits `MSGa = prefix_free_cat(Ya, ADa)` with optional associated data `ADa` to B. `ADa` MAY have length zero.

B computes a generator `g = G.calculate_generator(H, PRS, CI, sid)`, scalar `yb = G.sample_scalar()` and group element `Yb = G.scalar_mult(yb, g)`. B sends `MSGb = prefix_free_cat(Yb, ADb)` to A.

Note that as `prefix_free_cat` prepends the respectively length of the input fields, the receivers can parse `MSGa` and `MSGb` for subcomponents.

Upon reception of MSGa, B checks that MSGa was properly generated by `prefix_free_cat`. I.e. it checks that the actual length of MSGa matches the sum of the decoded prepended lengths for Ya and ADa. If this parsing fails, then B MUST abort. (Testvectors of examples for invalid messages are given in the appendix.) B then computes $K = G.\text{scalar_mult_vfy}(y_b, Y_a)$. B MUST abort if $K=G.I$. Otherwise B returns $ISK = H.\text{hash}(\text{prefix_free_cat}(G.DSI || \text{"_ISK"}, \text{sid}, K) || \text{concat}(\text{MSGa}, \text{MSGb}))$. B returns ISK and terminates.

Upon reception of MSGb, A parses MSGb for Yb and ADb. I.e. it checks that the actual length of MSGb matches the sum of the decoded prepended lengths for Yb and ADb. If this parsing fails, then A MUST abort. A then computes $K = G.\text{scalar_mult_vfy}(y_a, Y_b)$. A MUST abort if $K=G.I$. Otherwise A returns $ISK = H.\text{hash}(\text{prefix_free_cat}(G.DSI || \text{"_ISK"}, \text{sid}, K) || \text{concat}(\text{MSGa}, \text{MSGb}))$. A returns ISK and terminates.

The session key ISK returned by A and B is identical if and only if the supplied input parameters PRS, CI and sid match on both sides and transcript view (containing of MSGa and MSGb) of both parties match.

We note that the above protocol instructions implement a parallel setting with no specific initiator/responder and no assumptions about the order in which messages arrive. If implemented as initiator-responder protocol, the responder, say, B, starts with computation of the generator only upon reception of MSGa.

5. CPace cipher suites

This section documents RECOMMENDED CPace cipher suite configurations. Any cipher suite configuration for CPace is REQUIRED to specify

* A group environment G specified by

- Functions `G.sample_scalar()`, `G.scalar_mult()`, `G.scalar_mult_vfy()` and `G.calculate_generator()`
- A neutral element `G.I`
- A domain separation identifier string `G.DSI` unique for this cipher suite.

* A hash function H specified by

- Function `H.hash()`
- Constants `H.b_in_bytes`, `H.bmax_in_bytes` and `H.s_in_bytes`

For naming cipher suites we use the convention "CPACE-G-H". Currently, test vectors are available for the following RECOMMENDED cipher suites:

- * CPACE-X25519-SHA512. This suite uses curve G_X25519 defined in Section 6.2 and SHA-512 as hash function.
- * CPACE-X448-SHAKE256. This suite uses curve G_X448 defined in Section 6.2 and SHAKE-256 as hash function.
- * CPACE-P256_XMD:SHA-256_SSWU_NU_-SHA256. This suite instantiates G as specified in Section 6.4 using the `encode_to_curve` function P256_XMD:SHA-256_SSWU_NU_ from [I-D.irtf-cfrg-hash-to-curve] on curve NIST-P256, and hash function SHA-256.
- * CPACE-P384_XMD:SHA-384_SSWU_NU_-SHA384. This suite instantiates G as specified in Section 6.4 using the `encode_to_curve` function P384_XMD:SHA-384_SSWU_NU_ from [I-D.irtf-cfrg-hash-to-curve] on curve NIST-P384 with H = SHA-384.
- * CPACE-P521_XMD:SHA-512_SSWU_NU_-SHA512. This suite instantiates G as specified in Section 6.4 using the `encode_to_curve` function P521_XMD:SHA-384_SSWU_NU_ from [I-D.irtf-cfrg-hash-to-curve] on curve NIST-P384 with H = SHA-512.
- * CPACE-RISTR255-SHA512. This suite uses G_ristretto255 defined in Section 6.3 and H = SHA-512.
- * CPACE-DECAF448-SHAKE256 This suite uses G_decaf448 defined in Section 6.3 and H = SHAKE-256.

CPace can securely be implemented on further elliptic curves when following the guidance given in Section 8.

6. Implementation of recommended CPace cipher suites

6.1. Common function for computing generators

The different cipher suites for CPace defined in the upcoming sections share the same method for deterministically combining the individual strings PRS, CI, sid and the domain-separation identifier DSI to a generator string that we describe here. Let CPACE-G-H denote the cipher suite.

- * `generator_string(G.DSI, PRS, CI, sid, s_in_bytes)` denotes a function that returns the string `prefix_free_cat(G.DSI, PRS, zero_bytes(len_zpad), CI, sid)`.

```
* len_zpad = MAX(0, s_in_bytes - len(prepend_len(PRS)) -
  len(prepend_len(G.DSI)) - 1)
```

The zero padding of length `len_zpad` is designed such that the encoding of `G.DSI` and `PRS` together with the zero padding field completely fills the first input block (of length `s_in_bytes`) of the hash. As a result the number of bytes to hash becomes independent of the actual length of the password (`PRS`). (A reference implementation and test vectors are provided in the appendix.)

The introduction of a zero-padding within the generator string also helps mitigating attacks of a side-channel adversary that analyzes correlations between publicly known variable information with the low-entropy `PRS` string. Note that the hash of the first block is intentionally made independent of session-specific inputs, such as `sid` or `CI`.

6.2. CPace group objects `G_X25519` and `G_X448` for single-coordinate Ladders on Montgomery curves

In this section we consider the case of CPace when using the `X25519` and `X448` Diffie-Hellman functions from [RFC7748] operating on the Montgomery curves `Curve25519` and `Curve448` [RFC7748]. CPace implementations using single-coordinate ladders on further Montgomery curves SHALL use the definitions in line with the specifications for `X25519` and `X448` and review the guidance given in Section 8.

For the group environment `G_X25519` the following definitions apply:

```
* G_X25519.field_size_bytes = 32
* G_X25519.field_size_bits = 255
* G_X25519.sample_scalar() = sample_random_bytes(G.field_size_bytes)
* G_X25519.scalar_mult(y,g) = G.scalar_mult_vfy(y,g) = X25519(y,g)
* G_X25519.I = zero_bytes(G.field_size_bytes)
* G_X25519.DSI = "CPace255"
```

CPace cipher suites using `G_X25519` MUST use a hash function producing at least `H.b_max_in_bytes` ≥ 32 bytes of output. It is RECOMMENDED to use `G_X25519` in combination with SHA-512.

For `X448` the following definitions apply:

```
* G_X448.field_size_bytes = 56
```

```
* G_X448.field_size_bits = 448
* G_X448.sample_scalar() = sample_random_bytes(G.field_size_bytes)
* G_X448.scalar_mult(y,g) = G.scalar_mult_vfy(y,g) = X448(y,g)
* G_X448.I = zero_bytes(G.field_size_bytes)
* G_X448.DSI = "CPace448"
```

CPace cipher suites using G_X448 MUST use a hash function producing at least $H.b_max_in_bytes \geq 56$ bytes of output. It is RECOMMENDED to use G_X448 in combination with SHAKE-256.

For both G_X448 and G_X25519 the `G.calculate_generator(H, PRS,sid,CI)` function shall be implemented as follows.

- * First `gen_str = generator_string(G.DSI,PRS,CI,sid, H.s_in_bytes)` SHALL BE calculated using the input block size of the chosen hash function.
- * This string SHALL then BE hashed to the required length `gen_str_hash = H.hash(gen_str, G.field_size_bytes)`. Note that this implies that the permissible output length `H.maxb_in_bytes` MUST BE larger or equal to the field size of the group G for making a hashing function suitable.
- * This result is then considered as a field coordinate using the `u = decodeUCoordinate(gen_str_hash, G.field_size_bits)` function from [RFC7748] which we repeat in the appendix for convenience.
- * The result point `g` is then calculated as `(g,v) = map_to_curve_elligator2(u)` using the function from [I-D.irtf-cfrg-hash-to-curve]. Note that the `v` coordinate produced by the `map_to_curve_elligator2` function is not required for CPace and discarded. The appendix repeats the definitions from [I-D.irtf-cfrg-hash-to-curve] for convenience.

In the appendix we show sage code that can be used as reference implementation.

6.2.1. Verification tests

For single-coordinate Montgomery ladders on Montgomery curves verification tests according to Section 7 SHALL consider the `u` coordinate values that encode a low-order point on either, the curve or the quadratic twist.

In addition to that in case of G_X25519 the tests SHALL also verify that the implementation of G.scalar_mult_vfy(y,g) produces the expected results for non-canonical u coordinate values with bit #255 set, which also encode low-order points.

Corresponding test vectors are provided in the appendix.

6.3. CPace group objects G_Ristretto255 and G-Decaf448 for prime-order group abstractions

In this section we consider the case of CPace using the Ristretto255 and Decaf448 group abstractions [I-D.draft-irtf-cfrg-ristretto255-decaf448]. These abstractions define an encode and decode function, group operations using an internal encoding and a one-way-map. With the group abstractions there is a distinction between an internal representation of group elements and an external encoding of the same group element. In order to distinguish between these different representations, we prepend an underscore before values using the internal representation within this section.

For Ristretto255 the following definitions apply:

```
* G_Ristretto255.DSI = "CPaceRistretto255"
* G_Ristretto255.field_size_bytes = 32
* G_Ristretto255.group_size_bits = 252
* G_Ristretto255.group_order = 2^252 +
  27742317777372353535851937790883648493
```

CPace cipher suites using G_Ristretto255 MUST use a hash function producing at least H.b_max_in_bytes >= 64 bytes of output. It is RECOMMENDED to use G_Ristretto255 in combination with SHA-512.

For decaf448 the following definitions apply:

```
* G-Decaf448.DSI = "CPaceDecaf448"
* G-Decaf448.field_size_bytes = 56
* G-Decaf448.group_size_bits = 445
* G-Decaf448.group_order = l = 2^446 -
  1381806680989511535200738674851542
  6880336692474882178609894547503885
```

CPace cipher suites using G_Decaf448 MUST use a hash function producing at least $H.b_max_in_bytes \geq 112$ bytes of output. It is RECOMMENDED to use G_Decaf448 in combination with SHAKE-256.

For both abstractions the following definitions apply:

- * It is RECOMMENDED to implement `G.sample_scalar()` as follows.
 - Set `scalar = sample_random_bytes(G.group_size_bytes)`.
 - Then clear the most significant bits larger than `G.group_size_bits`.
 - Interpret the result as the little-endian encoding of an integer value and return the result.
- * Alternatively, if `G.sample_scalar()` is not implemented according to the above recommendation, it SHALL be implemented using uniform sampling between 1 and $(G.group_order - 1)$. Note that the more complex uniform sampling process can provide a larger side-channel attack surface for embedded systems in hostile environments.
- * `G.scalar_mult(y, _g)` SHALL operate on a scalar `y` and a group element `_g` in the internal representation of the group abstraction environment. It returns the value $Y = encode((_g)^y)$, i.e. it returns a value using the public encoding.
- * `G.I` = is the public encoding representation of the identity element.
- * `G.scalar_mult_vfy(y, X)` operates on a value using the public encoding and a scalar and is implemented as follows. If the `decode(X)` function fails, it returns `G.I`. Otherwise it returns `encode(decode(X)^y)`.
- * The `G.calculate_generator(H, PRS, sid, CI)` function SHALL return a decoded point and SHALL BE implemented as follows.
 - First `gen_str = generator_string(G.DSI, PRS, CI, sid, H.s_in_bytes)` is calculated using the input block size of the chosen hash function.
 - This string is then hashed to the required length `gen_str_hash = H.hash(gen_str, 2 * G.field_size_bytes)`. Note that this implies that the permissible output length `H.maxb_in_bytes` MUST BE larger or equal to twice the field size of the group `G` for making a hash function suitable.

- Finally the internal representation of the generator `_g` is calculated as `_g = one_way_map(gen_str_hash)` using the one-way map function from the abstraction.

Note that with these definitions the `scalar_mult` function operates on a decoded point `_g` and returns an encoded point, while the `scalar_mult_vfy(y,X)` function operates on an encoded point `X` (and also returns an encoded point).

6.3.1. Verification tests

For group abstractions verification tests according to Section 7 SHALL consider encodings of the neutral element and an octet string that does not decode to a valid group element.

6.4. CPace group objects for curves in Short-Weierstrass representation

The group environment objects `G` defined in this section for use with Short-Weierstrass curves, are parametrized by the choice of an elliptic curve and by choice of a suitable `encode_to_curve(str)` function. `encode_to_curve(str)` must map an octet string `str` to a point on the curve.

6.4.1. Curves and associated functions

Elliptic curves in Short-Weierstrass form are considered in [IEEE1363]. [IEEE1363] allows for both, curves of prime and non-prime order. However, for the procedures described in this section any suitable group MUST BE of prime order.

The specification for the group environment objects specified in this section closely follow the ECKAS-DH1 method from [IEEE1363]. I.e. we use the same methods and encodings and protocol substeps as employed in the TLS [RFC5246] [RFC8446] protocol family.

For CPace only the uncompressed full-coordinate encodings from [SEC1] (`x` and `y` coordinate) SHOULD be used. Commonly used curve groups are specified in [SEC2] and [RFC5639]. A typical representative of such a Short-Weierstrass curve is NIST-P256. Point verification as used in ECKAS-DH1 is described in Annex A.16.10. of [IEEE1363].

For deriving Diffie-Hellman shared secrets ECKAS-DH1 from [IEEE1363] specifies the use of an ECSVDP-DH method. We use ECSVDP-DH in combination with the identity map such that it either returns "error" or the `x`-coordinate of the Diffie-Hellman result point as shared secret in big endian format (fixed length output by FE2OSP without truncating leading zeros).

6.4.2. Suitable encode_to_curve methods

All the encode_to_curve methods specified in [I-D.irtf-cfrg-hash-to-curve] are suitable for CPace. For Short-Weierstrass curves it is RECOMMENDED to use the non-uniform variant of the SSWU mapping primitive from [I-D.irtf-cfrg-hash-to-curve] if a SSWU mapping is available for the chosen curve. (We recommend non-uniform maps in order to give implementations the flexibility to opt for x-coordinate-only scalar multiplication algorithms.)

6.4.3. Definition of the group environment G for Short-Weierstrass curves

In this paragraph we use the following notation for defining the group object G for a selected curve and encode_to_curve method:

- * With group_order we denote the order of the elliptic curve which MUST BE a prime.
- * With is_valid(X) we denote a method which operates on an octet stream according to [SEC1] of a point on the group and returns true if the point is valid or false otherwise. This is_valid(X) method SHALL be implemented according to Annex A.16.10. of [IEEE1363]. I.e. it shall return false if X encodes either the neutral element on the group or does not form a valid encoding of a point on the group.
- * With encode_to_curve(str) we denote a selected mapping function from [I-D.irtf-cfrg-hash-to-curve]. I.e. a function that maps octet string str to a point on the group. [I-D.irtf-cfrg-hash-to-curve] considers both, uniform and non-uniform mappings based on several different strategies. It is RECOMMENDED to use the nonuniform variant of the SSWU mapping primitive within [I-D.irtf-cfrg-hash-to-curve].
- * G.DSI denotes a domain-separation identifier string. G.DSI which SHALL BE obtained by the concatenation of "CPace" and the associated name of the cipher suite used for the encode_to_curve function as specified in [I-D.irtf-cfrg-hash-to-curve]. E.g. when using the map with the name "P384_XMD:SHA-384_SSWU_NU_" on curve NIST-P384 the resulting value SHALL BE G.DSI = "CPaceP384_XMD:SHA-384_SSWU_NU_".

Using the above definitions, the CPace functions required for the group object G are defined as follows.

- * `G.sample_scalar()` SHALL return a value between 1 and $(G.group_order - 1)$. The value sampling MUST BE uniformly random. It is RECOMMENDED to use rejection sampling for converting a uniform bitstring to a uniform value between 1 and $(G.group_order - 1)$.
- * `G.calculate_generator(H, PRS, sid, CI)` function SHALL be implemented as follows.
 - First `gen_str = generator_string(G.DSI, PRS, CI, sid, H.s_in_bytes)` is calculated.
 - Then the output of a call to `encode_to_curve(gen_str)` is returned, using the selected function from [I-D.irtf-cfrg-hash-to-curve].
- * `G.scalar_mult(s, X)` is a function that operates on a scalar s and an input point X . The input X shall use the same encoding as produced by the `G.calculate_generator` method above. `G.scalar_mult(s, X)` SHALL return an encoding of either the point X^s or the point X^{-s} according to [SEC1]. Implementations SHOULD use the full-coordinate format without compression, as important protocols such as TLS 1.3 removed support for compression. Implementations of `scalar_mult(s, X)` MAY output either X^s or X^{-s} as both points X^s and X^{-s} have the same x-coordinate and result in the same Diffie-Hellman shared secrets K . (This allows implementations to opt for x-coordinate-only scalar multiplication algorithms.)
- * `G.scalar_mult_vfy(s, X)` merges verification of point X according to [IEEE1363] A.16.10. and the the ECSVDP-DH procedure from [IEEE1363]. It SHALL BE implemented as follows:
 - If `is_valid(X) = False` then `G.scalar_mult_vfy(s, X)` SHALL return "error" as specified in [IEEE1363] A.16.10 and 7.2.1.
 - Otherwise `G.scalar_mult_vfy(s, X)` SHALL return the result of the ECSVDP-DH procedure from [IEEE1363] (section 7.2.1). I.e. it shall either return "error" (in case that X^s is the neutral element) or the secret shared value "z" (otherwise). "z" SHALL be encoded by using the big-endian encoding of the x-coordinate of the result point X^s according to [SEC1].
- * We represent the neutral element $G.I$ by using the representation of the "error" result case from [IEEE1363] as used in the `G.scalar_mult_vfy` method above.

6.4.4. Verification tests

For Short-Weierstrass curves verification tests according to Section 7 SHALL consider encodings of the point at infinity and an encoding of a point not on the group.

7. Implementation verification

Any CPace implementation MUST be tested against invalid or weak point attacks. Implementation MUST be verified to abort upon conditions where `G.scalar_mult_vfy` functions outputs G.I. For testing an implementation it is RECOMMENDED to include weak or invalid points in MSGa and MSGb and introduce this in a protocol run. It SHALL be verified that the abort condition is properly handled.

Moreover any implementation MUST be tested with respect invalid encodings of MSGa and MSGb where the length of the message does not match the specified encoding (i.e. where the sum of the prepended length information does not match the actual length of the message).

Corresponding test vectors are given in the appendix for all recommended cipher suites.

8. Security Considerations

A security proof of CPace is found in [AHH21]. This proof covers all recommended cipher suites included in this document. In the following sections we describe how to protect CPace against several attack families, such as relay-, length extension- or side channel attacks. We also describe aspects to consider when deviating from recommended cipher suites.

8.1. Party identifiers and relay attacks

If unique strings identifying the protocol partners are included either as part of the channel identifier CI, the session id sid or the associated data fields ADa, ADb, the ISK will provide implicit authentication also regarding the party identities. Incorporating party identifier strings is important for fending off relay attacks. Such attacks become relevant in a setting where several parties, say, A, B and C, share the same password PRS. An adversary might relay messages from a honest user A, who aims at interacting with user B, to a party C instead. If no party identifier strings are used, and B and C use the same PRS value, A might be establishing a common ISK key with C while assuming to interact with party B. Including and checking party identifiers can fend off such relay attacks.

8.2. Hashing and key derivation

In order to prevent analysis of length extension attacks on hash functions, all hash input strings in CPace are designed to be prefix-free strings which have the length of individual substrings prepended, enforced by the `prefix_free_cat()` function. This choice was made in order to make CPace suitable also for hash function instantiations using Merkle-Damgard constructions such as SHA-256 or SHA-512 along the lines of [CDMP05]. In case that an application wishes to use another form of encoding, the guidance given in [CDMP05] SHOULD BE considered.

Although already `K` is a shared value, it MUST NOT itself be used as an application key. Instead, `ISK` MUST BE used. Leakage of `K` to an adversary can lead to offline dictionary attacks.

As noted already in Section 4 it is RECOMMENDED to process `ISK` by use of a suitable strong key derivation function KDF (such as defined in [RFC5869]) first, before using the key in a higher-level protocol.

8.3. Key confirmation

In many applications it is advisable to add an explicit key confirmation round after the CPace protocol flow. However, as some applications might only require implicit authentication and as explicit authentication messages are already a built-in feature in many higher-level protocols (e.g. TLS 1.3) the CPace protocol described here does not mandate use of a key confirmation on the level of the CPace sub-protocol.

Already without explicit key confirmation, CPace enjoys weak forward security under the sCDH and sSDH assumptions [AHH21]. With added explicit confirmation, CPace enjoys perfect forward security also under the strong sCDH and sSDH assumptions [AHH21].

Note that in [ABKLX21] it was shown that an idealized variant of CPace also enjoys perfect forward security without explicit key confirmation. However this proof does not explicitly cover the recommended cipher suites in this document and requires the stronger assumption of an algebraic adversary model. For this reason, we recommend adding explicit key confirmation if perfect forward security is required.

When implementing explicit key confirmation, it is recommended to use an appropriate message-authentication code (MAC) such as HMAC [RFC2104] or CMAC [RFC4493] using a key `mac_key` derived from `ISK`.

One suitable option that works also in the parallel setting without message ordering is to proceed as follows.

- * First calculate `mac_key` as `mac_key = H.hash(b"CPaceMac" || ISK)`.
- * Then let each party send an authenticator tag `Ta`, `Tb` that is calculated over the protocol message that it has sent previously. I.e. let party A calculate its transmitted authentication code `Ta` as `Ta = MAC(mac_key, MSGa)` and let party B calculate its transmitted authentication code `Tb` as `Tb = MAC(mac_key, MSGb)`.
- * Let the receiving party check the remote authentication tag for the correct value and abort in case that it's incorrect.

8.4. Sampling of scalars

For curves over fields F_p where p is a prime close to a power of two, we recommend sampling scalars as a uniform bit string of length `field_size_bits`. We do so in order to reduce both, complexity of the implementation and reducing the attack surface with respect to side-channels for embedded systems in hostile environments. The effect of non-uniform sampling on security was demonstrated to be beginning in [AHH21] for the case of Curve25519 and Curve448. This analysis however does not transfer to most curves in Short-Weierstrass form. As a result, we recommend rejection sampling if G is as in Section 6.4.

8.5. Single-coordinate CPace on Montgomery curves

The recommended cipher suites for the Montgomery curves Curve25519 and Curve448 in Section 6.2 rely on the following properties [AHH21]:

- * The curve has order $(p * c)$ with p prime and c a small cofactor. Also the curve's quadratic twist must be of order $(p' * c')$ with p' prime and c' a cofactor.
- * The cofactor c' of the twist MUST BE EQUAL to or an integer multiple of the cofactor c of the curve.
- * Both field order q and group order p MUST BE close to a power of two along the lines of [AHH21], Appendix E.
- * The representation of the neutral element G .I MUST BE the same for both, the curve and its twist.

- * The implementation of `G.scalar_mult_vfy(y,X)` MUST map all `c` low-order points on the curve and all `c'` low-order points on the twist to `G.I`.

Montgomery curves other than the ones recommended here can use the specifications given in Section 6.2, given that the above properties hold.

8.6. Nonce values

Secret scalars `ya` and `yb` MUST NOT be reused. Values for `sid` SHOULD NOT be reused since the composability guarantees established by the simulation-based proof rely on the uniqueness of session ids [AHH21].

If CPace is used in a concurrent system, it is RECOMMENDED that a unique `sid` is generated by the higher-level protocol and passed to CPace. One suitable option is that `sid` is generated by concatenating ephemeral random strings contributed by both parties.

8.7. Side channel attacks

All state-of-the art methods for realizing constant-time execution SHOULD be used. In case that side channel attacks are to be considered practical for a given application, it is RECOMMENDED to pay special attention on computing the secret generator `G.calculate_generator(PRS,CI,sid)`. The most critical substep to consider might be the processing of the first block of the hash that includes the PRS string. The zero-padding introduced when hashing the sensitive PRS string can be expected to make the task for a side-channel attack somewhat more complex. Still this feature alone is not sufficient for ruling out power analysis attacks.

8.8. Quantum computers

CPace is proven secure under the hardness of the strong computational Simultaneous Diffie-Hellmann (sSDH) and strong computational Diffie-Hellmann (sCDH) assumptions in the group `G` (as defined in [AHH21]). These assumptions are not expected to hold any longer when large-scale quantum computers (LSQC) are available. Still, even in case that LSQC emerge, it is reasonable to assume that discrete-logarithm computations will remain costly. CPace with ephemeral session id values `sid` forces the adversary to solve one computational Diffie-Hellman problem per password guess [ES21]. In this sense, using the wording suggested by Steve Thomas on the CFRG mailing list, CPace is "quantum-annoying".

9. IANA Considerations

No IANA action is required.

10. Acknowledgements

Thanks to the members of the CFRG for comments and advice. Any comment and advice is appreciated.

11. References

11.1. Normative References

- [I-D.draft-irtf-cfrg-ristretto255-decaf448]
Valence, H. D., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I., and M. Hamburg, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-01, 4 August 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-01>>.
- [I-D.irtf-cfrg-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-13, 10 November 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13>>.
- [IEEE1363] "Standard Specifications for Public Key Cryptography, IEEE 1363", 2000.
- [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>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/rfc/rfc7748>>.
- [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>>.
- [SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", May 2009, <<http://www.secg.org/sec1-v2.pdf>>.

11.2. Informative References

- [ABKLX21] Abdalla, M., Barbosa, M., Katz, J., Loss, J., and J. Xu, "Algebraic Adversaries in the Universal Composability Framework.", n.d., <<https://eprint.iacr.org/2021/1218>>.
- [AHH21] Abdalla, M., Haase, B., and J. Hesse, "Security analysis of CPace", n.d., <<https://eprint.iacr.org/2021/114>>.
- [CDMP05] Coron, J-S., Dodis, Y., Malinaud, C., and P. Puniya, "Merkle-Damgaard Revisited: How to Construct a Hash Function", In Advances in Cryptology - CRYPTO 2005, pages 430-448, DOI 10.1007/11535218_26, 2005, <https://doi.org/10.1007/11535218_26>.
- [ES21] Eaton, E. and D. Stebila, "The 'quantum annoying' property of password-authenticated key exchange protocols.", n.d., <<https://eprint.iacr.org/2021/696>>.
- [FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", August 2015, <<https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>>.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/rfc/rfc2104>>.
- [RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <<https://www.rfc-editor.org/rfc/rfc4493>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/rfc/rfc5246>>.
- [RFC5639] Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <<https://www.rfc-editor.org/rfc/rfc5639>>.
- [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>>.

- [RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <<https://www.rfc-editor.org/rfc/rfc6234>>.
- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [SEC2] Standards for Efficient Cryptography Group (SECG), "SEC 2: Recommended Elliptic Curve Domain Parameters", January 2010, <<http://www.secg.org/sec2-v2.pdf>>.

Appendix A. CPace function definitions

A.1. Definition and test vectors for string utility functions

A.1.1. prepend_len function

```
def prepend_len(data):
    "prepend LEB128 encoding of length"
    length = len(data)
    length_encoded = b""
    while True:
        if length < 128:
            length_encoded += bytes([length])
        else:
            length_encoded += bytes([(length & 0x7f) + 0x80])
            length = int(length >> 7)
        if length == 0:
            break;
    return length_encoded + data
```

A.1.2. prepend_len test vectors

```

prepend_len(b''): (length: 1 bytes)
00
prepend_len(b"1234"): (length: 5 bytes)
0431323334
prepend_len(bytes(range(127))): (length: 128 bytes)
7f000102030405060708090a0b0c0d0e0f101112131415161718191a1b
1c1d1e1f202122232425262728292a2b2c2d2e2f303132333435363738
393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f505152535455
565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f707172
737475767778797a7b7c7d7e
prepend_len(bytes(range(128))): (length: 130 bytes)
8001000102030405060708090a0b0c0d0e0f101112131415161718191a
1b1c1d1e1f202122232425262728292a2b2c2d2e2f3031323334353637
38393a3b3c3d3e3f404142434445464748494a4b4c4d4e4f5051525354
55565758595a5b5c5d5e5f606162636465666768696a6b6c6d6e6f7071
72737475767778797a7b7c7d7e7f

```

A.1.3. prefix_free_cat function

```

def prefix_free_cat(*args):
    result = b''
    for arg in args:
        result += prepend_len(arg)
    return result

```

A.1.4. Testvector for prefix_free_cat()

```

prefix_free_cat(b"1234",b"5",b"",b"6789"):
(length: 13 bytes)
04313233340135000436373839

```

A.1.5. Examples for invalid encoded messages

The following messages are examples which have invalid encoded length fields. I.e. they are examples where parsing for the sum of the length of subfields as expected for a message generated for the prefix free concatenation does not give the correct length of the message. Parties MUST abort upon reception of such invalid messages as MSGa or MSGb.

```

Inv_MSG1 with invalid encoded length: (length: 3 bytes)
ffffff
Inv_MSG2 with invalid encoded length: (length: 3 bytes)
ffff03
Inv_MSG3 with invalid encoded length: (length: 4 bytes)
00ffff03
Inv_MSG4 with invalid encoded length: (length: 4 bytes)
00ffffff

```

A.2. Definition of generator_string function.

```
def generator_string(DSI,PRS,CI,sid,s_in_bytes):
    # Concat all input fields with prepended length information.
    # Add zero padding in the first hash block after DSI and PRS.
    len_zpad = max(0,s_in_bytes - 1 - len(prepend_len(PRS))
                  - len(prepend_len(DSI)))
    return prefix_free_cat(DSI, PRS, zero_bytes(len_zpad),
                          CI, sid)
```

A.3. Definitions and test vector ordered concatenation

A.3.1. Definitions for lexicographical ordering

For ordered concatenation lexicographical ordering of byte sequences is used:

```
def lexicographically_larger(bytes1,bytes2):
    "Returns True if bytes1 > bytes2 using lexicographical ordering."
    min_len = min (len(bytes1), len(bytes2))
    for m in range(min_len):
        if bytes1[m] > bytes2[m]:
            return True;
        elif bytes1[m] < bytes2[m]:
            return False;
    return len(bytes1) > len(bytes2)
```

A.3.2. Definitions for ordered concatenation

With the above definition of lexicographical ordering ordered concatenation is specified as follows.

```
def oCAT(bytes1,bytes2):
    if lexicographically_larger(bytes1,bytes2):
        return bytes1 + bytes2
    else:
        return bytes2 + bytes1
```

A.3.3. Test vectors ordered concatenation

```

string comparison for oCAT:
lexicographically_larger(b"\0", b"\0\0") == False
lexicographically_larger(b"\1", b"\0\0") == True
lexicographically_larger(b"\0\0", b"\0") == True
lexicographically_larger(b"\0\0", b"\1") == False
lexicographically_larger(b"\0\1", b"\1") == False
lexicographically_larger(b"ABCD", b"BCD") == False

```

```

oCAT(b"ABCD",b"BCD"): (length: 7 bytes)
42434441424344
oCAT(b"BCD",b"ABCDE"): (length: 8 bytes)
4243444142434445

```

A.4. Decoding and Encoding functions according to RFC7748

```

def decodeLittleEndian(b, bits):
    return sum([b[i] << 8*i for i in range((bits+7)/8)])

def decodeUCoordinate(u, bits):
    u_list = [ord(b) for b in u]
    # Ignore any unused bits.
    if bits % 8:
        u_list[-1] &= (1<<(bits%8))-1
    return decodeLittleEndian(u_list, bits)

def encodeUCoordinate(u, bits):
    u = u % p
    return ''.join([chr((u >> 8*i) & 0xff)
                    for i in range((bits+7)/8)])

```

A.5. Elligator 2 reference implementation

The Elligator 2 map requires a non-square field element Z which shall be calculated as follows.

```

def find_z_ell2(F):
    # Find nonsquare for Elligator2
    # Argument: F, a field object, e.g., F = GF(2^255 - 19)
    ctr = F.gen()
    while True:
        for Z_cand in (F(ctr), F(-ctr)):
            # Z must be a non-square in F.
            if is_square(Z_cand):
                continue
            return Z_cand
        ctr += 1

```

The values of the non-square Z only depend on the curve. The algorithm above results in a value of $Z = 2$ for Curve25519 and $Z=-1$ for Ed448.

The following code maps a field element r to an encoded field element which is a valid u -coordinate of a Montgomery curve with curve parameter A .

```
def elligator2(r, q, A, field_size_bits):
    # Inputs: field element r, field order q,
    #         curve parameter A and field size in bits
    Fq = GF(q); A = Fq(A); B = Fq(1);

    # get non-square z as specified in the hash2curve draft.
    z = Fq(find_z_ell2(Fq))
    powerForLegendreSymbol = floor((q-1)/2)

    v = - A / (1 + z * r^2)
    epsilon = (v^3 + A * v^2 + B * v)^powerForLegendreSymbol
    x = epsilon * v - (1 - epsilon) * A/2
    return encodeUCoordinate(Integer(x), field_size_bits)
```

Appendix B. Test vectors

B.1. Test vector for CPace using group X25519 and hash SHA-512

B.1.1. Test vectors for calculate_generator with group X25519

Inputs

```

H = SHA-512 with input block size 128 bytes.
PRS = b'Password' ; ZPAD length: 109 ; DSI = b'CPace255'
CI = b'\nAinitiator\nBresponder'
CI = 0a41696e69746961746f720a42726573706f6e646572
sid = 7e4b4791d6a8ef019b936c79fb7f2c57

```

Outputs

```

generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
(length: 168 bytes)
0843506163653235350850617373776f72646d00000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0a42726573706f6e646572107e4b4791d6a8ef019b936c79fb7f2c57
hash generator string: (length: 32 bytes)
10047198e8c4cacf0ab8a6d0ac337b8ae497209d042f7f3a50945863
94e821fc
decoded field element of 255 bits: (length: 32 bytes)
10047198e8c4cacf0ab8a6d0ac337b8ae497209d042f7f3a50945863
94e8217c
generator g: (length: 32 bytes)
4e6098733061c0e8486611a904fe5edb049804d26130a44131a6229e
55c5c321

```

B.1.2. Test vector for MSGa

Inputs

```

ADa = b'ADa'
ya (little endian): (length: 32 bytes)
45acf93116ae5d3dae995a7c627df2924321a8e857d9a200807131e3
8839b0c2

```

Outputs

```

Ya: (length: 32 bytes)
6f7fd31863b18b0cc9830fc842c60dea80120ccf2fd375498225e45a
52065361
MSGa: (length: 37 bytes)
206f7fd31863b18b0cc9830fc842c60dea80120ccf2fd375498225e4
5a5206536103414461

```

B.1.3. Test vector for MSGb

Inputs

```
ADb = b'ADb'
yb (little endian): (length: 32 bytes)
a145e914b347002d298ce2051394f0ed68cf3623dfe5db082c78ffa5
a667acdc
```

Outputs

```
Yb: (length: 32 bytes)
e1b730a4956c0f853d96c5d125cebeeea46952c07c6f66da65bd9ffd
2f71a462
MSGb: (length: 37 bytes)
20e1b730a4956c0f853d96c5d125cebeeea46952c07c6f66da65bd9f
fd2f71a46203414462
```

B.1.4. Test vector for secret points K

```
scalar_mult_vfy(ya,Yb): (length: 32 bytes)
2a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86d9e199
befa6024
scalar_mult_vfy(yb,Ya): (length: 32 bytes)
2a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86d9e199
befa6024
```

B.1.5. Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 74 bytes)
206f7fd31863b18b0cc9830fc842c60dea80120ccf2fd375498225e4
5a520653610341446120e1b730a4956c0f853d96c5d125cebeeea469
52c07c6f66da65bd9ffd2f71a46203414462
DSI = G.DSI_ISK, b'CPace255_ISK': (length: 12 bytes)
43506163653235355f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 137 bytes)
0c43506163653235355f49534b107e4b4791d6a8ef019b936c79fb7f
2c57202a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86
d9e199befa6024206f7fd31863b18b0cc9830fc842c60dea80120ccf
2fd375498225e45a520653610341446120e1b730a4956c0f853d96c5
d125cebeeea46952c07c6f66da65bd9ffd2f71a46203414462
ISK result: (length: 64 bytes)
99a9e0ff35acb94ad8af1cd6b32ac409dc7d00557ccd9a7d19d3b462
9e5f1f084f9332096162438c7ecc78331b4eda17e1a229a47182eccc
9ea58cd9cdcd8e9a
```

B.1.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 74 bytes)
20e1b730a4956c0f853d96c5d125cebeeeaa46952c07c6f66da65bd9f
fd2f71a46203414462206f7fd31863b18b0cc9830fc842c60dea8012
0ccf2fd375498225e45a5206536103414461
DSI = G.DSI_ISK, b'CPace255_ISK': (length: 12 bytes)
43506163653235355f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb) :
(length: 137 bytes)
0c43506163653235355f49534b107e4b4791d6a8ef019b936c79fb7f
2c57202a905bc5f0b93ee72ac4b6ea8723520941adfc892935bf6f86
d9e199befa602420e1b730a4956c0f853d96c5d125cebeeeaa46952c0
7c6f66da65bd9ffd2f71a46203414462206f7fd31863b18b0cc9830f
c842c60dea80120ccf2fd375498225e45a5206536103414461
ISK result: (length: 64 bytes)
3cd6a9670fa3ff211d829b845baa0f5ba9ad580c3ba0ee790bd0e9cd
556290a8ffce44419fbf94e4cb8e7fe9f454fd25dc13e689e4d6ab0a
c2211c70a8ac0062

```

B.1.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t tc_CI[] = {
    0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
    0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const uint8_t tc_sid[] = {
    0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
    0xfb,0x7f,0x2c,0x57,
};
const uint8_t tc_g[] = {
    0x4e,0x60,0x98,0x73,0x30,0x61,0xc0,0xe8,0x48,0x66,0x11,0xa9,
    0x04,0xfe,0x5e,0xdb,0x04,0x98,0x04,0xd2,0x61,0x30,0xa4,0x41,
    0x31,0xa6,0x22,0x9e,0x55,0xc5,0xc3,0x21,
};
const uint8_t tc_ya[] = {
    0x45,0xac,0xf9,0x31,0x16,0xae,0x5d,0x3d,0xae,0x99,0x5a,0x7c,
    0x62,0x7d,0xf2,0x92,0x43,0x21,0xa8,0xe8,0x57,0xd9,0xa2,0x00,
    0x80,0x71,0x31,0xe3,0x88,0x39,0xb0,0xc2,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x6f,0x7f,0xd3,0x18,0x63,0xb1,0x8b,0x0c,0xc9,0x83,0x0f,0xc8,
    0x42,0xc6,0x0d,0xea,0x80,0x12,0x0c,0xcf,0x2f,0xd3,0x75,0x49,
    0x82,0x25,0xe4,0x5a,0x52,0x06,0x53,0x61,
};

```

```

};
const uint8_t tc_yb[] = {
    0xa1,0x45,0xe9,0x14,0xb3,0x47,0x00,0x2d,0x29,0x8c,0xe2,0x05,
    0x13,0x94,0xf0,0xed,0x68,0xcf,0x36,0x23,0xdf,0xe5,0xdb,0x08,
    0x2c,0x78,0xff,0xa5,0xa6,0x67,0xac,0xdc,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0xe1,0xb7,0x30,0xa4,0x95,0x6c,0x0f,0x85,0x3d,0x96,0xc5,0xd1,
    0x25,0xce,0xbe,0xee,0xa4,0x69,0x52,0xc0,0x7c,0x6f,0x66,0xda,
    0x65,0xbd,0x9f,0xfd,0x2f,0x71,0xa4,0x62,
};
const uint8_t tc_K[] = {
    0x2a,0x90,0x5b,0xc5,0xf0,0xb9,0x3e,0xe7,0x2a,0xc4,0xb6,0xea,
    0x87,0x23,0x52,0x09,0x41,0xad,0xfc,0x89,0x29,0x35,0xbf,0x6f,
    0x86,0xd9,0xe1,0x99,0xbe,0xfa,0x60,0x24,
};
const uint8_t tc_ISK_IR[] = {
    0x99,0xa9,0xe0,0xff,0x35,0xac,0xb9,0x4a,0xd8,0xaf,0x1c,0xd6,
    0xb3,0x2a,0xc4,0x09,0xdc,0x7d,0x00,0x55,0x7c,0xcd,0x9a,0x7d,
    0x19,0xd3,0xb4,0x62,0x9e,0x5f,0x1f,0x08,0x4f,0x93,0x32,0x09,
    0x61,0x62,0x43,0x8c,0x7e,0xcc,0x78,0x33,0x1b,0x4e,0xda,0x17,
    0xe1,0xa2,0x29,0xa4,0x71,0x82,0xec,0xcc,0x9e,0xa5,0x8c,0xd9,
    0xcd,0xcd,0x8e,0x9a,
};
const uint8_t tc_ISK_SY[] = {
    0x3c,0xd6,0xa9,0x67,0x0f,0xa3,0xff,0x21,0x1d,0x82,0x9b,0x84,
    0x5b,0xaa,0x0f,0x5b,0xa9,0xad,0x58,0x0c,0x3b,0xa0,0xee,0x79,
    0x0b,0xd0,0xe9,0xcd,0x55,0x62,0x90,0xa8,0xff,0xce,0x44,0x41,
    0x9f,0xbf,0x94,0xe4,0xcb,0x8e,0x7f,0xe9,0xf4,0x54,0xfd,0x25,
    0xdc,0x13,0xe6,0x89,0xe4,0xd6,0xab,0x0a,0xc2,0x21,0x1c,0x70,
    0xa8,0xac,0x00,0x62,
};

```

B.1.8. Test vectors for G_X25519.scalar_mult_vfy: low order points

Test vectors for which G_X25519.scalar_mult_vfy(s_in,ux) must return the neutral element or would return the neutral element if bit #255 of field element representation was not correctly cleared. (The decodeUCoordinate function from RFC7748 mandates clearing bit #255 for field element representations for use in the X25519 function.).

```
u0: 0000000000000000000000000000000000000000000000000000000000000000
u1: 0100000000000000000000000000000000000000000000000000000000000000
u2: ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u3: e0eb7a7c3b41b8ae1656e3faf19fc46ada098deb9c32b1fd866205165f49b800
u4: 5f9c95bca3508c24b1d0b1559c83ef5b04445cc4581c8e86d8224eddd09f1157
u5: edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u6: daffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u7: eeffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff7f
u8: dbffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
u9: d9ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
ua: cdeb7a7c3b41b8ae1656e3faf19fc46ada098deb9c32b1fd866205165f49b880
ub: 4c9c95bca3508c24b1d0b1559c83ef5b04445cc4581c8e86d8224eddd09f11d7
```

u0 ... ub MUST be verified to produce the correct results q0 ... qb:

Additionally, u0,u1,u2,u3,u4,u5 and u7 MUST trigger the abort case when included in MSGa or MSGb.

```
s = af46e36bf0527c9d3b16154b82465edd62144c0ac1fc5a18506a2244ba449aff
qN = G_X25519.scalar_mult_vfy(s, uX)
q0: 0000000000000000000000000000000000000000000000000000000000000000
q1: 0000000000000000000000000000000000000000000000000000000000000000
q2: 0000000000000000000000000000000000000000000000000000000000000000
q3: 0000000000000000000000000000000000000000000000000000000000000000
q4: 0000000000000000000000000000000000000000000000000000000000000000
q5: 0000000000000000000000000000000000000000000000000000000000000000
q6: d8e2c776bbacd510d09fd9278b7edcd25fc5ae9adfb3b6e040e8d3b71b21806
q7: 0000000000000000000000000000000000000000000000000000000000000000
q8: c85c655ebe8be44ba9c0ffde69f2fe10194458d137f09bbff725ce58803cdb38
q9: db64dafa9b8fddl36914e61461935fe92aa372cb056314e1231bc4ec12417456
qa: e062dcd5376d58297be2618c7498f55baa07d7e03184e8aada20bca28888bf7a
qb: 993c6ad11c4c29da9a56f7691fd0ff8d732e49de6250b6c2e80003fff4629a175
```

B.2. Test vector for CPace using group X448 and hash SHAKE-256

B.2.1. Test vectors for calculate_generator with group X448

Inputs

Adb = b'Adb'

yb (little endian): (length: 56 bytes)

848b0779ff415f0af4ea14df9dd1d3c29ac41d836c7808896c4eba19
c51ac40a439caf5e61ec88c307c7d619195229412eaa73fb2a5ea20d

Outputs

Yb: (length: 56 bytes)

53c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c59df9c6
0422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d58

MSGb: (length: 61 bytes)

3853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0ac00c59df9
c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d
5803414462

B.2.4. Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 56 bytes)

e00af217556a40ccbc9822cc27a43542e45166a653aa4df746d5f8e1
e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a659997

scalar_mult_vfy(yb,Ya): (length: 56 bytes)

e00af217556a40ccbc9822cc27a43542e45166a653aa4df746d5f8e1
e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a659997

B.2.5. Test vector for ISK calculation initiator/responder

unordered cat of transcript : (length: 122 bytes)

38396bd11daf223711e575cac6021e3fa31558012048a1cec7876292
b96c61eda353fe04f33028d2352779668a934084da776c1c51a58ce4
b5034144613853c519fb490fde5a04bda8c18b327d0fc1a9391d19e0
ac00c59df9c60422284e593d6b092eac94f5aa644ed883f39bd4f04e
4beb6af86d5803414462

DSI = G.DSI_ISK, b'CPace448_ISK': (length: 12 bytes)

43506163653434385f49534b

prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 209 bytes)

0c43506163653434385f49534b105223e0cdc45d6575668d64c55200
412438e00af217556a40ccbc9822cc27a43542e45166a653aa4df746
d5f8e1e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a
65999738396bd11daf223711e575cac6021e3fa31558012048a1cec7
876292b96c61eda353fe04f33028d2352779668a934084da776c1c51
a58ce4b5034144613853c519fb490fde5a04bda8c18b327d0fc1a939
1d19e0ac00c59df9c60422284e593d6b092eac94f5aa644ed883f39b
d4f04e4beb6af86d5803414462

ISK result: (length: 64 bytes)

4030297722c1914711da6b2a224a44b53b30c05ab02c2a3d3ccc7272
a3333ce3a4564c17031b634e89f65681f52d5c3d1df7baeb88523d2e
481b3858aed86315

B.2.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 122 bytes)
3853c519fb490fde5a04bda8c18b327d0fcla9391d19e0ac00c59df9
c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb6af86d
580341446238396bd11daf223711e575cac6021e3fa31558012048a1
cec7876292b96c61eda353fe04f33028d2352779668a934084da776c
1c51a58ce4b503414461
DSI = G.DSI_ISK, b'CPace448_ISK': (length: 12 bytes)
43506163653434385f49534b
prefix_free_cat (DSI, sid, K) || oCAT (MSGa, MSGb) :
(length: 209 bytes)
0c43506163653434385f49534b105223e0cdc45d6575668d64c55200
412438e00af217556a40ccbc9822cc27a43542e45166a653aa4df746
d5f8e1e8df483e9baff71c9eb03ee20a688ad4e4d359f70ac9ec3f6a
6599973853c519fb490fde5a04bda8c18b327d0fcla9391d19e0ac00
c59df9c60422284e593d6b092eac94f5aa644ed883f39bd4f04e4beb
6af86d580341446238396bd11daf223711e575cac6021e3fa3155801
2048a1cec7876292b96c61eda353fe04f33028d2352779668a934084
da776c1c51a58ce4b503414461
ISK result: (length: 64 bytes)
925e95d1095dadlaf6378d5ef8b9a998bd3855bfc7d36cb5ca05b0a7
a93346abcb8cef04bceb28c38fdaf0cc608fd1dcd462ab523f3b7f75
2c77c411be3ac8fb

```

B.2.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50, 0x61, 0x73, 0x73, 0x77, 0x6f, 0x72, 0x64,
};
const uint8_t tc_CI[] = {
    0x0a, 0x41, 0x69, 0x6e, 0x69, 0x74, 0x69, 0x61, 0x74, 0x6f, 0x72, 0x0a,
    0x42, 0x72, 0x65, 0x73, 0x70, 0x6f, 0x6e, 0x64, 0x65, 0x72,
};
const uint8_t tc_sid[] = {
    0x52, 0x23, 0xe0, 0xcd, 0xc4, 0x5d, 0x65, 0x75, 0x66, 0x8d, 0x64, 0xc5,
    0x52, 0x00, 0x41, 0x24,
};
const uint8_t tc_g[] = {
    0x6f, 0xda, 0xe1, 0x47, 0x18, 0xeb, 0x75, 0x06, 0xdd, 0x96, 0xe3, 0xf7,
    0x79, 0x78, 0x96, 0xef, 0xdb, 0x8d, 0xb9, 0xec, 0x07, 0x97, 0x48, 0x5c,
    0x9c, 0x48, 0xa1, 0x92, 0x2e, 0x44, 0x96, 0x1d, 0xa0, 0x97, 0xf2, 0x90,
    0x8b, 0x08, 0x4a, 0x5d, 0xe3, 0x3a, 0xb6, 0x71, 0x63, 0x06, 0x60, 0xd2,
    0x7d, 0x79, 0xff, 0xd6, 0xee, 0x8e, 0xc8, 0x46,
};
const uint8_t tc_ysa[] = {
    0x21, 0xb4, 0xf4, 0xbd, 0x9e, 0x64, 0xed, 0x35, 0x5c, 0x3e, 0xb6, 0x76,
    0xa2, 0x8e, 0xbe, 0xda, 0xf6, 0xd8, 0xf1, 0x7b, 0xdc, 0x36, 0x59, 0x95,
};

```



```
    0xb3,0x19,0x09,0x71,0x53,0x04,0x40,0x80,0x51,0x6b,0xd0,0x83,
    0xbf,0xcc,0xe6,0x61,0x21,0xa3,0x07,0x26,0x46,0x99,0x4c,0x84,
    0x30,0xcc,0x38,0x2b,0x8d,0xc5,0x43,0xe8,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x39,0x6b,0xd1,0x1d,0xaf,0x22,0x37,0x11,0xe5,0x75,0xca,0xc6,
    0x02,0x1e,0x3f,0xa3,0x15,0x58,0x01,0x20,0x48,0xa1,0xce,0xc7,
    0x87,0x62,0x92,0xb9,0x6c,0x61,0xed,0xa3,0x53,0xfe,0x04,0xf3,
    0x30,0x28,0xd2,0x35,0x27,0x79,0x66,0x8a,0x93,0x40,0x84,0xda,
    0x77,0x6c,0x1c,0x51,0xa5,0x8c,0xe4,0xb5,
};
const uint8_t tc_yb[] = {
    0x84,0x8b,0x07,0x79,0xff,0x41,0x5f,0x0a,0xf4,0xea,0x14,0xdf,
    0x9d,0xd1,0xd3,0xc2,0x9a,0xc4,0x1d,0x83,0x6c,0x78,0x08,0x89,
    0x6c,0x4e,0xba,0x19,0xc5,0x1a,0xc4,0x0a,0x43,0x9c,0xaf,0x5e,
    0x61,0xec,0x88,0xc3,0x07,0xc7,0xd6,0x19,0x19,0x52,0x29,0x41,
    0x2e,0xaa,0x73,0xfb,0x2a,0x5e,0xa2,0x0d,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0x53,0xc5,0x19,0xfb,0x49,0x0f,0xde,0x5a,0x04,0xbd,0xa8,0xc1,
    0x8b,0x32,0x7d,0x0f,0xc1,0xa9,0x39,0x1d,0x19,0xe0,0xac,0x00,
    0xc5,0x9d,0xf9,0xc6,0x04,0x22,0x28,0x4e,0x59,0x3d,0x6b,0x09,
    0x2e,0xac,0x94,0xf5,0xaa,0x64,0x4e,0xd8,0x83,0xf3,0x9b,0xd4,
    0xf0,0x4e,0x4b,0xeb,0x6a,0xf8,0x6d,0x58,
};
const uint8_t tc_K[] = {
    0xe0,0x0a,0xf2,0x17,0x55,0x6a,0x40,0xcc,0xbc,0x98,0x22,0xcc,
    0x27,0xa4,0x35,0x42,0xe4,0x51,0x66,0xa6,0x53,0xaa,0x4d,0xf7,
    0x46,0xd5,0xf8,0xe1,0xe8,0xdf,0x48,0x3e,0x9b,0xaf,0xf7,0x1c,
    0x9e,0xb0,0x3e,0xe2,0x0a,0x68,0x8a,0xd4,0xe4,0xd3,0x59,0xf7,
    0x0a,0xc9,0xec,0x3f,0x6a,0x65,0x99,0x97,
};
const uint8_t tc_ISK_IR[] = {
    0x40,0x30,0x29,0x77,0x22,0xc1,0x91,0x47,0x11,0xda,0x6b,0x2a,
    0x22,0x4a,0x44,0xb5,0x3b,0x30,0xc0,0x5a,0xb0,0x2c,0x2a,0x3d,
    0x3c,0xcc,0x72,0x72,0xa3,0x33,0x3c,0xe3,0xa4,0x56,0x4c,0x17,
    0x03,0x1b,0x63,0x4e,0x89,0xf6,0x56,0x81,0xf5,0x2d,0x5c,0x3d,
    0x1d,0xf7,0xba,0xeb,0x88,0x52,0x3d,0x2e,0x48,0x1b,0x38,0x58,
    0xae,0xd8,0x63,0x15,
};
const uint8_t tc_ISK_SY[] = {
    0x92,0x5e,0x95,0xd1,0x09,0x5d,0xad,0x1a,0xf6,0x37,0x8d,0x5e,
```

```

0xf8,0xb9,0xa9,0x98,0xbd,0x38,0x55,0xbf,0xc7,0xd3,0x6c,0xb5,
0xca,0x05,0xb0,0xa7,0xa9,0x33,0x46,0xab,0xcb,0x8c,0xef,0x04,
0xbc,0xeb,0x28,0xc3,0x8f,0xda,0xf0,0xcc,0x60,0x8f,0xd1,0xdc,
0xd4,0x62,0xab,0x52,0x3f,0x3b,0x7f,0x75,0x2c,0x77,0xc4,0x11,
0xbe,0x3a,0xc8,0xfb,
};

```

B.2.8. Test vectors for G_X448.scalar_mult_vfy: low order points

Test vectors for which G_X448.scalar_mult_vfy(s_in,ux) must return the neutral element. This includes points that are non-canonically encoded, i.e. have coordinate values larger than the field prime.

Weak points for X448 smaller than the field prime (canonical)

```

u0: (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
u1: (length: 56 bytes)
0100000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
u2: (length: 56 bytes)
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe

```

Weak points for X448 larger or equal to the field prime (non-canonical)

```

u3: (length: 56 bytes)
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe
u4: (length: 56 bytes)
00000000000000000000000000000000000000000000000000000000000000ff
fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffe

```

All of the above points u0 ... u4 MUST trigger the abort case when included in the protocol messages MSGa or MSGb.

Expected results for X448 resp. G_X448.scalar_mult_vfy

```
scalar s: (length: 56 bytes)
af8a14218bf2a2062926d2ea9b8fe4e8b6817349b6ed2feble5d64d7a4
523f15fceed70fb111e870dc58d191e66a14d3e9d482d04432cadd
G_X448.scalar_mult_vfy(s,u0): (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u1): (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u2): (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u3): (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
G_X448.scalar_mult_vfy(s,u4): (length: 56 bytes)
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Test vectors for scalar_mult with nonzero outputs

```
scalar s: (length: 56 bytes)
af8a14218bf2a2062926d2ea9b8fe4e8b6817349b6ed2feble5d64d7a4
523f15fceed70fb111e870dc58d191e66a14d3e9d482d04432cadd
point coordinate u_curve on the curve: (length: 56 bytes)
ab0c68d772ec2eb9de25c49700e46d6325e66d6aa39d7b65eb84a68c55
69d47bd71b41f3e0d210f44e146dec8926b174acb3f940a0b82cab
G_X448.scalar_mult_vfy(s,u_curve): (length: 56 bytes)
3b0fa9bc40a6fdc78c9e06ff7a54c143c5d52f365607053bf0656f5142
0496295f910a101b38edc1acd3bd240fd55dcb7a360553b8a7627e

point coordinate u_twist on the twist: (length: 56 bytes)
c981cd1elf72d9c35c7d7cf6be426757c0dc8206a2fcfa564a8e7618c0
3c0e61f9a2eb1c3e0dd97d6e9b1010f5edd03397a83f5a914cb3ff
G_X448.scalar_mult_vfy(s,u_twist): (length: 56 bytes)
d0a2bb7e9c5c2c627793d8342f23b759fe7d9e3320a85ca4fd61376331
50ffd9a9148a9b75c349fac43d64bec49a6e126cc92cbfbf353961
```

B.3. Test vector for CPace using group ristretto255 and hash SHA-512

B.3.1. Test vectors for calculate_generator with group ristretto255

Inputs

```

H   = SHA-512 with input block size 128 bytes.
PRS = b'Password' ; ZPAD length: 100 ;
DSI = b'CPaceRistretto255'
CI  = b'\nAinitiator\nBresponder'
CI  = 0a41696e69746961746f720a42726573706f6e646572
sid = 7e4b4791d6a8ef019b936c79fb7f2c57

```

Outputs

```

generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
(length: 168 bytes)
11435061636552697374726574746f3235350850617373776f726464
00000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000
0a42726573706f6e646572107e4b4791d6a8ef019b936c79fb7f2c57
hash result: (length: 64 bytes)
a5ce446f63a1ae6dlfee80fa67d0b4004a4b1283ec5549a462bf33a6
c1ae06a0871f9bf48545f49b2a792eed255ac04f52758c9c60448306
810b44e986e3dcbb
encoded generator g: (length: 32 bytes)
9c5712178570957204d89ac11achef789dd076992ba361429acb2bc3
8c71d14c

```

B.3.2. Test vector for MSGa

Inputs

```

ADa = b'ADa'
ya (little endian): (length: 32 bytes)
1433ddl9359992d4e06d740d3993d429af6338ffb4531ce175d22449
853a790b

```

Outputs

```

Ya: (length: 32 bytes)
a8fc42c4d57b3c7346661011122a00563d0995fd72b62123ae244400
e86d7b1a
MSGa: (length: 37 bytes)
20a8fc42c4d57b3c7346661011122a00563d0995fd72b62123ae2444
00e86d7b1a03414461

```

B.3.3. Test vector for MSGb

Inputs

```
ADb = b'ADb'
yb (little endian): (length: 32 bytes)
0e6566d32d80a5a1135f99c27f2d637aa24da23027c3fa76b9d1cfd9
742fdc00
```

Outputs

```
Yb: (length: 32 bytes)
fc8e84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd2662
36676d63
MSGb: (length: 37 bytes)
20fc8e84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd26
6236676d6303414462
```

B.3.4. Test vector for secret points K

```
scalar_mult_vfy(ya,Yb): (length: 32 bytes)
3efef1706f42efa354020b087b37fbd9f81cf72a16f4947e4a042a7f
1aaa2b6f
scalar_mult_vfy(yb,Ya): (length: 32 bytes)
3efef1706f42efa354020b087b37fbd9f81cf72a16f4947e4a042a7f
1aaa2b6f
```

B.3.5. Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 74 bytes)
20a8fc42c4d57b3c7346661011122a00563d0995fd72b62123ae2444
00e86d7b1a0341446120fc8e84ae4ab725909af05a56ef9714db6930
e4a5589b3fee6cdd266236676d6303414462
DSI = G.DSI_ISK, b'CPaceRistretto255_ISK':
(length: 21 bytes)
435061636552697374726574746f3235355f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 146 bytes)
15435061636552697374726574746f3235355f49534b107e4b4791d6
a8ef019b936c79fb7f2c57203efef1706f42efa354020b087b37fbd9
f81cf72a16f4947e4a042a7f1aaa2b6f20a8fc42c4d57b3c73466610
11122a00563d0995fd72b62123ae244400e86d7b1a0341446120fc8e
84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd26623667
6d6303414462
ISK result: (length: 64 bytes)
0e33c5822bd495dea94ba7af161501f1b2d6a16d464b5d6e1a53dcbf
b9244b9ba66c09c430fffdfe4fb4e99b4ea46f991a272de0431c132c
2c79fd6dela7e5e4
```

B.3.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 74 bytes)
20fc8e84ae4ab725909af05a56ef9714db6930e4a5589b3fee6cdd26
6236676d630341446220a8fc42c4d57b3c7346661011122a00563d09
95fd72b62123ae244400e86d7b1a03414461
DSI = G.DSI_ISK, b'CPaceRistretto255_ISK':
(length: 21 bytes)
435061636552697374726574746f3235355f49534b
prefix_free_cat(DSI,sid,K) || oCAT(MSGa,MSGb):
(length: 146 bytes)
15435061636552697374726574746f3235355f49534b107e4b4791d6
a8ef019b936c79fb7f2c57203efef1706f42efa354020b087b37fbd9
f81cf72a16f4947e4a042a7f1aaa2b6f20fc8e84ae4ab725909af05a
56ef9714db6930e4a5589b3fee6cdd266236676d630341446220a8fc
42c4d57b3c7346661011122a00563d0995fd72b62123ae244400e86d
7b1a03414461
ISK result: (length: 64 bytes)
ca36335be682a480a9fc63977d044a10ff7adfcda0f2978fbcf8713d
2a4e23e25c05a9a02edcfbfff2ede65b752f8ealf4454d764ad8ed860
7c158ef662614567

```

B.3.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t tc_CI[] = {
    0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
    0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const uint8_t tc_sid[] = {
    0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,
    0xfb,0x7f,0x2c,0x57,
};
const uint8_t tc_g[] = {
    0x9c,0x57,0x12,0x17,0x85,0x70,0x95,0x72,0x04,0xd8,0x9a,0xc1,
    0x1a,0xcb,0xef,0x78,0x9d,0xd0,0x76,0x99,0x2b,0xa3,0x61,0x42,
    0x9a,0xcb,0x2b,0xc3,0x8c,0x71,0xd1,0x4c,
};
const uint8_t tc_ya[] = {
    0x14,0x33,0xdd,0x19,0x35,0x99,0x92,0xd4,0xe0,0x6d,0x74,0x0d,
    0x39,0x93,0xd4,0x29,0xaf,0x63,0x38,0xff,0xb4,0x53,0x1c,0xe1,
    0x75,0xd2,0x24,0x49,0x85,0x3a,0x79,0x0b,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0xa8,0xfc,0x42,0xc4,0xd5,0x7b,0x3c,0x73,0x46,0x66,0x10,0x11,

```

```

    0x12,0x2a,0x00,0x56,0x3d,0x09,0x95,0xfd,0x72,0xb6,0x21,0x23,
    0xae,0x24,0x44,0x00,0xe8,0x6d,0x7b,0x1a,
};
const uint8_t tc_yb[] = {
    0x0e,0x65,0x66,0xd3,0x2d,0x80,0xa5,0xa1,0x13,0x5f,0x99,0xc2,
    0x7f,0x2d,0x63,0x7a,0xa2,0x4d,0xa2,0x30,0x27,0xc3,0xfa,0x76,
    0xb9,0xd1,0xcf,0xd9,0x74,0x2f,0xdc,0x00,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0xfc,0x8e,0x84,0xae,0x4a,0xb7,0x25,0x90,0x9a,0xf0,0x5a,0x56,
    0xef,0x97,0x14,0xdb,0x69,0x30,0xe4,0xa5,0x58,0x9b,0x3f,0xee,
    0x6c,0xdd,0x26,0x62,0x36,0x67,0x6d,0x63,
};
const uint8_t tc_K[] = {
    0x3e,0xfe,0xf1,0x70,0x6f,0x42,0xef,0xa3,0x54,0x02,0x0b,0x08,
    0x7b,0x37,0xfb,0xd9,0xf8,0x1c,0xf7,0x2a,0x16,0xf4,0x94,0x7e,
    0x4a,0x04,0x2a,0x7f,0x1a,0xaa,0x2b,0x6f,
};
const uint8_t tc_ISK_IR[] = {
    0x0e,0x33,0xc5,0x82,0x2b,0xd4,0x95,0xde,0xa9,0x4b,0xa7,0xaf,
    0x16,0x15,0x01,0xf1,0xb2,0xd6,0xa1,0x6d,0x46,0x4b,0x5d,0x6e,
    0x1a,0x53,0xdc,0xbf,0xb9,0x24,0x4b,0x9b,0xa6,0x6c,0x09,0xc4,
    0x30,0xff,0xfd,0xfe,0x4f,0xb4,0xe9,0x9b,0x4e,0xa4,0x6f,0x99,
    0x1a,0x27,0x2d,0xe0,0x43,0x1c,0x13,0x2c,0x2c,0x79,0xfd,0x6d,
    0xe1,0xa7,0xe5,0xe4,
};
const uint8_t tc_ISK_SY[] = {
    0xca,0x36,0x33,0x5b,0xe6,0x82,0xa4,0x80,0xa9,0xfc,0x63,0x97,
    0x7d,0x04,0x4a,0x10,0xff,0x7a,0xdf,0xcd,0xa0,0xf2,0x97,0x8f,
    0xbc,0xf8,0x71,0x3d,0x2a,0x4e,0x23,0xe2,0x5c,0x05,0xa9,0xa0,
    0x2e,0xdc,0xfb,0xff,0x2e,0xde,0x65,0xb7,0x52,0xf8,0xea,0x1f,
    0x44,0x54,0xd7,0x64,0xad,0x8e,0xd8,0x60,0x7c,0x15,0x8e,0xf6,
    0x62,0x61,0x45,0x67,
};

```

B.3.8. Test case for scalar_mult with valid inputs

```

s: (length: 32 bytes)
  7cd0e075fa7955ba52c02759a6c90dbbfc10e6d40aea8d283e407d88
  cf538a05
X: (length: 32 bytes)
  021ca069484e890c9e494d8ed6bb0f66cbd9a8f0ef67168f36c51e0e
  feb8f347
G.scalar_mult(s,decode(X)): (length: 32 bytes)
  62aaa018755dc881902097c2a993c0b7c0a4fe33bce2c0182b46a44c
  40b95119
G.scalar_mult_vfy(s,X): (length: 32 bytes)
  62aaa018755dc881902097c2a993c0b7c0a4fe33bce2c0182b46a44c
  40b95119

```

B.3.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When points Y_i1 or Y_i2 are included in MSGa or MSGb the protocol MUST abort.

```

s: (length: 32 bytes)
  7cd0e075fa7955ba52c02759a6c90dbbfc10e6d40aea8d283e407d88
  cf538a05
Y_i1: (length: 32 bytes)
  011ca069484e890c9e494d8ed6bb0f66cbd9a8f0ef67168f36c51e0e
  feb8f347
Y_i2 == G.I: (length: 32 bytes)
  0000000000000000000000000000000000000000000000000000000000000000
  00000000
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I

```

B.4. Test vector for CPace using group decaf448 and hash SHAKE-256

B.4.1. Test vectors for calculate_generator with group decaf448

Inputs

```

H = SHAKE-256 with input block size 136 bytes.
PRS = b'Password' ; ZPAD length: 112 ;
DSI = b'CPaceDecaf448'
CI = b'\nAinitiator\nBresponder'
CI = 0a41696e69746961746f720a42726573706f6e646572
sid = 5223e0cdc45d6575668d64c552004124

```

Outputs

```

generator_string(G.DSI,PRS,CI,sid,H.s_in_bytes):
(length: 176 bytes)
0d435061636544656361663434380850617373776f72647000000000
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000160a4169
6e69746961746f720a42726573706f6e646572105223e0cdc45d6575
668d64c552004124
hash result: (length: 112 bytes)
8955b426ff1d3a22032d21c013cf94134cee9a4235e93261a4911edb
f68f2945f0267c983954262c7f59badb9caf468ebe21b7e9885657af
b8f1a3b783c2047ba519e113ecf81b2b580dd481f499beabd401cc77
1d28915fb750011209040f5f03b2ceb5e5eb259c96b478382d5a5c57
encoded generator g: (length: 56 bytes)
c811b3f6b0d27b58a74d8274bf5f9ca6b7ada15b0bf57b79a6b45c13
2eb0c28bdcc3abf4e5932cea97a80997ead1c146b98b1a1f1def30f3

```

B.4.2. Test vector for MSGa

Inputs

```

ADa = b'ADa'
ya (little endian): (length: 56 bytes)
d8d2e26c821a12d7f59a8dee023d3f6155976152e16c73cbf68c303d
f0404399f0a7b614a65df50a9788f00b410586b443f738ad7ff03930

```

Outputs

```

Ya: (length: 56 bytes)
223f95a5430a2f2a499431696d23ea2d0a90f432e5491e45e4005f3d
d785e7be1235b79252670099bc993c2df5c261dfb7a8989f091e2be3
MSGa: (length: 61 bytes)
38223f95a5430a2f2a499431696d23ea2d0a90f432e5491e45e4005f
3dd785e7be1235b79252670099bc993c2df5c261dfb7a8989f091e2b
e303414461

```

B.4.3. Test vector for MSGb

Inputs

Adb = b'Adb'

yb (little endian): (length: 56 bytes)

91bae9793f4a8aceb1b5c54375a7ed1858a79a6e72dab959c8bdf3a7
5ac9bb4de2a25af4d4a9a5c5bc5441d19b8e3f6fcce7196c6afc2236

Outputs

Yb: (length: 56 bytes)

b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc361e979c9e0d3a0967
e630094ba3d1555821ac1f979996ef5ce79f012ffe279ac89b287bee

MSGb: (length: 61 bytes)

38b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc361e979c9e0d3a09
67e630094ba3d1555821ac1f979996ef5ce79f012ffe279ac89b287b
ee03414462

B.4.4. Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 56 bytes)

dc504938fb70eb13916697aa3e076e82537c171aa326121399c896fe
ea0e198b41b6bae300bb86f8c61d4b170eee4717b5497016f34364a9

scalar_mult_vfy(yb,Ya): (length: 56 bytes)

dc504938fb70eb13916697aa3e076e82537c171aa326121399c896fe
ea0e198b41b6bae300bb86f8c61d4b170eee4717b5497016f34364a9

B.4.5. Test vector for ISK calculation initiator/responder

unordered cat of transcript : (length: 122 bytes)

38223f95a5430a2f2a499431696d23ea2d0a90f432e5491e45e4005f
3dd785e7be1235b79252670099bc993c2df5c261dfb7a8989f091e2b
e30341446138b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc361e97
9c9e0d3a0967e630094ba3d1555821ac1f979996ef5ce79f012ffe27
9ac89b287bee03414462

DSI = G.DSI_ISK, b'CPaceDecaf448_ISK': (length: 17 bytes)

435061636544656361663434385f49534b

prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 214 bytes)

11435061636544656361663434385f49534b105223e0cdc45d657566
8d64c55200412438dc504938fb70eb13916697aa3e076e82537c171a
a326121399c896feea0e198b41b6bae300bb86f8c61d4b170eee4717
b5497016f34364a938223f95a5430a2f2a499431696d23ea2d0a90f4
32e5491e45e4005f3dd785e7be1235b79252670099bc993c2df5c261
dfb7a8989f091e2be30341446138b6ba0a336c103c6c92019ae4cfbc
b88d8f6bfc361e979c9e0d3a0967e630094ba3d1555821ac1f979996
ef5ce79f012ffe279ac89b287bee03414462

ISK result: (length: 64 bytes)

ebe28369491f8899a5af3b339d4993881b69d22607c58719da6eaab3
8f0d9025eae413ca2b072b156ce4a0d4778ff471a63c4d908cab70bc
2081951d504cbb03

B.4.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 122 bytes)
38b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc361e979c9e0d3a09
67e630094ba3d1555821ac1f979996ef5ce79f012ffe279ac89b287b
ee0341446238223f95a5430a2f2a499431696d23ea2d0a90f432e549
1e45e4005f3dd785e7be1235b79252670099bc993c2df5c261dfb7a8
989f091e2be303414461
DSI = G.DSI_ISK, b'CPaceDecaf448_ISK': (length: 17 bytes)
435061636544656361663434385f49534b
prefix_free_cat (DSI, sid, K) || oCAT (MSGa, MSGb) :
(length: 214 bytes)
11435061636544656361663434385f49534b105223e0cdc45d657566
8d64c55200412438dc504938fb70eb13916697aa3e076e82537c171a
a326121399c896feea0e198b41b6bae300bb86f8c61d4b170eee4717
b5497016f34364a938b6ba0a336c103c6c92019ae4cfbcb88d8f6bfc
361e979c9e0d3a0967e630094ba3d1555821ac1f979996ef5ce79f01
2ffe279ac89b287bee0341446238223f95a5430a2f2a499431696d23
ea2d0a90f432e5491e45e4005f3dd785e7be1235b79252670099bc99
3c2df5c261dfb7a8989f091e2be303414461
ISK result: (length: 64 bytes)
2996d1953320581b587f473cfd5c974c5a8597b22b37fefe49bdb7b8
4073424f7f7a6e456498665a69530741398c6010bdb346f79944acc9
0c5c537fa35cd29a

```

B.4.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50, 0x61, 0x73, 0x73, 0x77, 0x6f, 0x72, 0x64,
};
const uint8_t tc_CI[] = {
    0x0a, 0x41, 0x69, 0x6e, 0x69, 0x74, 0x69, 0x61, 0x74, 0x6f, 0x72, 0x0a,
    0x42, 0x72, 0x65, 0x73, 0x70, 0x6f, 0x6e, 0x64, 0x65, 0x72,
};
const uint8_t tc_sid[] = {
    0x52, 0x23, 0xe0, 0xcd, 0xc4, 0x5d, 0x65, 0x75, 0x66, 0x8d, 0x64, 0xc5,
    0x52, 0x00, 0x41, 0x24,
};
const uint8_t tc_g[] = {
    0xc8, 0x11, 0xb3, 0xf6, 0xb0, 0xd2, 0x7b, 0x58, 0xa7, 0x4d, 0x82, 0x74,
    0xbf, 0x5f, 0x9c, 0xa6, 0xb7, 0xad, 0xa1, 0x5b, 0x0b, 0xf5, 0x7b, 0x79,
    0xa6, 0xb4, 0x5c, 0x13, 0x2e, 0xb0, 0xc2, 0x8b, 0xdc, 0xc3, 0xab, 0xf4,
    0xe5, 0x93, 0x2c, 0xea, 0x97, 0xa8, 0x09, 0x97, 0xea, 0xd1, 0xc1, 0x46,
    0xb9, 0x8b, 0x1a, 0x1f, 0x1d, 0xef, 0x30, 0xf3,
};
const uint8_t tc_ya[] = {
    0xd8, 0xd2, 0xe2, 0x6c, 0x82, 0x1a, 0x12, 0xd7, 0xf5, 0x9a, 0x8d, 0xee,
    0x02, 0x3d, 0x3f, 0x61, 0x55, 0x97, 0x61, 0x52, 0xe1, 0x6c, 0x73, 0xcb,
};

```

```
    0xf6,0x8c,0x30,0x3d,0xf0,0x40,0x43,0x99,0xf0,0xa7,0xb6,0x14,
    0xa6,0x5d,0xf5,0x0a,0x97,0x88,0xf0,0x0b,0x41,0x05,0x86,0xb4,
    0x43,0xf7,0x38,0xad,0x7f,0xf0,0x39,0x30,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x22,0x3f,0x95,0xa5,0x43,0x0a,0x2f,0x2a,0x49,0x94,0x31,0x69,
    0x6d,0x23,0xea,0x2d,0x0a,0x90,0xf4,0x32,0xe5,0x49,0x1e,0x45,
    0xe4,0x00,0x5f,0x3d,0xd7,0x85,0xe7,0xbe,0x12,0x35,0xb7,0x92,
    0x52,0x67,0x00,0x99,0xbc,0x99,0x3c,0x2d,0xf5,0xc2,0x61,0xdf,
    0xb7,0xa8,0x98,0x9f,0x09,0x1e,0x2b,0xe3,
};
const uint8_t tc_yb[] = {
    0x91,0xba,0xe9,0x79,0x3f,0x4a,0x8a,0xce,0xb1,0xb5,0xc5,0x43,
    0x75,0xa7,0xed,0x18,0x58,0xa7,0x9a,0x6e,0x72,0xda,0xb9,0x59,
    0xc8,0xbd,0xf3,0xa7,0x5a,0xc9,0xbb,0x4d,0xe2,0xa2,0x5a,0xf4,
    0xd4,0xa9,0xa5,0xc5,0xbc,0x54,0x41,0xd1,0x9b,0x8e,0x3f,0x6f,
    0xcc,0xe7,0x19,0x6c,0x6a,0xfc,0x22,0x36,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0xb6,0xba,0x0a,0x33,0x6c,0x10,0x3c,0x6c,0x92,0x01,0x9a,0xe4,
    0xcf,0xbc,0xb8,0x8d,0x8f,0x6b,0xfc,0x36,0x1e,0x97,0x9c,0x9e,
    0x0d,0x3a,0x09,0x67,0xe6,0x30,0x09,0x4b,0xa3,0xd1,0x55,0x58,
    0x21,0xac,0x1f,0x97,0x99,0x96,0xef,0x5c,0xe7,0x9f,0x01,0x2f,
    0xfe,0x27,0x9a,0xc8,0x9b,0x28,0x7b,0xee,
};
const uint8_t tc_K[] = {
    0xdc,0x50,0x49,0x38,0xfb,0x70,0xeb,0x13,0x91,0x66,0x97,0xaa,
    0x3e,0x07,0x6e,0x82,0x53,0x7c,0x17,0x1a,0xa3,0x26,0x12,0x13,
    0x99,0xc8,0x96,0xfe,0xea,0x0e,0x19,0x8b,0x41,0xb6,0xba,0xe3,
    0x00,0xbb,0x86,0xf8,0xc6,0x1d,0x4b,0x17,0x0e,0xee,0x47,0x17,
    0xb5,0x49,0x70,0x16,0xf3,0x43,0x64,0xa9,
};
const uint8_t tc_ISK_IR[] = {
    0xeb,0xe2,0x83,0x69,0x49,0x1f,0x88,0x99,0xa5,0xaf,0x3b,0x33,
    0x9d,0x49,0x93,0x88,0x1b,0x69,0xd2,0x26,0x07,0xc5,0x87,0x19,
    0xda,0x6e,0xaa,0xb3,0x8f,0x0d,0x90,0x25,0xea,0xe4,0x13,0xca,
    0x2b,0x07,0x2b,0x15,0x6c,0xe4,0xa0,0xd4,0x77,0x8f,0xf4,0x71,
    0xa6,0x3c,0x4d,0x90,0x8c,0xab,0x70,0xbc,0x20,0x81,0x95,0x1d,
    0x50,0x4c,0xbb,0x03,
};
const uint8_t tc_ISK_SY[] = {
    0x29,0x96,0xd1,0x95,0x33,0x20,0x58,0x1b,0x58,0x7f,0x47,0x3c,
```

```

0xfd, 0x5c, 0x97, 0x4c, 0x5a, 0x85, 0x97, 0xb2, 0x2b, 0x37, 0xfe, 0xfe,
0x49, 0xbd, 0xb7, 0xb8, 0x40, 0x73, 0x42, 0x4f, 0x7f, 0x7a, 0x6e, 0x45,
0x64, 0x98, 0x66, 0x5a, 0x69, 0x53, 0x07, 0x41, 0x39, 0x8c, 0x60, 0x10,
0xbd, 0xb3, 0x46, 0xf7, 0x99, 0x44, 0xac, 0xc9, 0x0c, 0x5c, 0x53, 0x7f,
0xa3, 0x5c, 0xd2, 0x9a,
};

```

B.4.8. Test case for scalar_mult with valid inputs

```

s: (length: 56 bytes)
  ddlbc7015daabb7672129cc35a3ba815486b139deff9bdeca7a4fc61
  34323d34658761e90ff079972a7ca8aa5606498f4f4f0ebc0933a819
X: (length: 56 bytes)
  c803a6c8171ac38b66c5306553f45a487a24eb8581414444715bd2e5
  cf4c749a3b56a550f3c9a6ea3efa6e11ae6a6da12b98ef2f51174b9a
G.scalar_mult(s, decode(X)): (length: 56 bytes)
  b831a1f804fd3c902ae82f731d298aebf9152ea855f5b5da5ee88584
  84c55a7f65cc3ccf5f678496dc4cb1c8d6bc7ed17d2fe535fdc8f60e
G.scalar_mult_vfy(s, X): (length: 56 bytes)
  b831a1f804fd3c902ae82f731d298aebf9152ea855f5b5da5ee88584
  84c55a7f65cc3ccf5f678496dc4cb1c8d6bc7ed17d2fe535fdc8f60e

```

B.4.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When points Y_i1 or Y_i2 are included in MSGa or MSGb the protocol MUST abort.

```

s: (length: 56 bytes)
  ddlbc7015daabb7672129cc35a3ba815486b139deff9bdeca7a4fc61
  34323d34658761e90ff079972a7ca8aa5606498f4f4f0ebc0933a819
Y_i1: (length: 56 bytes)
  c703a6c8171ac38b66c5306553f45a487a24eb8581414444715bd2e5
  cf4c749a3b56a550f3c9a6ea3efa6e11ae6a6da12b98ef2f51174b9a
Y_i2 == G.I: (length: 56 bytes)
  0000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000000000000000000000000000000000000000000000
G.scalar_mult_vfy(s, Y_i1) = G.scalar_mult_vfy(s, Y_i2) = G.I

```

B.5. Test vector for CPace using group NIST P-256 and hash SHA-256

B.5.1. Test vectors for calculate_generator with group NIST P-256

Inputs

H = SHA-256 with input block size 64 bytes.
 PRS = b'Password' ; ZPAD length: 23 ;
 DSI = b'CPaceP256_XMD:SHA-256_SSWU_NU_'
 CI = b'\nAinitiator\nBresponder'
 CI = 0a41696e69746961746f720a42726573706f6e646572
 sid = 34b36454cab2e7842c389f7d88ecb7df

Outputs

generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
 (length: 104 bytes)
 1e4350616365503235365f584d443a5348412d3235365f535357555f
 4e555f0850617373776f72641700000000000000000000000000
 00000000000000000000160a41696e69746961746f720a42726573706f6e
 6465721034b36454cab2e7842c389f7d88ecb7df
 generator g: (length: 65 bytes)
 04993b46e30ba9cfc3dc2d3ae2cf9733cf03994e74383c4e1b4a92e8
 d6d466b321c4a642979162fbde9e1c9a6180bd27a0594491e4c231f5
 1006d0bf7992d07127

B.5.2. Test vector for MSGa

Inputs

ADa = b'ADa'
 ya (big endian): (length: 32 bytes)
 c9e47ca5debd2285727af47e55f5b7763fa79719da428f800190cc66
 59b4eafb

Outputs

Ya: (length: 65 bytes)
 0478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d051c
 a94593df5946314120faa87165cba131c1da3aac429dc3d99a9bac7d
 4c4cbb8570b4d5ea10
 Alternative correct value for Ya: $g^{(-ya)}$:
 (length: 65 bytes)
 0478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d051c
 a94593df59b9cebede05578e9b345ece3e25c553bd623c2666645382
 b3b3447a8f4b2a15ef
 MSGa: (length: 70 bytes)
 410478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d05
 1ca94593df5946314120faa87165cba131c1da3aac429dc3d99a9bac
 7d4c4cbb8570b4d5ea1003414461

B.5.3. Test vector for MSGb

Inputs

ADb = b'ADb'

yb (big endian): (length: 32 bytes)

a0b768ba7555621d133012d1dee27a0013c1bcfddd675811df12771e
44d77b10

Outputs

Yb: (length: 65 bytes)

04df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f65
4247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd558e6
fa57bf1f801aae7d3a

Alternative correct value for Yb: $g^{(-yb)}$:

(length: 65 bytes)

04df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f65
424795995140536f7b6035de75071d2eda7148282608cc01b42aa719
05a840e07fe55182c5

MSGb: (length: 70 bytes)

4104df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f
654247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd558
e6fa57bf1f801aae7d3a03414462

B.5.4. Test vector for secret points K

scalar_mult_vfy(ya, Yb): (length: 32 bytes)

27f7059d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23
fbab1037

scalar_mult_vfy(yb, Ya): (length: 32 bytes)

27f7059d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23
fbab1037

B.5.5. Test vector for ISK calculation initiator/responder

```

unordered cat of transcript : (length: 140 bytes)
410478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d05
1ca94593df5946314120faa87165cba131c1da3aac429dc3d99a9bac
7d4c4cbb8570b4d5ea10034144614104df13ffa89b0ce3cc553b1495
ff027886564d94b8d9165cd50e5f654247959951bfac90839fca218b
f8e2d1258eb7d7d9f733fe4cd558e6fa57bf1f801aae7d3a03414462
DSI = G.DSI_ISK, b'CPaceP256_XMD:SHA-256_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503235365f584d443a5348412d3235365f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K)||MSGa||MSGb: (length: 225 bytes)
224350616365503235365f584d443a5348412d3235365f535357555f
4e555f5f49534b1034b36454cab2e7842c389f7d88ecb7df2027f705
9d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23fbab10
37410478ac925a6e3447a537627a2163be005a422f55c08385c1ef7d
051ca94593df5946314120faa87165cba131c1da3aac429dc3d99a9b
ac7d4c4cbb8570b4d5ea10034144614104df13ffa89b0ce3cc553b14
95ff027886564d94b8d9165cd50e5f654247959951bfac90839fca21
8bf8e2d1258eb7d7d9f733fe4cd558e6fa57bf1f801aae7d3a034144
62
ISK result: (length: 32 bytes)
ddc1b133c387ecf344c0b496bc1223656cd6e7d99a5def8b3b026796
50811fc9

```

B.5.6. Test vector for ISK calculation parallel execution


```

ordered cat of transcript : (length: 140 bytes)
4104df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e5f
654247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd558
e6fa57bf1f801aae7d3a03414462410478ac925a6e3447a537627a21
63be005a422f55c08385clef7d051ca94593df5946314120faa87165
cba131c1da3aac429dc3d99a9bac7d4c4cbb8570b4d5ea1003414461
DSI = G.DSI_ISK, b'CPaceP256_XMD:SHA-256_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503235365f584d443a5348412d3235365f535357555f4e
555f5f49534b
prefix_free_cat (DSI, sid, K) || oCAT (MSGa, MSGb) :
(length: 225 bytes)
224350616365503235365f584d443a5348412d3235365f535357555f
4e555f5f49534b1034b36454cab2e7842c389f7d88ecb7df2027f705
9d88f02007dc18c911c9b4034d3c0f13f8f7ed9603b0927f23fbab10
374104df13ffa89b0ce3cc553b1495ff027886564d94b8d9165cd50e
5f654247959951bfac90839fca218bf8e2d1258eb7d7d9f733fe4cd5
58e6fa57bf1f801aae7d3a03414462410478ac925a6e3447a537627a
2163be005a422f55c08385clef7d051ca94593df5946314120faa871
65cba131c1da3aac429dc3d99a9bac7d4c4cbb8570b4d5ea10034144
61
ISK result: (length: 32 bytes)
6ea775b0fb3c31502687565a52150fc595c63fe901a11d5fc1995cd5
089a17ae

```

B.5.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50, 0x61, 0x73, 0x73, 0x77, 0x6f, 0x72, 0x64,
};
const uint8_t tc_CI[] = {
    0x0a, 0x41, 0x69, 0x6e, 0x69, 0x74, 0x69, 0x61, 0x74, 0x6f, 0x72, 0x0a,
    0x42, 0x72, 0x65, 0x73, 0x70, 0x6f, 0x6e, 0x64, 0x65, 0x72,
};
const uint8_t tc_sid[] = {
    0x34, 0xb3, 0x64, 0x54, 0xca, 0xb2, 0xe7, 0x84, 0x2c, 0x38, 0x9f, 0x7d,
    0x88, 0xec, 0xb7, 0xdf,
};
const uint8_t tc_g[] = {
    0x04, 0x99, 0x3b, 0x46, 0xe3, 0x0b, 0xa9, 0xcf, 0xc3, 0xdc, 0x2d, 0x3a,
    0xe2, 0xcf, 0x97, 0x33, 0xcf, 0x03, 0x99, 0x4e, 0x74, 0x38, 0x3c, 0x4e,
    0x1b, 0x4a, 0x92, 0xe8, 0xd6, 0xd4, 0x66, 0xb3, 0x21, 0xc4, 0xa6, 0x42,
    0x97, 0x91, 0x62, 0xfb, 0xde, 0x9e, 0x1c, 0x9a, 0x61, 0x80, 0xbd, 0x27,
    0xa0, 0x59, 0x44, 0x91, 0xe4, 0xc2, 0x31, 0xf5, 0x10, 0x06, 0xd0, 0xbf,
    0x79, 0x92, 0xd0, 0x71, 0x27,
};
const uint8_t tc_ysa[] = {
    0xc9, 0xe4, 0x7c, 0xa5, 0xde, 0xbd, 0x22, 0x85, 0x72, 0x7a, 0xf4, 0x7e,

```

```

    0x55,0xf5,0xb7,0x76,0x3f,0xa7,0x97,0x19,0xda,0x42,0x8f,0x80,
    0x01,0x90,0xcc,0x66,0x59,0xb4,0xea,0xfb,
};
const uint8_t tc_ADa[] = {
    0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
    0x04,0x78,0xac,0x92,0x5a,0x6e,0x34,0x47,0xa5,0x37,0x62,0x7a,
    0x21,0x63,0xbe,0x00,0x5a,0x42,0x2f,0x55,0xc0,0x83,0x85,0xc1,
    0xef,0x7d,0x05,0x1c,0xa9,0x45,0x93,0xdf,0x59,0x46,0x31,0x41,
    0x20,0xfa,0xa8,0x71,0x65,0xcb,0xa1,0x31,0xc1,0xda,0x3a,0xac,
    0x42,0x9d,0xc3,0xd9,0x9a,0x9b,0xac,0x7d,0x4c,0x4c,0xbb,0x85,
    0x70,0xb4,0xd5,0xea,0x10,
};
const uint8_t tc_yb[] = {
    0xa0,0xb7,0x68,0xba,0x75,0x55,0x62,0x1d,0x13,0x30,0x12,0xd1,
    0xde,0xe2,0x7a,0x00,0x13,0xc1,0xbc,0xfd,0xdd,0x67,0x58,0x11,
    0xdf,0x12,0x77,0x1e,0x44,0xd7,0x7b,0x10,
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0x04,0xdf,0x13,0xff,0xa8,0x9b,0x0c,0xe3,0xcc,0x55,0x3b,0x14,
    0x95,0xff,0x02,0x78,0x86,0x56,0x4d,0x94,0xb8,0xd9,0x16,0x5c,
    0xd5,0x0e,0x5f,0x65,0x42,0x47,0x95,0x99,0x51,0xbf,0xac,0x90,
    0x83,0x9f,0xca,0x21,0x8b,0xf8,0xe2,0xd1,0x25,0x8e,0xb7,0xd7,
    0xd9,0xf7,0x33,0xfe,0x4c,0xd5,0x58,0xe6,0xfa,0x57,0xbf,0x1f,
    0x80,0x1a,0xae,0x7d,0x3a,
};
const uint8_t tc_K[] = {
    0x27,0xf7,0x05,0x9d,0x88,0xf0,0x20,0x07,0xdc,0x18,0xc9,0x11,
    0xc9,0xb4,0x03,0x4d,0x3c,0x0f,0x13,0xf8,0xf7,0xed,0x96,0x03,
    0xb0,0x92,0x7f,0x23,0xfb,0xab,0x10,0x37,
};
const uint8_t tc_ISK_IR[] = {
    0xdd,0xc1,0xb1,0x33,0xc3,0x87,0xec,0xf3,0x44,0xc0,0xb4,0x96,
    0xbc,0x12,0x23,0x65,0x6c,0xd6,0xe7,0xd9,0x9a,0x5d,0xef,0x8b,
    0x3b,0x02,0x67,0x96,0x50,0x81,0x1f,0xc9,
};
const uint8_t tc_ISK_SY[] = {
    0x6e,0xa7,0x75,0xb0,0xfb,0x3c,0x31,0x50,0x26,0x87,0x56,0x5a,
    0x52,0x15,0x0f,0xc5,0x95,0xc6,0x3f,0xe9,0x01,0xa1,0x1d,0x5f,
    0xc1,0x99,0x5c,0xd5,0x08,0x9a,0x17,0xae,
};

```

B.5.8. Test case for scalar_mult_vfy with correct inputs

```

s: (length: 32 bytes)
  f012501c091ff9b99a123fffe571d8bc01e8077ee581362e1bd21399
  0835643b
X: (length: 65 bytes)
  0476ab88669dc640ca098b3d19ed87084d22d7e7c86b3b87451554d6
  93a7d98fb6bf0a6938fe0cec7be7563499ba3792909c8b9f4c936ef5
  2828b78a8d6254f49c
G.scalar_mult(s,X) (full coordinates): (length: 65 bytes)
  0492b0eb1fe6a988797a85e6de8ec5de7ec685c83164570d79f0d568
  b918bfe7718b049dac20ea4631d8c4f321ddb48d70416f4929eb9a85
  2528114d3a560537c7
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 32 bytes)
  92b0eb1fe6a988797a85e6de8ec5de7ec685c83164570d79f0d568b9
  18bfe771

```

B.5.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When including Y_i1 or Y_i2 in MSGa or MSGb the protocol MUST abort.

```

s: (length: 32 bytes)
  f012501c091ff9b99a123fffe571d8bc01e8077ee581362e1bd21399
  0835643b
Y_i1: (length: 65 bytes)
  0476ab88669dc640ca098b3d19ed87084d22d7e7c86b3b87451554d6
  93a7d98fb6bf0a6938fe0cec7be7563499ba3792909c8b9f4c936ef5
  2828b78a8d6254f4f3
Y_i2: (length: 1 bytes)
  00
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I

```

B.6. Test vector for CPace using group NIST P-384 and hash SHA-384

B.6.1. Test vectors for calculate_generator with group NIST P-384

B.6.3. Test vector for MSGb

Inputs

ADb = b'ADb'

yb (big endian): (length: 48 bytes)

50b0e36b95a2edfaa8342b843dddc90b175330f2399c1b36586dedda
3c255975f30be6a750f9404fccc62a6323b5e471

Outputs

Yb: (length: 97 bytes)

04e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262f7
46efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff8138317c8045c4
d2550e1566832b94acb91b670c4c4c00e59f5c15c74d4260e490caca
aa860c11b8f369b72d5871bd94Alternative correct value for Yb: $g^{(-yb)}$:

(length: 97 bytes)

04e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262f7
46efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff7ec7ce837fba3b
2daaf1ea997cd46b5346e498f3b3b3ff1a60a3ea38b2bd9f1a6f3535
5479f3ee470c9648d3a78e426b

MSGb: (length: 102 bytes)

6104e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262
f746efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff8138317c8045
c4d2550e1566832b94acb91b670c4c4c00e59f5c15c74d4260e490ca
caaa860c11b8f369b72d5871bd9403414462

B.6.4. Test vector for secret points K

scalar_mult_vfy(ya, Yb): (length: 48 bytes)

e5ef578c410effb4ec114998a59fa5832f6101be479f1a97b021f224
e378c3fb1f77f87a92e39fb415edf5458b3815bf

scalar_mult_vfy(yb, Ya): (length: 48 bytes)

e5ef578c410effb4ec114998a59fa5832f6101be479f1a97b021f224
e378c3fb1f77f87a92e39fb415edf5458b3815bf

B.6.5. Test vector for ISK calculation initiator/responder

```

unordered cat of transcript : (length: 204 bytes)
61047214fc512921b3fa0b555b41d841c9c20227fa1ab0dda5bfc051
f6de9be7983e6df11d4e8da738b739adfb85d8f5e80b2b4bbc66f3d
ffc02136ee19773d05f9c0242c0dd51857763de98a2fdfec73a4b101
0cbc419c7b23b50adedbb3ff6644034144616104e34cbd45b13ad115
52ea7100b19899fa52662e268f2086e21262f746efcb18e4b51ecfaf
2e8ebab82addb6245f9bb1ff8138317c8045c4d2550e1566832b94ac
b91b670c4c4c00e59f5c15c74d4260e490cacaaa860c11b8f369b72d
5871bd9403414462
DSI = G.DSI_ISK, b'CPaceP384_XMD:SHA-384_SSWU_NU__ISK' :
(length: 34 bytes)
4350616365503338345f584d443a5348412d3338345f535357555f4e
555f5f49534b
prefix_free_cat(DSI,sid,K) || MSGa || MSGb: (length: 305 bytes)
224350616365503338345f584d443a5348412d3338345f535357555f
4e555f5f49534b105b3773aa90e8f23c61563a4b645b276c30e5ef57
8c410effb4ec114998a59fa5832f6101be479f1a97b021f224e378c3
fb1f77f87a92e39fb415edf5458b3815bf61047214fc512921b3fa0b
555b41d841c9c20227fa1ab0dda5bfc051f6de9be7983e6df11d4e8d
a738b739adfb85d8f5e80b2b4bbc66f3dffc02136ee19773d05f9c0
242c0dd51857763de98a2fdfec73a4b1010cbc419c7b23b50adedbb3
ff6644034144616104e34cbd45b13ad11552ea7100b19899fa52662e
268f2086e21262f746efcb18e4b51ecfaf2e8ebab82addb6245f9bb1
ff8138317c8045c4d2550e1566832b94acb91b670c4c4c00e59f5c15
c74d4260e490cacaaa860c11b8f369b72d5871bd9403414462
ISK result: (length: 48 bytes)
401601de4a9f25bd57fc85985c9abf1de75191d68306b584547e6ac9
e959cf2df49a9bf2205c3617ce99a169971bdbf8

```

B.6.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 204 bytes)
6104e34cbd45b13ad11552ea7100b19899fa52662e268f2086e21262
f746efcb18e4b51ecfaf2e8ebab82addb6245f9bb1ff8138317c8045
c4d2550e1566832b94acb91b670c4c4c00e59f5c15c74d4260e490ca
caaa860c11b8f369b72d5871bd940341446261047214fc512921b3fa
0b555b41d841c9c20227fa1ab0dda5bfc051f6de9be7983e6df11d4e
8da738b739adfbdb85d8f5e80b2b4bbcb66f3dffc02136ee19773d05f9
c0242c0dd51857763de98a2fdfec73a4b1010cbc419c7b23b50adedb
b3ff664403414461
DSI = G.DSI_ISK, b'CPaceP384_XMD:SHA-384_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503338345f584d443a5348412d3338345f535357555f4e
555f5f49534b
prefix_free_cat (DSI,sid,K) || oCAT (MSGa,MSGb) :
(length: 305 bytes)
224350616365503338345f584d443a5348412d3338345f535357555f
4e555f5f49534b105b3773aa90e8f23c61563a4b645b276c30e5ef57
8c410effb4ec114998a59fa5832f6101be479f1a97b021f224e378c3
fb1f77f87a92e39fb415edf5458b3815bf6104e34cbd45b13ad11552
ea7100b19899fa52662e268f2086e21262f746efcb18e4b51ecfaf2e
8ebab82addb6245f9bb1ff8138317c8045c4d2550e1566832b94acb9
1b670c4c4c00e59f5c15c74d4260e490caca860c11b8f369b72d58
71bd940341446261047214fc512921b3fa0b555b41d841c9c20227fa
1ab0dda5bfc051f6de9be7983e6df11d4e8da738b739adfbdb85d8f5e
80b2b4bbcb66f3dffc02136ee19773d05f9c0242c0dd51857763de98a
2fdfec73a4b1010cbc419c7b23b50adedbb3ff664403414461
ISK result: (length: 48 bytes)
1eb17f7f7126a07acd510e9d60c84f63dc0113ac34f8d359e8f692a9
06f828bde926d9ff65202c9801e9884aa05a43b6

```

B.6.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t tc_CI[] = {
    0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
    0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const uint8_t tc_sid[] = {
    0x5b,0x37,0x73,0xaa,0x90,0xe8,0xf2,0x3c,0x61,0x56,0x3a,0x4b,
    0x64,0x5b,0x27,0x6c,
};
const uint8_t tc_g[] = {
    0x04,0xbb,0x6f,0x04,0x6a,0x60,0x1d,0x0a,0x0b,0x13,0x4c,0x62,
    0x21,0xe2,0x0e,0x83,0xc3,0xf9,0xac,0x03,0x90,0xbe,0x56,0xc5,
    0xa9,0x5b,0x68,0xeb,0xf4,0x1c,0x82,0xad,0xe6,0xf4,0x97,0x7e,
    0xa2,0x13,0x41,0x23,0x9d,0x19,0x4c,0x38,0xda,0xbd,0x1a,0x7e,

```

```
0xb5,0x88,0x7d,0x9f,0xed,0x25,0x50,0xa1,0xd5,0xe6,0x78,0x93,
0x27,0xf2,0xa0,0x39,0xcd,0x9c,0x41,0x23,0x9b,0x24,0x0f,0x77,
0x5f,0x5f,0x2b,0xef,0x87,0x44,0x56,0x1b,0x3a,0x7e,0x98,0xf3,
0x22,0x34,0xcb,0x1b,0x31,0x8f,0x66,0x61,0x6d,0xe7,0x77,0xae,
0xef,
};
const uint8_t tc_ya[] = {
0xef,0x43,0x3d,0xd5,0xad,0x14,0x2c,0x86,0x0e,0x7c,0xb6,0x40,
0x0d,0xd3,0x15,0xd3,0x88,0xd5,0xec,0x54,0x20,0xc5,0x50,0xe9,
0xd6,0xf0,0x90,0x7f,0x37,0x5d,0x98,0x8b,0xc4,0xd7,0x04,0x83,
0x7e,0x43,0x56,0x1c,0x49,0x7e,0x7d,0xd9,0x3e,0xdc,0xdb,0x9d,
};
const uint8_t tc_ADa[] = {
0x41,0x44,0x61,
};
const uint8_t tc_Ya[] = {
0x04,0x72,0x14,0xfc,0x51,0x29,0x21,0xb3,0xfa,0x0b,0x55,0x5b,
0x41,0xd8,0x41,0xc9,0xc2,0x02,0x27,0xfa,0x1a,0xb0,0xdd,0xa5,
0xbf,0xc0,0x51,0xf6,0xde,0x9b,0xe7,0x98,0x3e,0x6d,0xf1,0x1d,
0x4e,0x8d,0xa7,0x38,0xb7,0x39,0xad,0xfb,0xd8,0x5d,0x8f,0x5e,
0x80,0xb2,0xb4,0xbb,0xc6,0x6f,0x3d,0xff,0xc0,0x21,0x36,0xee,
0x19,0x77,0x3d,0x05,0xf9,0xc0,0x24,0x2c,0x0d,0xd5,0x18,0x57,
0x76,0x3d,0xe9,0x8a,0x2f,0xdf,0xec,0x73,0xa4,0xb1,0x01,0x0c,
0xbc,0x41,0x9c,0x7b,0x23,0xb5,0x0a,0xde,0xdb,0xb3,0xff,0x66,
0x44,
};
const uint8_t tc_yb[] = {
0x50,0xb0,0xe3,0x6b,0x95,0xa2,0xed,0xfa,0xa8,0x34,0x2b,0x84,
0x3d,0xdd,0xc9,0x0b,0x17,0x53,0x30,0xf2,0x39,0x9c,0x1b,0x36,
0x58,0x6d,0xed,0xda,0x3c,0x25,0x59,0x75,0xf3,0x0b,0xe6,0xa7,
0x50,0xf9,0x40,0x4f,0xcc,0xc6,0x2a,0x63,0x23,0xb5,0xe4,0x71,
};
const uint8_t tc_ADb[] = {
0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
0x04,0xe3,0x4c,0xbd,0x45,0xb1,0x3a,0xd1,0x15,0x52,0xea,0x71,
0x00,0xb1,0x98,0x99,0xfa,0x52,0x66,0x2e,0x26,0x8f,0x20,0x86,
0xe2,0x12,0x62,0xf7,0x46,0xef,0xcb,0x18,0xe4,0xb5,0x1e,0xcf,
0xaf,0x2e,0x8e,0xba,0xb8,0x2a,0xdd,0xb6,0x24,0x5f,0x9b,0xb1,
0xff,0x81,0x38,0x31,0x7c,0x80,0x45,0xc4,0xd2,0x55,0x0e,0x15,
0x66,0x83,0x2b,0x94,0xac,0xb9,0x1b,0x67,0x0c,0x4c,0x4c,0x00,
0xe5,0x9f,0x5c,0x15,0xc7,0x4d,0x42,0x60,0xe4,0x90,0xca,0xca,
0xaa,0x86,0x0c,0x11,0xb8,0xf3,0x69,0xb7,0x2d,0x58,0x71,0xbd,
0x94,
};
const uint8_t tc_K[] = {
0xe5,0xef,0x57,0x8c,0x41,0x0e,0xff,0xb4,0xec,0x11,0x49,0x98,
```



```

0xa5,0x9f,0xa5,0x83,0x2f,0x61,0x01,0xbe,0x47,0x9f,0x1a,0x97,
0xb0,0x21,0xf2,0x24,0xe3,0x78,0xc3,0xfb,0x1f,0x77,0xf8,0x7a,
0x92,0xe3,0x9f,0xb4,0x15,0xed,0xf5,0x45,0x8b,0x38,0x15,0xbf,
};
const uint8_t tc_ISK_IR[] = {
0x40,0x16,0x01,0xde,0x4a,0x9f,0x25,0xbd,0x57,0xfc,0x85,0x98,
0x5c,0x9a,0xbf,0x1d,0xe7,0x51,0x91,0xd6,0x83,0x06,0xb5,0x84,
0x54,0x7e,0x6a,0xc9,0xe9,0x59,0xcf,0x2d,0xf4,0x9a,0x9b,0xf2,
0x20,0x5c,0x36,0x17,0xce,0x99,0xa1,0x69,0x97,0x1b,0xdb,0xf8,
};
const uint8_t tc_ISK_SY[] = {
0x1e,0xb1,0x7f,0x7f,0x71,0x26,0xa0,0x7a,0xcd,0x51,0x0e,0x9d,
0x60,0xc8,0x4f,0x63,0xdc,0x01,0x13,0xac,0x34,0xf8,0xd3,0x59,
0xe8,0xf6,0x92,0xa9,0x06,0xf8,0x28,0xbd,0xe9,0x26,0xd9,0xff,
0x65,0x20,0x2c,0x98,0x01,0xe9,0x88,0x4a,0xa0,0x5a,0x43,0xb6,
};

```

B.6.8. Test case for scalar_mult_vfy with correct inputs

```

s: (length: 48 bytes)
6e8a99a5cdd408eae98e1b8aed286e7b12adbbdac7f2c628d9060ce9
2ae0d90bd57a564fd3500fbcce3425dc94ba0ade
X: (length: 97 bytes)
04a32d8d8e1057d37b090d92f46d0bac1874e6cd7c13774774385c30
39fa8fa3539884b436e49743d2d6279f5bd69dda5fe79fc6ecfb8547
bf32d8c64ac51f177a70041a1300944f255eea38ca7e964c9d02c5e7
e28d744e7cdc0bd80437363999
G.scalar_mult(s,X) (full coordinates): (length: 97 bytes)
045eb8202664ec20fed23ed6005c7be398174946a0f6a8a2e5fd2fed
9ca159f22652899f820a2d472f926f57de30035a9d11e8006fb66e79
f3db5d58bd5688954c7284d1e4a616a935dfb761955be13d29de5745
074a863140dcc9a5c0056ced3b
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 48 bytes)
5eb8202664ec20fed23ed6005c7be398174946a0f6a8a2e5fd2fed9c
a159f22652899f820a2d472f926f57de30035a9d

```

B.6.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When including Y_i1 or Y_i2 in MSGa or MSGb the protocol MUST abort.

```

s: (length: 48 bytes)
  6e8a99a5cdd408eae98e1b8aed286e7b12adbbdac7f2c628d9060ce9
  2ae0d90bd57a564fd3500fbcce3425dc94ba0ade
Y_i1: (length: 97 bytes)
  04a32d8d8e1057d37b090d92f46d0bac1874e6cd7c13774774385c30
  39fa8fa3539884b436e49743d2d6279f5bd69dda5fe79fc6ecfb8547
  bf32d8c64ac51f177a70041a1300944f255eea38ca7e964c9d02c5e7
  e28d744e7cdc0bd80437363938
Y_i2: (length: 1 bytes)
  00
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I

```

B.7. Test vector for CPace using group NIST P-521 and hash SHA-512

B.7.1. Test vectors for calculate_generator with group NIST P-521

Inputs

```

H   = SHA-512 with input block size 128 bytes.
PRS = b'Password' ; ZPAD length: 87 ;
DSI = b'CPaceP521_XMD:SHA-512_SSWU_NU_'
CI  = b'\nAinitiator\nBresponder'
CI  = 0a41696e69746961746f720a42726573706f6e646572
sid = 7e4b4791d6a8ef019b936c79fb7f2c57

```

Outputs

```

generator_string(PRS,G.DSI,CI,sid,H.s_in_bytes):
(length: 168 bytes)
  1e4350616365503532315f584d443a5348412d3531325f535357555f
  4e555f0850617373776f7264570000000000000000000000000000
  0000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000
  0a42726573706f6e646572107e4b4791d6a8ef019b936c79fb7f2c57
generator g: (length: 133 bytes)
  0400523c2be75a6fdb50e33d917597f182810ea6afe04b7297fccdfc
  f8c1c9f0f1a0c794056c729c275a654d1f9f52cd3d1d0ecc8f2f6a1b
  ab958d36cc539c558496a901bbe4fd573f2a6e6cc0c9afee3ee25c4b
  6f0474dd012eff5af0cbf55c4ec3c0ab4f1187353f815eb2a01ebc52
  d076d45a77a9b86d14fb21066df1d09f10b0a97546

```

B.7.2. Test vector for MSGa

Inputs

```
ADa = b'ADa'
```

```
ya (big endian): (length: 66 bytes)
```

```
016fac7bb757452e7b788d68a1510eda90113c65db1213fa08927d50
bcf2635fd66ca254e82927071001353e265082fd609af47ad06fab42
0c2295df4056ee9ff997
```

Outputs

```
Ya: (length: 133 bytes)
```

```
0400484dcee6d54cb356830cd764079360a03b06a7db1a82188e09c9
2e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f5999b7
e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e81826e
348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec7013a6
10fb2f3b4fb958cc860dd10c98745b9d89e37f2bf9
```

```
Alternative correct value for Ya:  $g^{(-ya)}$ :
```

```
(length: 133 bytes)
```

```
0400484dcee6d54cb356830cd764079360a03b06a7db1a82188e09c9
2e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f5999b7
e545d9fdf59f4c9acd408900528c1fe13dd519133edd30da817e7d91
cb732bef2246dba39e77601684d444674dfc714d12dc1676138fec59
ef04d0c4b046a73379f22ef3678ba462761c80d406
```

```
MSGa: (length: 139 bytes)
```

```
85010400484dcee6d54cb356830cd764079360a03b06a7db1a82188e
09c92e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f59
99b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e81
826e348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec70
13a610fb2f3b4fb958cc860dd10c98745b9d89e37f2bf903414461
```

B.7.3. Test vector for MSGb

Inputs

Adb = b'Adb'

yb (big endian): (length: 66 bytes)

```
011a946e2d0f48dc440ae3f4fd9126198237042fd1d41d037068c284
6d43ec130cbc55ef1208496be068f8682bcaf6156e51598e27c1fb24
d77b43957bbcl29bab80
```

Outputs

Yb: (length: 133 bytes)

```
0401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f0eef
9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc814639a9e
caff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6f724
2e65c36e7b960646c89aaf0262a4803ee4c90dlb58775a409a135bd1
8fedbf4ba0eae172b4fe8a0fada83d699e44f2f861
```

Alternative correct value for Yb: $g^{(-yb)}$:

(length: 133 bytes)

```
0401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f0eef
9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc814639a9e
caff07e87073313174763701b3d20e413754194dad18016e424908db
d19a3c918469f9b9376550fd9d5b7fc11b36f2e4a788a5bf65eca42e
701240b45f151e8d4b0175f05257c29661bb0d079e
```

MSGb: (length: 139 bytes)

```
85010401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f
0eef9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc81463
9a9ecaff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6
f7242e65c36e7b960646c89aaf0262a4803ee4c90dlb58775a409a13
5bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2f86103414462
```

B.7.4. Test vector for secret points K

scalar_mult_vfy(ya,Yb): (length: 66 bytes)

```
0070a7460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f1433
3ed8b873b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a237
45900f462f405debfd51
```

scalar_mult_vfy(yb,Ya): (length: 66 bytes)

```
0070a7460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f1433
3ed8b873b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a237
45900f462f405debfd51
```

B.7.5. Test vector for ISK calculation initiator/responder

```
unordered cat of transcript : (length: 278 bytes)
85010400484dcee6d54cb356830cd764079360a03b06a7db1a82188e
09c92e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f59
99b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e81
826e348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec70
13a610fb2f3b4fb958cc860dd10c98745b9d89e37f2bf90341446185
010401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f0e
ef9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc814639a
9ecaff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6f7
242e65c36e7b960646c89aaf0262a4803ee4c90dlb58775a409a135b
dl8fedbf4ba0eae172b4fe8a0fada83d699e44f2f86103414462
DSI = G.DSI_ISK, b'CPaceP521_XMD:SHA-512_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503532315f584d443a5348412d3531325f535357555f4e
555f5f49534b
prefix_free_cat (DSI,sid,K) ||MSGa||MSGb: (length: 397 bytes)
224350616365503532315f584d443a5348412d3531325f535357555f
4e555f5f49534b107e4b4791d6a8ef019b936c79fb7f2c57420070a7
460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f14333ed8b8
73b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a23745900f
462f405debfd5185010400484dcee6d54cb356830cd764079360a03b
06a7db1a82188e09c92e02d7e78a1e3710da9554db11697d242893e2
114d6cbee89f5999b7e545d9fdf59f4c9acd408901ad73e01ec22ae6
ecc122cf257e81826e348cd410ddb9245c61889fe97b2bbb98b2038e
b2ed23e989ec7013a610fb2f3b4fb958cc860dd10c98745b9d89e37f
2bf90341446185010401edf767bd7d9e67ff137b8f3210c55e9192e9
ac8a10f32a2f0eef9ce34524a543e0d4eb9b3328ca114b02ab23b291
f61b5bc814639a9ecaff07e870733131747637004c2df1bec8abe6b2
52e7fe91bdb6f7242e65c36e7b960646c89aaf0262a4803ee4c90dlb
58775a409a135bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2f8
6103414462
ISK result: (length: 64 bytes)
2b2c534c352c446773bd334fac2f2c50ef8cd7991bd4e070f85b0367
a2f7ffca445066cf20b756773687e1038b170896ec2677fe722acb0f
9e6c2f11830e808a
```

B.7.6. Test vector for ISK calculation parallel execution

```

ordered cat of transcript : (length: 278 bytes)
85010401edf767bd7d9e67ff137b8f3210c55e9192e9ac8a10f32a2f
0eef9ce34524a543e0d4eb9b3328ca114b02ab23b291f61b5bc81463
9a9ecaff07e870733131747637004c2df1bec8abe6b252e7fe91bdb6
f7242e65c36e7b960646c89aaf0262a4803ee4c90d1b58775a409a13
5bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2f8610341446285
010400484dcee6d54cb356830cd764079360a03b06a7db1a82188e09
c92e02d7e78a1e3710da9554db11697d242893e2114d6cbee89f5999
b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ecc122cf257e8182
6e348cd410ddb9245c61889fe97b2bbb98b2038eb2ed23e989ec7013
a610fb2f3b4fb958cc860dd10c98745b9d89e37f2bf903414461
DSI = G.DSI_ISK, b'CPaceP521_XMD:SHA-512_SSWU_NU__ISK':
(length: 34 bytes)
4350616365503532315f584d443a5348412d3531325f535357555f4e
555f5f49534b
prefix_free_cat (DSI,sid,K) || oCAT (MSGa,MSGb) :
(length: 397 bytes)
224350616365503532315f584d443a5348412d3531325f535357555f
4e555f5f49534b107e4b4791d6a8ef019b936c79fb7f2c57420070a7
460122c65d86bf9dd012ab45fc94be362619d1a1f0e75f14333ed8b8
73b5724616b88dadaaba5f28bb783aeb01f60df5fdb8c0a23745900f
462f405debfd5185010401edf767bd7d9e67ff137b8f3210c55e9192
e9ac8a10f32a2f0eef9ce34524a543e0d4eb9b3328ca114b02ab23b2
91f61b5bc814639a9ecaff07e870733131747637004c2df1bec8abe6
b252e7fe91bdb6f7242e65c36e7b960646c89aaf0262a4803ee4c90d
1b58775a409a135bd18fedbf4ba0eae172b4fe8a0fada83d699e44f2
f8610341446285010400484dcee6d54cb356830cd764079360a03b06
a7db1a82188e09c92e02d7e78a1e3710da9554db11697d242893e211
4d6cbee89f5999b7e545d9fdf59f4c9acd408901ad73e01ec22ae6ec
c122cf257e81826e348cd410ddb9245c61889fe97b2bbb98b2038eb2
ed23e989ec7013a610fb2f3b4fb958cc860dd10c98745b9d89e37f2b
f903414461
ISK result: (length: 64 bytes)
78c4dd7136309a2bbe1fdef3cf24a08690006b0c9de253b770c147dd
0800681c82e4e67a388ed1cd9182e595b8e9e3f2976a0e6dab48b2cd
205b19489e20f571

```

B.7.7. Corresponding ANSI-C initializers

```

const uint8_t tc_PRS[] = {
    0x50,0x61,0x73,0x73,0x77,0x6f,0x72,0x64,
};
const uint8_t tc_CI[] = {
    0x0a,0x41,0x69,0x6e,0x69,0x74,0x69,0x61,0x74,0x6f,0x72,0x0a,
    0x42,0x72,0x65,0x73,0x70,0x6f,0x6e,0x64,0x65,0x72,
};
const uint8_t tc_sid[] = {
    0x7e,0x4b,0x47,0x91,0xd6,0xa8,0xef,0x01,0x9b,0x93,0x6c,0x79,

```

```
    0xfb, 0x7f, 0x2c, 0x57,
};
const uint8_t tc_g[] = {
    0x04, 0x00, 0x52, 0x3c, 0x2b, 0xe7, 0x5a, 0x6f, 0xdb, 0x50, 0xe3, 0x3d,
    0x91, 0x75, 0x97, 0xf1, 0x82, 0x81, 0x0e, 0xa6, 0xaf, 0xe0, 0x4b, 0x72,
    0x97, 0xfc, 0xcd, 0xfc, 0xf8, 0xc1, 0xc9, 0xf0, 0xf1, 0xa0, 0xc7, 0x94,
    0x05, 0x6c, 0x72, 0x9c, 0x27, 0x5a, 0x65, 0x4d, 0x1f, 0x9f, 0x52, 0xcd,
    0x3d, 0x1d, 0x0e, 0xcc, 0x8f, 0x2f, 0x6a, 0x1b, 0xab, 0x95, 0x8d, 0x36,
    0xcc, 0x53, 0x9c, 0x55, 0x84, 0x96, 0xa9, 0x01, 0xbb, 0xe4, 0xfd, 0x57,
    0x3f, 0x2a, 0x6e, 0x6c, 0xc0, 0xc9, 0xaf, 0xee, 0x3e, 0xe2, 0x5c, 0x4b,
    0x6f, 0x04, 0x74, 0xdd, 0x01, 0x2e, 0xff, 0x5a, 0xf0, 0xcb, 0xf5, 0x5c,
    0x4e, 0xc3, 0xc0, 0xab, 0x4f, 0x11, 0x87, 0x35, 0x3f, 0x81, 0x5e, 0xb2,
    0xa0, 0x1e, 0xbc, 0x52, 0xd0, 0x76, 0xd4, 0x5a, 0x77, 0xa9, 0xb8, 0x6d,
    0x14, 0xfb, 0x21, 0x06, 0x6d, 0xf1, 0xd0, 0x9f, 0x10, 0xb0, 0xa9, 0x75,
    0x46,
};
const uint8_t tc_ya[] = {
    0x01, 0x6f, 0xac, 0x7b, 0xb7, 0x57, 0x45, 0x2e, 0x7b, 0x78, 0x8d, 0x68,
    0xa1, 0x51, 0x0e, 0xda, 0x90, 0x11, 0x3c, 0x65, 0xdb, 0x12, 0x13, 0xfa,
    0x08, 0x92, 0x7d, 0x50, 0xbc, 0xf2, 0x63, 0x5f, 0xd6, 0x6c, 0xa2, 0x54,
    0xe8, 0x29, 0x27, 0x07, 0x10, 0x01, 0x35, 0x3e, 0x26, 0x50, 0x82, 0xfd,
    0x60, 0x9a, 0xf4, 0x7a, 0xd0, 0x6f, 0xab, 0x42, 0x0c, 0x22, 0x95, 0xdf,
    0x40, 0x56, 0xee, 0x9f, 0xf9, 0x97,
};
const uint8_t tc_ADa[] = {
    0x41, 0x44, 0x61,
};
const uint8_t tc_Ya[] = {
    0x04, 0x00, 0x48, 0x4d, 0xce, 0xe6, 0xd5, 0x4c, 0xb3, 0x56, 0x83, 0x0c,
    0xd7, 0x64, 0x07, 0x93, 0x60, 0xa0, 0x3b, 0x06, 0xa7, 0xdb, 0x1a, 0x82,
    0x18, 0x8e, 0x09, 0xc9, 0x2e, 0x02, 0xd7, 0xe7, 0x8a, 0x1e, 0x37, 0x10,
    0xda, 0x95, 0x54, 0xdb, 0x11, 0x69, 0x7d, 0x24, 0x28, 0x93, 0xe2, 0x11,
    0x4d, 0x6c, 0xbe, 0xe8, 0x9f, 0x59, 0x99, 0xb7, 0xe5, 0x45, 0xd9, 0xfd,
    0xf5, 0x9f, 0x4c, 0x9a, 0xcd, 0x40, 0x89, 0x01, 0xad, 0x73, 0xe0, 0x1e,
    0xc2, 0x2a, 0xe6, 0xec, 0xc1, 0x22, 0xcf, 0x25, 0x7e, 0x81, 0x82, 0x6e,
    0x34, 0x8c, 0xd4, 0x10, 0xdd, 0xb9, 0x24, 0x5c, 0x61, 0x88, 0x9f, 0xe9,
    0x7b, 0x2b, 0xbb, 0x98, 0xb2, 0x03, 0x8e, 0xb2, 0xed, 0x23, 0xe9, 0x89,
    0xec, 0x70, 0x13, 0xa6, 0x10, 0xfb, 0x2f, 0x3b, 0x4f, 0xb9, 0x58, 0xcc,
    0x86, 0x0d, 0xd1, 0x0c, 0x98, 0x74, 0x5b, 0x9d, 0x89, 0xe3, 0x7f, 0x2b,
    0xf9,
};
const uint8_t tc_yb[] = {
    0x01, 0x1a, 0x94, 0x6e, 0x2d, 0x0f, 0x48, 0xdc, 0x44, 0x0a, 0xe3, 0xf4,
    0xfd, 0x91, 0x26, 0x19, 0x82, 0x37, 0x04, 0x2f, 0xd1, 0xd4, 0x1d, 0x03,
    0x70, 0x68, 0xc2, 0x84, 0x6d, 0x43, 0xec, 0x13, 0x0c, 0xbc, 0x55, 0xef,
    0x12, 0x08, 0x49, 0x6b, 0xe0, 0x68, 0xf8, 0x68, 0x2b, 0xca, 0xf6, 0x15,
    0x6e, 0x51, 0x59, 0x8e, 0x27, 0xc1, 0xfb, 0x24, 0xd7, 0x7b, 0x43, 0x95,
    0x7b, 0xbc, 0x12, 0x9b, 0xab, 0x80,
};
```

```
};
const uint8_t tc_ADb[] = {
    0x41,0x44,0x62,
};
const uint8_t tc_Yb[] = {
    0x04,0x01,0xed,0xf7,0x67,0xbd,0x7d,0x9e,0x67,0xff,0x13,0x7b,
    0x8f,0x32,0x10,0xc5,0x5e,0x91,0x92,0xe9,0xac,0x8a,0x10,0xf3,
    0x2a,0x2f,0x0e,0xef,0x9c,0xe3,0x45,0x24,0xa5,0x43,0xe0,0xd4,
    0xeb,0x9b,0x33,0x28,0xca,0x11,0x4b,0x02,0xab,0x23,0xb2,0x91,
    0xf6,0x1b,0x5b,0xc8,0x14,0x63,0x9a,0x9e,0xca,0xff,0x07,0xe8,
    0x70,0x73,0x31,0x31,0x74,0x76,0x37,0x00,0x4c,0x2d,0xf1,0xbe,
    0xc8,0xab,0xe6,0xb2,0x52,0xe7,0xfe,0x91,0xbd,0xb6,0xf7,0x24,
    0x2e,0x65,0xc3,0x6e,0x7b,0x96,0x06,0x46,0xc8,0x9a,0xaf,0x02,
    0x62,0xa4,0x80,0x3e,0xe4,0xc9,0x0d,0x1b,0x58,0x77,0x5a,0x40,
    0x9a,0x13,0x5b,0xd1,0x8f,0xed,0xbf,0x4b,0xa0,0xea,0xe1,0x72,
    0xb4,0xfe,0x8a,0x0f,0xad,0xa8,0x3d,0x69,0x9e,0x44,0xf2,0xf8,
    0x61,
};
const uint8_t tc_K[] = {
    0x00,0x70,0xa7,0x46,0x01,0x22,0xc6,0x5d,0x86,0xbf,0x9d,0xd0,
    0x12,0xab,0x45,0xfc,0x94,0xbe,0x36,0x26,0x19,0xd1,0xa1,0xf0,
    0xe7,0x5f,0x14,0x33,0x3e,0xd8,0xb8,0x73,0xb5,0x72,0x46,0x16,
    0xb8,0x8d,0xad,0xaa,0xba,0x5f,0x28,0xbb,0x78,0x3a,0xeb,0x01,
    0xf6,0x0d,0xf5,0xfd,0xb8,0xc0,0xa2,0x37,0x45,0x90,0x0f,0x46,
    0x2f,0x40,0x5d,0xeb,0xfd,0x51,
};
const uint8_t tc_ISK_IR[] = {
    0x2b,0x2c,0x53,0x4c,0x35,0x2c,0x44,0x67,0x73,0xbd,0x33,0x4f,
    0xac,0x2f,0x2c,0x50,0xef,0x8c,0xd7,0x99,0x1b,0xd4,0xe0,0x70,
    0xf8,0x5b,0x03,0x67,0xa2,0xf7,0xff,0xca,0x44,0x50,0x66,0xcf,
    0x20,0xb7,0x56,0x77,0x36,0x87,0xe1,0x03,0x8b,0x17,0x08,0x96,
    0xec,0x26,0x77,0xfe,0x72,0x2a,0xcb,0x0f,0x9e,0x6c,0x2f,0x11,
    0x83,0x0e,0x80,0x8a,
};
const uint8_t tc_ISK_SY[] = {
    0x78,0xc4,0xdd,0x71,0x36,0x30,0x9a,0x2b,0xbe,0x1f,0xde,0xf3,
    0xcf,0x24,0xa0,0x86,0x90,0x00,0x6b,0x0c,0x9d,0xe2,0x53,0xb7,
    0x70,0xc1,0x47,0xdd,0x08,0x00,0x68,0x1c,0x82,0xe4,0xe6,0x7a,
    0x38,0x8e,0xd1,0xcd,0x91,0x82,0xe5,0x95,0xb8,0xe9,0xe3,0xf2,
    0x97,0x6a,0x0e,0x6d,0xab,0x48,0xb2,0xcd,0x20,0x5b,0x19,0x48,
    0x9e,0x20,0xf5,0x71,
};
```

B.7.8. Test case for scalar_mult_vfy with correct inputs


```

s: (length: 66 bytes)
0182dd7925f1753419e4bf83429763acd37d64000cd5a175edf53a15
87dd986bc95acc1506991702b6ba1a9ee2458fee8efc00198cf0088c
480965ef65ff2048b856
X: (length: 133 bytes)
0400bf0a2632f954515e65c55553e25cde4c8bf3a48e5df86a3ef845
fcf15c8d9a4640171188ff835df48b8f934070d225daa591e270a9cc
539b82e8dc145caf38aeb900c30b83a1c9792e95c4a25f75b58001d3
6331c2b71a86591e1b510a1740335bc9947da1f6bab91b86900c9258
b28ee7b5ea33af2a8138a75cde4287613ab6673bcc
G.scalar_mult(s,X) (full coordinates): (length: 133 bytes)
040100763e7ebe6a051e2195b1980686a2a5d7edbc1d9284e38d1e9e
13673b65b6b3b5cb1b1ab146a315c32425edee8fdca06a07cf72d26d
31e38ec6a38481b4f18d8600b2a7df9cc7db6cbf75b2eee98f9f14e5
e24a789d45b9709278e8b74b30eb32d55fb8cfea4258dcf9de7fb36a
67326584d5c8121c4802801115b908b937361c9828
G.scalar_mult_vfy(s,X) (only X-coordinate):
(length: 66 bytes)
0100763e7ebe6a051e2195b1980686a2a5d7edbc1d9284e38d1e9e13
673b65b6b3b5cb1b1ab146a315c32425edee8fdca06a07cf72d26d31
e38ec6a38481b4f18d86

```

B.7.9. Invalid inputs for scalar_mult_vfy

For these test cases scalar_mult_vfy(y,.) MUST return the representation of the neutral element G.I. When including Y_i1 or Y_i2 in MSGa or MSGb the protocol MUST abort.

```

s: (length: 66 bytes)
0182dd7925f1753419e4bf83429763acd37d64000cd5a175edf53a15
87dd986bc95acc1506991702b6ba1a9ee2458fee8efc00198cf0088c
480965ef65ff2048b856
Y_i1: (length: 133 bytes)
0400bf0a2632f954515e65c55553e25cde4c8bf3a48e5df86a3ef845
fcf15c8d9a4640171188ff835df48b8f934070d225daa591e270a9cc
539b82e8dc145caf38aeb900c30b83a1c9792e95c4a25f75b58001d3
6331c2b71a86591e1b510a1740335bc9947da1f6bab91b86900c9258
b28ee7b5ea33af2a8138a75cde4287613ab6673b3a
Y_i2: (length: 1 bytes)
00
G.scalar_mult_vfy(s,Y_i1) = G.scalar_mult_vfy(s,Y_i2) = G.I

```

Authors' Addresses

Michel Abdalla
DFINITY - Zurich

Email: michel.abdalla@gmail.com

Bjoern Haase
Endress + Hauser Liquid Analysis - Gerlingen
Email: bjoern.m.haase@web.de

Julia Hesse
IBM Research Europe - Zurich
Email: JHS@zurich.ibm.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: 12 August 2022

A. Davidson
Brave Software
A. Faz-Hernandez
N. Sullivan
C.A. Wood
Cloudflare, Inc.
8 February 2022

Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups
draft-irtf-cfrg-voprf-09

Abstract

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol between client and server for computing the output of a Pseudorandom Function (PRF). The server provides the PRF secret key, and the client provides the PRF input. At the end of the protocol, the client learns the PRF output without learning anything about the PRF secret key, and the server learns neither the PRF input nor output. An OPRF can also satisfy a notion of 'verifiability', called a VOPRF. A VOPRF ensures clients can verify that the server used a specific private key during the execution of the protocol. A VOPRF can also be partially-oblivious, called a POPRF. A POPRF allows clients and servers to provide public input to the PRF computation. This document specifies an OPRF, VOPRF, and POPRF instantiated within standard prime-order groups, including elliptic curves.

Discussion Venues

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

Source for this draft and an issue tracker can be found at
<https://github.com/cfrg/draft-irtf-cfrg-voprf>.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 12 August 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	4
1.1. Change log	4
1.2. Requirements	7
1.3. Notation and Terminology	7
2. Preliminaries	8
2.1. Prime-Order Group	8
2.2. Discrete Log Equivalence Proofs	10
2.2.1. Proof Generation	10
2.2.2. Proof Verification	13
3. Protocol	16
3.1. Configuration	18
3.2. Key Generation and Context Setup	18
3.3. Online Protocol	20
3.3.1. OPRF Protocol	21
3.3.2. VOPRF Protocol	23
3.3.3. POPRF Protocol	24
4. Ciphersuites	27
4.1. Ciphersuite Registry	28
4.1.1. OPRF(ristretto255, SHA-512)	28
4.1.2. OPRF(decaf448, SHAKE-256)	29
4.1.3. OPRF(P-256, SHA-256)	29
4.1.4. OPRF(P-384, SHA-384)	30
4.1.5. OPRF(P-521, SHA-512)	30
4.2. Input Validation	31
4.2.1. DeserializeElement Validation	31

4.2.2. DeserializeScalar Validation	31
4.3. Future Ciphersuites	32
5. Application Considerations	32
5.1. Input Limits	32
5.2. External Interface Recommendations	32
5.3. Error Considerations	33
5.4. POPRF Public Input	33
6. Security Considerations	34
6.1. Security Properties	34
6.2. Security Assumptions	35
6.2.1. OPRF and VOPRF Assumptions	35
6.2.2. POPRF Assumptions	36
6.2.3. Static Diffie Hellman Attack and Security Limits	36
6.3. Domain Separation	37
6.4. Timing Leaks	37
7. Acknowledgements	37
8. References	37
8.1. Normative References	37
8.2. Informative References	38
Appendix A. Test Vectors	40
A.1. OPRF(ristretto255, SHA-512)	41
A.1.1. OPRF Mode	41
A.1.2. VOPRF Mode	41
A.1.3. POPRF Mode	43
A.2. OPRF(dec448, SHAKE-256)	44
A.2.1. OPRF Mode	44
A.2.2. VOPRF Mode	45
A.2.3. POPRF Mode	47
A.3. OPRF(P-256, SHA-256)	49
A.3.1. OPRF Mode	49
A.3.2. VOPRF Mode	50
A.3.3. POPRF Mode	51
A.4. OPRF(P-384, SHA-384)	53
A.4.1. OPRF Mode	53
A.4.2. VOPRF Mode	54
A.4.3. POPRF Mode	55
A.5. OPRF(P-521, SHA-512)	57
A.5.1. OPRF Mode	57
A.5.2. VOPRF Mode	58
A.5.3. POPRF Mode	60
Authors' Addresses	62

1. Introduction

A Pseudorandom Function (PRF) $F(k, x)$ is an efficiently computable function taking a private key k and a value x as input. This function is pseudorandom if the keyed function $K(_) = F(k, _)$ is indistinguishable from a randomly sampled function acting on the same domain and range as $K()$. An Oblivious PRF (OPRF) is a two-party protocol between a server and a client, where the server holds a PRF key k and the client holds some input x . The protocol allows both parties to cooperate in computing $F(k, x)$ such that the client learns $F(k, x)$ without learning anything about k ; and the server does not learn anything about x or $F(k, x)$. A Verifiable OPRF (VOPRF) is an OPRF wherein the server also proves to the client that $F(k, x)$ was produced by the key k corresponding to the server's public key the client knows. A Partially-Oblivious PRF (POPRF) is a variant of a VOPRF wherein client and server interact in computing $F(k, x, y)$, for some PRF F with server-provided key k , client-provided input x , and public input y , and client receives proof that $F(k, x, y)$ was computed using k corresponding to the public key that the client knows. A POPRF with fixed input y is functionally equivalent to a VOPRF.

OPRFs have a variety of applications, including: password-protected secret sharing schemes [JKKX16], privacy-preserving password stores [SJKS17], and password-authenticated key exchange or PAKE [I-D.irtf-cfrg-opaque]. Verifiable POPRFs are necessary in some applications such as Privacy Pass [I-D.ietf-privacypass-protocol]. Verifiable POPRFs have also been used for password-protected secret sharing schemes such as that of [JKK14].

This document specifies OPRF, VOPRF, and POPRF protocols built upon prime-order groups. The document describes each protocol variant, along with application considerations, and their security properties.

1.1. Change log

draft-09 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-09>):

- * Split syntax for OPRF, VOPRF, and POPRF functionalities.
- * Make Blind function fallible for invalid private and public inputs.
- * Specify key generation.
- * Remove serialization steps from core protocol functions.
- * Refactor protocol presentation for clarity.

- * Simplify security considerations.
- * Update application interface considerations.
- * Update test vectors.

draft-08 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-08>):

- * Adopt partially-oblivious PRF construction from [TCRSTW21].
- * Update P-384 suite to use SHA-384 instead of SHA-512.
- * Update test vectors.
- * Apply various editorial changes.

draft-07 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-07>):

- * Bind blinding mechanism to mode (additive for verifiable mode and multiplicative for base mode).
- * Add explicit errors for deserialization.
- * Document explicit errors and API considerations.
- * Adopt SHAKE-256 for decaf448 ciphersuite.
- * Normalize HashToScalar functionality for all ciphersuites.
- * Refactor and generalize DLEQ proof functionality and domain separation tags for use in other protocols.
- * Update test vectors.
- * Apply various editorial changes.

draft-06 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-06>):

- * Specify of group element and scalar serialization.
- * Remove info parameter from the protocol API and update domain separation guidance.
- * Fold Unblind function into Finalize.
- * Optimize ComputeComposites for servers (using knowledge of the private key).

- * Specify deterministic key generation method.
- * Update test vectors.
- * Apply various editorial changes.

draft-05 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-05>):

- * Move to ristretto255 and decaf448 ciphersuites.
- * Clean up ciphersuite definitions.
- * Pin domain separation tag construction to draft version.
- * Move key generation outside of context construction functions.
- * Editorial changes.

draft-04 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-04>):

- * Introduce Client and Server contexts for controlling verifiability and required functionality.
- * Condense API.
- * Remove batching from standard functionality (included as an extension)
- * Add Curve25519 and P-256 ciphersuites for applications that prevent strong-DH oracle attacks.
- * Provide explicit prime-order group API and instantiation advice for each ciphersuite.
- * Proof-of-concept implementation in sage.
- * Remove privacy considerations advice as this depends on applications.

draft-03 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-03>):

- * Certify public key during VerifiableFinalize.
- * Remove protocol integration advice.
- * Add text discussing how to perform domain separation.
- * Drop OPRF_/VOPRF_ prefix from algorithm names.

- * Make prime-order group assumption explicit.
- * Changes to algorithms accepting batched inputs.
- * Changes to construction of batched DLEQ proofs.
- * Updated ciphersuites to be consistent with hash-to-curve and added OPRF specific ciphersuites.

draft-02 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-02>):

- * Added section discussing cryptographic security and static DH oracles.
- * Updated batched proof algorithms.

draft-01 (<https://tools.ietf.org/html/draft-irtf-cfrg-voprf-01>):

- * Updated ciphersuites to be in line with <https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-04>.
- * Made some necessary modular reductions more explicit.

1.2. Requirements

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.

1.3. Notation and Terminology

The following functions and notation are used throughout the document.

- * For any object x , we write $\text{len}(x)$ to denote its length in bytes.
- * For two byte arrays x and y , write $x || y$ to denote their concatenation.
- * I2OSP and OS2IP: Convert a byte array to and from a non-negative integer as described in [RFC8017]. Note that these functions operate on byte arrays in big-endian byte order.

All algorithms and procedures described in this document are laid out in a Python-like pseudocode. The data types `PrivateInput` and `PublicInput` are opaque byte strings of arbitrary length no larger than 2^{13} octets.

String values such as "Finalize" are ASCII string literals.

The following terms are used throughout this document.

- * PRF: Pseudorandom Function.
- * OPRF: Oblivious Pseudorandom Function.
- * VOPRF: Verifiable Oblivious Pseudorandom Function.
- * POPRF: Partially Oblivious Pseudorandom Function.
- * Client: Protocol initiator. Learns pseudorandom function evaluation as the output of the protocol.
- * Server: Computes the pseudorandom function over a private key. Learns nothing about the client's input or output.

2. Preliminaries

The protocols in this document have two primary dependencies:

- * Group: A prime-order group implementing the API described below in Section 2.1, with base point defined in the corresponding reference for each group. (See Section 4 for these base points.)
- * Hash: A cryptographic hash function whose output length is N_h bytes long.

Section 4 specifies ciphersuites as combinations of Group and Hash.

2.1. Prime-Order Group

In this document, we assume the construction of an additive, prime-order group `Group` for performing all mathematical operations. Such groups are uniquely determined by the choice of the prime p that defines the order of the group. (There may, however, exist different representations of the group for a single p . Section 4 lists specific groups which indicate both order and representation.) We use $\text{GF}(p)$ to represent the finite field of order p . For the purpose of understanding and implementing this document, we take $\text{GF}(p)$ to be equal to the set of integers defined by $\{0, 1, \dots, p-1\}$.

The fundamental group operation is addition $+$ with identity element I . For any elements A and B of the group, $A + B = B + A$ is also a member of the group. Also, for any A in the group, there exists an element $-A$ such that $A + (-A) = (-A) + A = I$. Scalar multiplication is equivalent to the repeated application of the group operation on an element A with itself $r-1$ times, this is denoted as $r*A = A + \dots + A$. For any element A , $p*A=I$. Scalar base multiplication is equivalent to the repeated application of the group operation on the fixed group generator with itself $r-1$ times, and is denoted as $\text{ScalarBaseMult}(r)$. The set of scalars corresponds to $\text{GF}(p)$. This document uses types `Element` and `Scalar` to denote elements of the group and its set of scalars, respectively.

We now detail a number of member functions that can be invoked on a prime-order group.

- * `Order()`: Outputs the order of the group (i.e. p).
- * `Identity()`: Outputs the identity element of the group (i.e. I).
- * `HashToGroup(x)`: A member function of `Group` that deterministically maps an array of bytes x to an element of `Group`. The map must ensure that, for any adversary receiving $R = \text{HashToGroup}(x)$, it is computationally difficult to reverse the mapping. This function is optionally parameterized by a domain separation tag (DST); see Section 4.
- * `HashToScalar(x)`: A member function of `Group` that deterministically maps an array of bytes x to an element in $\text{GF}(p)$. This function is optionally parameterized by a DST; see Section 4.
- * `RandomScalar()`: A member function of `Group` that chooses at random a non-zero element in $\text{GF}(p)$.
- * `ScalarInverse(s)`: Compute the multiplicative inverse of input `Scalar` s modulo the prime order of the group p .
- * `SerializeElement(A)`: A member function of `Group` that maps a group element A to a unique byte array `buf` of fixed length N_e . The output type of this function is `SerializedElement`.
- * `DeserializeElement(buf)`: A member function of `Group` that maps a byte array `buf` to a group element A , or fails if the input is not a valid byte representation of an element. This function can raise a `DeserializeError` if deserialization fails or A is the identity element of the group; see Section 4.2.

- * `SerializeScalar(s)`: A member function of `Group` that maps a scalar element `s` to a unique byte array `buf` of fixed length `Ns`. The output type of this function is `SerializedScalar`.
- * `DeserializeScalar(buf)`: A member function of `Group` that maps a byte array `buf` to a scalar `s`, or fails if the input is not a valid byte representation of a scalar. This function can raise a `DeserializeError` if deserialization fails; see Section 4.2.

It is convenient in cryptographic applications to instantiate such prime-order groups using elliptic curves, such as those detailed in [SEC2]. For some choices of elliptic curves (e.g. those detailed in [RFC7748], which require accounting for cofactors) there are some implementation issues that introduce inherent discrepancies between standard prime-order groups and the elliptic curve instantiation. In this document, all algorithms that we detail assume that the group is a prime-order group, and this **MUST** be upheld by any implementation. That is, any curve instantiation should be written such that any discrepancies with a prime-order group instantiation are removed. See Section 4 for advice corresponding to the implementation of this interface for specific definitions of elliptic curves.

2.2. Discrete Log Equivalence Proofs

Another important piece of the OPRF protocols in this document is proving that the discrete log of two values is identical in zero knowledge, i.e., without revealing the discrete logarithm. This is referred to as a discrete log equivalence (DLEQ) proof. This section describes functions for non-interactively proving and verifying this type of statement, built on a Chaum-Pedersen [ChaumPedersen] proof. It is split into two sub-sections: one for generating the proof, which is done by servers in the verifiable protocols, and another for verifying the proof, which is done by clients in the protocol.

2.2.1. Proof Generation

Generating a proof is done with the `GenerateProof` function, defined below. This function takes four `Elements`, `A`, `B`, `C`, and `D`, and a single group `Scalar` `k`, and produces a proof that $k \cdot A == B$ and $k \cdot C == D$. The output is a value of type `Proof`, which is a tuple of two `Scalar` values.

Input:

```
Scalar k
Element A
Element B
Element C
Element D
```

Output:

```
Proof proof
```

Parameters:

```
Group G
```

```
def GenerateProof(k, A, B, C, D)
    Cs = [C]
    Ds = [D]
    (M, Z) = ComputeCompositesFast(k, B, Cs, Ds)

    r = G.RandomScalar()
    t2 = r * A
    t3 = r * M

    Bm = G.SerializeElement(B)
    a0 = G.SerializeElement(M)
    a1 = G.SerializeElement(Z)
    a2 = G.SerializeElement(t2)
    a3 = G.SerializeElement(t3)

    h2Input = I2OSP(len(Bm), 2) || Bm ||
              I2OSP(len(a0), 2) || a0 ||
              I2OSP(len(a1), 2) || a1 ||
              I2OSP(len(a2), 2) || a2 ||
              I2OSP(len(a3), 2) || a3 ||
              "Challenge"

    c = G.HashToScalar(h2Input)
    s = (r - c * k) mod G.Order()

    return [c, s]
```

The helper function `ComputeCompositesFast` is as defined below.

Input:

```
Scalar k
Element B
Element Cs[m]
Element Ds[m]
```

Output:

```
Element M
Element Z
```

Parameters:

```
Group G
PublicInput contextString
```

```
def ComputeCompositesFast(k, B, Cs, Ds):
    Bm = G.SerializeElement(B)
    seedDST = "Seed-" || contextString
    h1Input = I2OSP(len(Bm), 2) || Bm ||
              I2OSP(len(seedDST), 2) || seedDST
    seed = Hash(h1Input)

    M = G.Identity()
    for i = 0 to range(m):
        Ci = G.SerializeElement(Cs[i])
        Di = G.SerializeElement(Ds[i])
        h2Input = I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
                  I2OSP(len(Ci), 2) || Ci ||
                  I2OSP(len(Di), 2) || Di ||
                  "Composite"

        di = G.HashToScalar(h2Input)
        M = di * Cs[i] + M

    Z = k * M

    return (M, Z)
```

When used in the protocol described in Section 3, the parameter contextString is as defined in Section 3.1.

ComputeCompositesFast takes lists of inputs, rather than a single input. Applications can take advantage of this functionality by invoking GenerateProof on batches of inputs to produce a combined, constant-size proof. In particular, servers can produce a single, constant-sized proof for N DLEQ inputs, rather than one proof per

DLEQ input. This optimization benefits clients and servers since it amortizes the cost of proof generation and bandwidth across multiple requests.

2.2.2. Proof Verification

Verifying a proof is done with the `VerifyProof` function, defined below. This function takes four Elements, `A`, `B`, `C`, and `D`, along with a `Proof` value output from `GenerateProof`. It outputs a single boolean value indicating whether or not the proof is valid for the given DLEQ inputs.

Input:

```

Element A
Element B
Element C
Element D
Proof proof

```

Output:

```

boolean verified

```

Parameters:

```

Group G

```

Errors: DeserializeError

```

def VerifyProof(A, B, C, D, proof):
    Cs = [C]
    Ds = [D]

    (M, Z) = ComputeComposites(B, Cs, Ds)
    c = proof[0]
    s = proof[1]

    t2 = ((s * A) + (c * B))
    t3 = ((s * M) + (c * Z))

    Bm = G.SerializeElement(B)
    a0 = G.SerializeElement(M)
    a1 = G.SerializeElement(Z)
    a2 = G.SerializeElement(t2)
    a3 = G.SerializeElement(t3)

    h2Input = I2OSP(len(Bm), 2) || Bm ||
              I2OSP(len(a0), 2) || a0 ||
              I2OSP(len(a1), 2) || a1 ||
              I2OSP(len(a2), 2) || a2 ||
              I2OSP(len(a3), 2) || a3 ||
              "Challenge"

    expectedC = G.HashToScalar(h2Input)

    return expectedC == c

```

The definition of ComputeComposites is given below.

Input:

```
Element B
Element Cs[m]
Element Ds[m]
```

Output:

```
Element M
Element Z
```

Parameters:

```
Group G
PublicInput contextString
```

```
def ComputeComposites(B, Cs, Ds):
    Bm = G.SerializeElement(B)
    seedDST = "Seed-" || contextString
    h1Input = I2OSP(len(Bm), 2) || Bm ||
              I2OSP(len(seedDST), 2) || seedDST
    seed = Hash(h1Input)

    M = G.Identity()
    Z = G.Identity()
    for i = 0 to m-1:
        Ci = G.SerializeElement(Cs[i])
        Di = G.SerializeElement(Ds[i])
        h2Input = I2OSP(len(seed), 2) || seed || I2OSP(i, 2) ||
                  I2OSP(len(Ci), 2) || Ci ||
                  I2OSP(len(Di), 2) || Di ||
                  "Composite"

        di = G.HashToScalar(h2Input)
        M = di * Cs[i] + M
        Z = di * Ds[i] + Z

    return (M, Z)
```

When used in the protocol described in Section 3, the parameter contextString is as defined in Section 3.1.

As with the proof generation case, proof verification can be batched. ComputeComposites is defined in terms of a batch of inputs. Implementations can take advantage of this behavior by also batching inputs to VerifyProof, respectively.

3. Protocol

In this section, we define three OPRF protocol variants -- a base mode, verifiable mode, and partially-oblivious mode -- with the following properties.

In the base mode, a client and server interact to compute $\text{output} = F(\text{skS}, \text{input})$, where input is the client's private input, skS is the server's private key, and output is the OPRF output. The client learns output and the server learns nothing. This interaction is shown below.

Client	Server(skS)
<div style="display: flex; justify-content: space-between;"> blind, blindedElement = Blind(input) </div>	
	blindedElement ----->
	evaluatedElement = Evaluate(blindedElement)
	evaluatedElement <-----
<div style="display: flex; justify-content: space-between;"> output = Finalize(input, blind, evaluatedElement, blindedElement) </div>	

Figure 1: OPRF protocol overview

In the verifiable mode, the client additionally receives proof that the server used skS in computing the function. To achieve verifiability, as in the original work of [JKK14], the server provides a zero-knowledge proof that the key provided as input by the server in the Evaluate function is the same key as it used to produce the server's public key, pkS , which the client receives as input to the protocol. This proof does not reveal the server's private key to the client. This interaction is shown below.

```

Client(pkS)          <---- pkS ----- Server(skS)
-----
blind, blindedElement = Blind(input)

                        blindedElement
                        ----->

evaluatedElement, proof = Evaluate(blindedElement)

evaluatedElement, proof
<-----

output = Finalize(input, blind, evaluatedElement,
                    blindedElement, proof)

```

Figure 2: VOPRF protocol overview with additional proof

The partially-oblivious mode extends the VOPRF mode such that the client and server can additionally provide a public input info that is used in computing the pseudorandom function. That is, the client and server interact to compute $\text{output} = F(\text{skS}, \text{input}, \text{info})$. To support additional public input, the client and server augment the pkS and skS, respectively, using the info value, as in [TCRSTW21].

```

Client(pkS, info)     <---- pkS ----- Server(skS, info)
-----
blind, blindedElement, tweakedKey = Blind(input, info)

                        blindedElement
                        ----->

evaluatedElement, proof = Evaluate(blindedElement, info)

evaluatedElement, proof
<-----

output = Finalize(input, blind, evaluatedElement,
                    blindedElement, proof, info, tweakedKey)

```

Figure 3: POPRF protocol overview with additional public input

Each protocol consists of an offline setup phase and an online phase, described in Section 3.2 and Section 3.3, respectively. Configuration details for the offline phase are described in Section 3.1.

3.1. Configuration

Each of the three protocol variants are identified with a one-byte value:

Mode	Value
modeOPRF	0x00
modeVOPRF	0x01
modePOPRF	0x02

Table 1

Additionally, each protocol variant is instantiated with a ciphersuite, or suite. Each ciphersuite is identified with a two-byte value, referred to as suiteID; see Section 4 for the registry of initial values.

The mode and ciphersuite ID values are combined to create a "context string" used throughout the protocol with the following function:

```
def CreateContextString(mode, suiteID):
    return "VOPRF09-" || I2OSP(mode, 1) || I2OSP(suiteID, 2)

[[RFC editor: please change "VOPRF09" to "RFCXXXX", where XXXX is the
final number, here and elsewhere before publication.]]
```

3.2. Key Generation and Context Setup

In the offline setup phase, both the client and server create a context used for executing the online phase of the protocol after agreeing on a mode and ciphersuite value suiteID. The server key pair (skS, pkS) is generated using the following function, which accepts a randomly generated seed of length Ns and optional public info string. The constant Ns corresponds to the size of a serialized Scalar and is defined in Section 2.1.

Input:

```
opaque seed[Ns]
PublicInput info
```

Output:

```
Scalar skS
Element pkS
```

Parameters:

```
Group G
PublicInput contextString
```

Errors: DeriveKeyPairError

```
def DeriveKeyPair(seed, info):
    contextString = CreateContextString(mode, suiteID)
    deriveInput = seed || I2OSP(len(info), 2) || info
    counter = 0
    skS = 0
    while skS == 0:
        if counter > 255:
            raise DeriveKeyPairError
        skS = G.HashToScalar(deriveInput || I2OSP(counter, 1),
                             DST = "DeriveKeyPair" || contextString)
        counter = counter + 1
    pkS = G.ScalarBaseMult(skS)
    return skS, pkS
```

The OPRF variant server and client contexts are created as follows:

```
def SetupOPRFServer(suiteID, skS):
    contextString = CreateContextString(modeOPRF, suiteID)
    return OPRFServerContext(contextString, skS)

def SetupOPRFClient(suiteID):
    contextString = CreateContextString(modeOPRF, suiteID)
    return OPRFClientContext(contextString)
```

The VOPRF variant server and client contexts are created as follows:

```
def SetupVOPRFServer(suiteID, skS, pkS):  
    contextString = CreateContextString(modeVOPRF, suiteID)  
    return VOPRFServerContext(contextString, skS)  
  
def SetupVOPRFClient(suiteID, pkS):  
    contextString = CreateContextString(modeVOPRF, suiteID)  
    return VOPRFClientContext(contextString, pkS)
```

The POPRF variant server and client contexts are created as follows:

```
def SetupPOPRFServer(suiteID, skS, pkS):  
    contextString = CreateContextString(modePOPRF, suiteID)  
    return POPRFServerContext(contextString, skS)  
  
def SetupPOPRFClient(suiteID, pkS):  
    contextString = CreateContextString(modePOPRF, suiteID)  
    return POPRFClientContext(contextString, pkS)
```

3.3. Online Protocol

In the online phase, the client and server engage in a two message protocol to compute the protocol output. This section describes the protocol details for each protocol variant. Throughout each description the following parameters are assumed to exist:

- * G , a prime-order Group implementing the API described in Section 2.1.
- * `contextString`, a PublicInput domain separation tag constructed during context setup as created in Section 3.1.
- * `skS` and `pkS`, a Scalar and Element representing the private and public keys configured for client and server in Section 3.2.

Applications serialize protocol messages between client and server for transmission. Specifically, values of type `Element` are serialized to `SerializedElement` values, and values of type `Proof` are serialized as the concatenation of two `SerializedScalar` values. Deserializing these values can fail, in which case the application MUST abort the protocol with a `DeserializeError` failure.

Applications MUST check that input `Element` values received over the wire are not the group identity element. This check is handled when deserializing `Element` values using `DeserializeElement`; see Section 4.2 for more information on input validation.

3.3.1. OPRF Protocol

The OPRF protocol begins with the client blinding its input, as described by the Blind function below. Note that this function can fail with an `InvalidInputError` error for certain inputs that map to the group identity element. Dealing with this failure is an application-specific decision; see Section 5.3.

Input:

PrivateInput input

Output:

Scalar blind
Element blindedElement

Parameters:

Group G

Errors: `InvalidInputError`

```
def Blind(input):  
    blind = G.RandomScalar()  
    P = G.HashToGroup(input)  
    if P == G.Identity():  
        raise InvalidInputError  
    blindedElement = blind * P  
  
    return blind, blindedElement
```

Clients store blind locally, and send blindedElement to the server for evaluation. Upon receipt, servers process blindedElement using the Evaluate function described below.

Input:

Element blindedElement

Output:

Element evaluatedElement

Parameters:

Scalar skS

```
def Evaluate(blindedElement):  
    evaluatedElement = skS * blindedElement  
    return evaluatedElement
```

Servers send the output evaluatedElement to clients for processing.
Recall that servers may batch multiple client inputs to Evaluate.

Upon receipt of evaluatedElement, clients process it to complete the
OPRF evaluation with the Finalize function described below.

Input:

PrivateInput input
Scalar blind
Element evaluatedElement

Output:

opaque output[Nh]

Parameters:

Group G

```
def Finalize(input, blind, evaluatedElement):  
    N = G.ScalarInverse(blind) * evaluatedElement  
    unblindedElement = G.SerializeElement(N)  
  
    hashInput = I2OSP(len(input), 2) || input ||  
                I2OSP(len(unblindedElement), 2) || unblindedElement ||  
                "Finalize"  
    return Hash(hashInput)
```


3.3.2. VOPRF Protocol

The VOPRF protocol begins with the client blinding its input, using the same Blind function as in Section 3.3.1. Clients store the output blind locally and send blindedElement to the server for evaluation. Upon receipt, servers process blindedElement to compute an evaluated element and DLEQ proof using the following Evaluate function.

Input:

Element blindedElement

Output:

Element evaluatedElement
Proof proof

Parameters:

Group G
Scalar skS
Element pkS

```
def Evaluate(blindedElement):  
    evaluatedElement = skS * blindedElement  
    proof = GenerateProof(skS, G.Generator(), pkS,  
                          blindedElement, evaluatedElement)  
    return evaluatedElement, proof
```

The server sends both evaluatedElement and proof back to the client. Upon receipt, the client processes both values to complete the VOPRF computation using the Finalize function below.

Input:

```
PrivateInput input
Scalar blind
Element evaluatedElement
Element blindedElement
Proof proof
```

Output:

```
opaque output[Nh]
```

Parameters:

```
Group G
Element pkS
```

Errors: VerifyError

```
def Finalize(input, blind, evaluatedElement, blindedElement, proof):
    if VerifyProof(G.Generator(), pkS, blindedElement,
        evaluatedElement, proof) == false:
        raise VerifyError

    N = G.ScalarInverse(blind) * evaluatedElement
    unblindedElement = G.SerializeElement(N)

    hashInput = I2OSP(len(input), 2) || input ||
        I2OSP(len(unblindedElement), 2) || unblindedElement ||
        "Finalize"
    return Hash(hashInput)
```

3.3.3. POPRF Protocol

The POPRF protocol begins with the client blinding its input, using the following modified Blind function. Note that this function can fail with an InvalidInputError error for certain private inputs that map to the group identity element, as well as certain public inputs that map to invalid public keys for server evaluation. Dealing with either failure is an application-specific decision; see Section 5.3.

Input:

```
PrivateInput input
PublicInput info
```

Output:

```
Scalar blind
Element blindedElement
Element tweakedKey
```

Parameters:

```
Group G
Element pkS
```

Errors: InvalidInputError

```
def Blind(input, info):
    framedInfo = "Info" || I2OSP(len(info), 2) || info
    m = G.HashToScalar(framedInfo)
    T = G.ScalarBaseMult(m)
    tweakedKey = T + pkS
    if tweakedKey == G.Identity():
        raise InvalidInputError

    blind = G.RandomScalar()
    P = G.HashToGroup(input)
    if P == G.Identity():
        raise InvalidInputError

    blindedElement = blind * P

    return blind, blindedElement, tweakedKey
```

Clients store the outputs blind and tweakedKey locally and send blindedElement to the server for evaluation. Upon receipt, servers process blindedElement to compute an evaluated element and DLEQ proof using the following Evaluate function.

Input:

Element blindedElement
PublicInput info

Output:

Element evaluatedElement
Proof proof

Parameters:

Group G
Scalar skS
Element pkS

Errors: InverseError

```
def Evaluate(blindedElement, info):  
    framedInfo = "Info" || I2OSP(len(info), 2) || info  
    m = G.HashToScalar(framedInfo)  
    t = skS + m  
    if t == 0:  
        raise InverseError  
  
    evaluatedElement = G.ScalarInverse(t) * blindedElement  
  
    tweakedKey = G.ScalarBaseMult(t)  
    proof = GenerateProof(t, G.Generator(), tweakedKey,  
                          evaluatedElement, blindedElement)  
  
    return evaluatedElement, proof
```

The server sends both evaluatedElement and proof back to the client. Upon receipt, the client processes both values to complete the VOPRF computation using the Finalize function below.

Input:

```

PrivateInput input
Scalar blind
Element evaluatedElement
Element blindedElement
Proof proof
PublicInput info
Element tweakedKey

```

Output:

```
opaque output[Nh]
```

Parameters:

```

Group G
Element pkS

```

Errors: VerifyError

```

def Finalize(input, blind, evaluatedElement, blindedElement,
              proof, info, tweakedKey):
    if VerifyProof(G.Generator(), tweakedKey, evaluatedElement,
                  blindedElement, proof) == false:
        raise VerifyError

    N = G.ScalarInverse(blind) * evaluatedElement
    unblindedElement = G.SerializeElement(N)

    hashInput = I2OSP(len(input), 2) || input ||
                I2OSP(len(info), 2) || info ||
                I2OSP(len(unblindedElement), 2) || unblindedElement ||
                "Finalize"
    return Hash(hashInput)

```

4. Ciphersuites

A ciphersuite (also referred to as 'suite' in this document) for the protocol wraps the functionality required for the protocol to take place. The ciphersuite should be available to both the client and server, and agreement on the specific instantiation is assumed throughout.

A ciphersuite contains instantiations of the following functionalities:

- * **Group:** A prime-order Group exposing the API detailed in Section 2.1, with base point defined in the corresponding reference for each group. Each group also specifies HashToGroup, HashToScalar, and serialization functionalities. For HashToGroup, the domain separation tag (DST) is constructed in accordance with the recommendations in [I-D.irtf-cfrg-hash-to-curve], Section 3.1. For HashToScalar, each group specifies an integer order that is used in reducing integer values to a member of the corresponding scalar field.
- * **Hash:** A cryptographic hash function whose output length is N_h bytes long.

This section specifies an initial registry of ciphersuites with supported groups and hash functions. It also includes implementation details for each ciphersuite, focusing on input validation, as well as requirements for future ciphersuites.

4.1. Ciphersuite Registry

For each ciphersuite, contextString is that which is computed in the Setup functions. Applications should take caution in using ciphersuites targeting P-256 and ristretto255. See Section 6.2 for related discussion.

4.1.1. OPRF(ristretto255, SHA-512)

- * **Group:** ristretto255 [RISTRETTO]
 - **HashToGroup():** Use hash_to_ristretto255 [I-D.irtf-cfrg-hash-to-curve] with DST = "HashToGroup-" || contextString, and expand_message = expand_message_xmd using SHA-512.
 - **HashToScalar():** Compute uniform_bytes using expand_message = expand_message_xmd, DST = "HashToScalar-" || contextString, and output length 64, interpret uniform_bytes as a 512-bit integer in little-endian order, and reduce the integer modulo Order().
 - **Serialization:** Both group elements and scalars are encoded in $N_e = N_s = 32$ bytes. For group elements, use the 'Encode' and 'Decode' functions from [RISTRETTO]. For scalars, ensure they are fully reduced modulo Order() and in little-endian order.
- * **Hash:** SHA-512, and $N_h = 64$.
- * **ID:** 0x0001

4.1.2. OPRF(dec448, SHAKE-256)

- * Group: dec448 [RISTRETTO]
 - HashToGroup(): Use `hash_to_dec448` [I-D.irtf-cfrg-hash-to-curve] with `DST = "HashToGroup-" || contextString`, and `expand_message = expand_message_xof` using SHAKE-256.
 - HashToScalar(): Compute `uniform_bytes` using `expand_message = expand_message_xof`, `DST = "HashToScalar-" || contextString`, and output length 64, interpret `uniform_bytes` as a 512-bit integer in little-endian order, and reduce the integer modulo `Order()`.
 - Serialization: Both group elements and scalars are encoded in `Ne = Ns = 56` bytes. For group elements, use the 'Encode' and 'Decode' functions from [RISTRETTO]. For scalars, ensure they are fully reduced modulo `Order()` and in little-endian order.
- * Hash: SHAKE-256, and `Nh = 64`.
- * ID: 0x0002

4.1.3. OPRF(P-256, SHA-256)

- * Group: P-256 (secp256r1) [x9.62]
 - HashToGroup(): Use `hash_to_curve` with suite `P256_XMD:SHA-256_SSWU_RO_` [I-D.irtf-cfrg-hash-to-curve] and `DST = "HashToGroup-" || contextString`.
 - HashToScalar(): Use `hash_to_field` from [I-D.irtf-cfrg-hash-to-curve] using `L = 48`, `expand_message_xmd` with SHA-256, `DST = "HashToScalar-" || contextString`, and prime modulus equal to `Order()`.
 - Serialization: Elements are serialized as `Ne = 33` byte strings using compressed point encoding for the curve [SEC1]. Scalars are serialized as `Ns = 32` byte strings by fully reducing the value modulo `Order()` and in big-endian order.
- * Hash: SHA-256, and `Nh = 32`.
- * ID: 0x0003

4.1.4. OPRF(P-384, SHA-384)

- * Group: P-384 (secp384r1) [x9.62]
 - HashToGroup(): Use hash_to_curve with suite P384_XMD:SHA-384_SSWU_RO_ [I-D.irtf-cfrg-hash-to-curve] and DST = "HashToGroup-" || contextString.
 - HashToScalar(): Use hash_to_field from [I-D.irtf-cfrg-hash-to-curve] using L = 72, expand_message_xmd with SHA-384, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().
 - Serialization: Elements are serialized as Ne = 49 byte strings using compressed point encoding for the curve [SEC1]. Scalars are serialized as Ns = 48 byte strings by fully reducing the value modulo Order() and in big-endian order.
- * Hash: SHA-384, and Nh = 48.
- * ID: 0x0004

4.1.5. OPRF(P-521, SHA-512)

- * Group: P-521 (secp521r1) [x9.62]
 - HashToGroup(): Use hash_to_curve with suite P521_XMD:SHA-512_SSWU_RO_ [I-D.irtf-cfrg-hash-to-curve] and DST = "HashToGroup-" || contextString.
 - HashToScalar(): Use hash_to_field from [I-D.irtf-cfrg-hash-to-curve] using L = 98, expand_message_xmd with SHA-512, DST = "HashToScalar-" || contextString, and prime modulus equal to Order().
 - Serialization: Elements are serialized as Ne = 67 byte strings using compressed point encoding for the curve [SEC1]. Scalars are serialized as Ns = 66 byte strings by fully reducing the value modulo Order() and in big-endian order.
- * Hash: SHA-512, and Nh = 64.
- * ID: 0x0005

4.2. Input Validation

The `DeserializeElement` and `DeserializeScalar` functions instantiated for a particular prime-order group corresponding to a ciphersuite MUST adhere to the description in Section 2.1. This section describes how both `DeserializeElement` and `DeserializeScalar` are implemented for all prime-order groups included in the above ciphersuite list.

4.2.1. `DeserializeElement` Validation

The `DeserializeElement` function attempts to recover a group element from an arbitrary byte array. This function validates that the element is a proper member of the group and is not the identity element, and returns an error if either condition is not met.

For P-256, P-384, and P-521 ciphersuites, this function performs partial public-key validation as defined in Section 5.6.2.3.4 of [keyagreement]. This includes checking that the coordinates are in the correct range, that the point is on the curve, and that the point is not the point at infinity. If these checks fail, deserialization returns an error.

For ristretto255 and decaf448, elements are deserialized by invoking the `Decode` function from [RISTRETTO], Section 4.3.1 and [RISTRETTO], Section 5.3.1, respectively, which returns false if the input is invalid. If this function returns false, deserialization returns an error.

4.2.2. `DeserializeScalar` Validation

The `DeserializeScalar` function attempts to recover a scalar field element from an arbitrary byte array. Like `DeserializeElement`, this function validates that the element is a member of the scalar field and returns an error if this condition is not met.

For P-256, P-384, and P-521 ciphersuites, this function ensures that the input, when treated as a big-endian integer, is a value between 0 and `Order() - 1`. For ristretto255 and decaf448, this function ensures that the input, when treated as a little-endian integer, is a value between 0 and `Order() - 1`.

4.3. Future Ciphersuites

A critical requirement of implementing the prime-order group using elliptic curves is a method to instantiate the function `HashToGroup`, that maps inputs to group elements. In the elliptic curve setting, this deterministically maps inputs `x` (as byte arrays) to uniformly chosen points on the curve.

In the security proof of the construction `Hash` is modeled as a random oracle. This implies that any instantiation of `HashToGroup` must be pre-image and collision resistant. In Section 4 we give instantiations of this functionality based on the functions described in [I-D.irtf-cfrg-hash-to-curve]. Consequently, any OPRF implementation must adhere to the implementation and security considerations discussed in [I-D.irtf-cfrg-hash-to-curve] when instantiating the function.

Additionally, future ciphersuites must take care when choosing the security level of the group. See Section 6.2.3 for additional details.

5. Application Considerations

This section describes considerations for applications, including external interface recommendations, explicit error treatment, and public input representation for the POPRF protocol variant.

5.1. Input Limits

Application inputs, expressed as `PrivateInput` or `PublicInput` values, MUST be smaller than 2^{13} bytes in length. Applications that require longer inputs can use a cryptographic hash function to map these longer inputs to a fixed-length input that fits within the `PublicInput` or `PrivateInput` length bounds. Note that some cryptographic hash functions have input length restrictions themselves, but these limits are often large enough to not be a concern in practice. For example, SHA-256 has an input limit of 2^{61} bytes.

5.2. External Interface Recommendations

The protocol functions in Section 3.3 are specified in terms of prime-order group `Elements` and `Scalars`. However, applications can treat these as internal functions, and instead expose interfaces that operate in terms of wire format messages.

5.3. Error Considerations

Some OPRF variants specified in this document have fallible operations. For example, Finalize and Evaluate can fail if any element received from the peer fails deserialization. The explicit errors generated throughout this specification, along with the conditions that lead to each error, are as follows:

- * `VerifyError`: Verifiable OPRF proof verification failed; Section 3.3.2 and Section 3.3.3.
- * `DeserializeError`: Group Element or Scalar deserialization failure; Section 2.1 and Section 3.3.

There are other explicit errors generated in this specification, however they occur with negligible probability in practice. We note them here for completeness.

- * `InvalidInputError`: OPRF Blind input produces an invalid output element; Section 3.3.1 and Section 3.3.3.
- * `InverseError`: A tweaked private key is invalid (has no multiplicative inverse); Section 2.1 and Section 3.3.

In general, the errors in this document are meant as a guide to implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, implementations might run out of memory and return a corresponding error.

5.4. POPRF Public Input

Functionally, the VOPRF and POPRF variants differ in that the POPRF variant admits public input, whereas the VOPRF variant does not. Public input allows clients and servers to cryptographically bind additional data to the POPRF output. A POPRF with fixed public input is functionally equivalent to a VOPRF. However, there are differences in the underlying security assumptions made about each variant; see Section 6.2 for more details.

This public input is known to both parties at the start of the protocol. It is RECOMMENDED that this public input be constructed with some type of higher-level domain separation to avoid cross protocol attacks or related issues. For example, protocols using this construction might ensure that the public input uses a unique, prefix-free encoding. See [I-D.irtf-cfrg-hash-to-curve], Section 10.4 for further discussion on constructing domain separation values.

Implementations of the POPRF may choose to not let applications control info in cases where this value is fixed or otherwise not useful to the application. In this case, the resulting protocol is functionally equivalent to the VOPRF, which does not admit public input.

6. Security Considerations

This section discusses the cryptographic security of our protocol, along with some suggestions and trade-offs that arise from the implementation of the OPRF variants in this document. Note that the syntax of the POPRF variant is different from that of the OPRF and VOPRF variants since it admits an additional public input, but the same security considerations apply.

6.1. Security Properties

The security properties of an OPRF protocol with functionality $y = F(k, x)$ include those of a standard PRF. Specifically:

- * Pseudorandomness: F is pseudorandom if the output $y = F(k, x)$ on any input x is indistinguishable from uniformly sampling any element in F 's range, for a random sampling of k .

In other words, consider an adversary that picks inputs x from the domain of F and evaluates F on (k, x) (without knowledge of randomly sampled k). Then the output distribution $F(k, x)$ is indistinguishable from the output distribution of a randomly chosen function with the same domain and range.

A consequence of showing that a function is pseudorandom, is that it is necessarily non-malleable (i.e. we cannot compute a new evaluation of F from an existing evaluation). A genuinely random function will be non-malleable with high probability, and so a pseudorandom function must be non-malleable to maintain indistinguishability.

- * Unconditional input secrecy: The server does not learn anything about the client input x , even with unbounded computation.

In other words, an attacker with infinite compute cannot recover any information about the client's private input x from an invocation of the protocol.

Additionally, for the VOPRF and POPRF protocol variants, there is an additional security property:

- * **Verifiable:** The client must only complete execution of the protocol if it can successfully assert that the POPRF output it computes is correct. This is taken with respect to the POPRF key held by the server.

Any VOPRF or POPRF that satisfies the 'verifiable' security property is known as 'verifiable'. In practice, the notion of verifiability requires that the server commits to the key before the actual protocol execution takes place. Then the client verifies that the server has used the key in the protocol using this commitment. In the following, we may also refer to this commitment as a public key.

Finally, the POPRF variant also has the following security property:

- * **Partial obliviousness:** The server must learn nothing about the client's private input or the output of the function. In addition, the client must learn nothing about the server's private key. Both client and server learn the public input (info).

Essentially, partial obliviousness tells us that, even if the server learns the client's private input x at some point in the future, then the server will not be able to link any particular POPRF evaluation to x . This property is also known as unlinkability [DGSTV18].

6.2. Security Assumptions

Below, we discuss the cryptographic security of each protocol variant from Section 3, relative to the necessary cryptographic assumptions that need to be made.

6.2.1. OPRF and VOPRF Assumptions

The OPRF and VOPRF protocol variants in this document are based on [JKK14]. In fact, the VOPRF construction is identical to the [JKK14] construction, except that this document supports batching so that multiple evaluations can happen at once whilst only constructing one proof object. This is enabled using an established batching technique.

The pseudorandomness and input secrecy (and verifiability) of the OPRF (and VOPRF) variants is based on the assumption that the One-More Gap Computational Diffie Hellman (CDH) is computationally difficult to solve in the corresponding prime-order group. The original paper [JKK14] gives a security proof that the construction satisfies the security guarantees of a VOPRF protocol Section 6.1 under the One-More Gap CDH assumption in the universal composability (UC) security framework.

6.2.2. POPRF Assumptions

The POPRF construction in this document is based on the construction known as 3HashSDHI given by [TCRSTW21]. The construction is identical to 3HashSDHI, except that this design can optionally perform multiple POPRF evaluations in one go, whilst only constructing one NIZK proof object. This is enabled using an established batching technique.

Pseudorandomness, input secrecy, verifiability, and partial obliviousness of the POPRF variant is based on the assumption that the One-More Gap Strong Diffie-Hellman Inversion (SDHI) assumption from [TCRSTW21] is computationally difficult to solve in the corresponding prime-order group. [TCRSTW21] show that both the One-More Gap CDH assumption and the One-More Gap SDHI assumption reduce to the q-DL (Discrete Log) assumption in the algebraic group model, for some q number of Evaluate queries. (The One-More Gap CDH assumption was the hardness assumption used to evaluate the OPRF and VOPRF designs based on [JKK14], which is a predecessor to the POPRF variant in Section 3.3.3.)

6.2.3. Static Diffie Hellman Attack and Security Limits

A side-effect of the OPRF protocol variants in this document is that they allow instantiation of an oracle for constructing static DH samples; see [BG04] and [Cheon06]. These attacks are meant to recover (bits of) the server private key. Best-known attacks reduce the security of the prime-order group instantiation by $\log_2(Q)/2$ bits, where Q is the number of Evaluate() calls made by the attacker.

As a result of this class of attack, choosing prime-order groups with a 128-bit security level instantiates an OPRF with a reduced security level of $128 - (\log_2(Q)/2)$ bits of security. Moreover, such attacks are only possible for those certain applications where the adversary can query the OPRF directly. Applications can mitigate against this problem in a variety of ways, e.g., by rate-limiting client queries to Evaluate() or by rotating private keys. In applications where such an oracle is not made available this security loss does not apply.

In most cases, it would require an informed and persistent attacker to launch a highly expensive attack to reduce security to anything much below 100 bits of security. Applications that admit the aforementioned oracle functionality, and that cannot tolerate discrete logarithm security of lower than 128 bits, are RECOMMENDED to choose groups that target a higher security level, such as decaf448 (used by ciphersuite 0x0002), P-384 (used by 0x0004), or P-521 (used by 0x0005).

6.3. Domain Separation

Applications SHOULD construct input to the protocol to provide domain separation. Any system which has multiple OPRF applications should distinguish client inputs to ensure the OPRF results are separate. Guidance for constructing info can be found in [I-D.irtf-cfrg-hash-to-curve], Section 3.1.

6.4. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST run in constant time. This includes all prime-order group operations and proof-specific operations that operate on secret data, including GenerateProof() and Evaluate().

7. Acknowledgements

This document resulted from the work of the Privacy Pass team [PrivacyPass]. The authors would also like to acknowledge helpful conversations with Hugo Krawczyk. Eli-Shaoul Khedouri provided additional review and comments on key consistency. Daniel Bourdrez, Tatiana Bradley, Sofia Celi, Frank Denis, Kevin Lewi, Christopher Patton, and Bas Westerbaan also provided helpful input and contributions to the document.

8. References

8.1. Normative References

[I-D.ietf-privacypass-protocol]
Celi, S., Davidson, A., Faz-Hernandez, A., Valdez, S., and C. A. Wood, "Privacy Pass Issuance Protocol", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-02, 31 January 2022, <<https://www.ietf.org/archive/id/draft-ietf-privacypass-protocol-02.txt>>.

[I-D.irtf-cfrg-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-13, 10 November 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-hash-to-curve-13.txt>>.

- [I-D.irtf-cfrg-opaque]
Bourdrez, D., Krawczyk, H., Lewi, K., and C. A. Wood, "The OPAQUE Asymmetric PAKE Protocol", Work in Progress, Internet-Draft, draft-irtf-cfrg-opaque-07, 25 October 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-opaque-07.txt>>.
- [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/info/rfc2119>>.
- [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/info/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/info/rfc8174>>.
- [RISTRETTO]
Valence, H. D., Grigg, J., Tankersley, G., Valsorda, F., Lovecruft, I., and M. Hamburg, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-01, 4 August 2021, <<https://www.ietf.org/archive/id/draft-irtf-cfrg-ristretto255-decaf448-01.txt>>.

8.2. Informative References

- [BG04] "The Static Diffie-Hellman Problem", <<https://eprint.iacr.org/2004/306>>.
- [ChaumPedersen]
"Wallet Databases with Observers", n.d., <https://chaum.com/publications/Wallet_Databases.pdf>.
- [Cheon06] "Security Analysis of the Strong Diffie-Hellman Problem", <<https://www.iacr.org/archive/eurocrypt2006/40040001/40040001.pdf>>.
- [DGSTV18] "Privacy Pass, Bypassing Internet Challenges Anonymously", <<https://www.degruyter.com/view/j/popets.2018.2018.issue-3/popets-2018-0026/popets-2018-0026.xml>>.

- [JKK14] "Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only model", <<https://eprint.iacr.org/2014/650>>.
- [JKKX16] "Highly-Efficient and Composable Password-Protected Secret Sharing (Or, How to Protect Your Bitcoin Wallet Online)", <<https://eprint.iacr.org/2016/144>>.
- [keyagreement] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", DOI 10.6028/nist.sp.800-56ar3, National Institute of Standards and Technology report, April 2018, <<https://doi.org/10.6028/nist.sp.800-56ar3>>.
- [PrivacyPass] "Privacy Pass", <<https://github.com/privacypass/challenge-bypass-server>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [SEC1] Standards for Efficient Cryptography Group (SECG), .. "SEC 1: Elliptic Curve Cryptography", <<https://www.secg.org/sec1-v2.pdf>>.
- [SEC2] Standards for Efficient Cryptography Group (SECG), .. "SEC 2: Recommended Elliptic Curve Domain Parameters", <<http://www.secg.org/sec2-v2.pdf>>.
- [SJKS17] "SPHINX, A Password Store that Perfectly Hides from Itself", <<https://eprint.iacr.org/2018/695>>.
- [TCRSTW21] "A Fast and Simple Partially Oblivious PRF, with Applications", <<https://eprint.iacr.org/2021/864>>.
- [x9.62] ANSI, "Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62-1998, September 1998.

Appendix A. Test Vectors

This section includes test vectors for the protocol variants specified in this document. For each ciphersuite specified in Section 4, there is a set of test vectors for the protocol when run the OPRF, VOPRF, and POPRF modes. Each test vector lists the batch size for the evaluation. Each test vector value is encoded as a hexadecimal byte string. The label for each test vector value is described below.

- * "Input": The private client input, an opaque byte string.
- * "Info": The public info, an opaque byte string. Only present for POPRF vectors.
- * "Blind": The blind value output by Blind(), a serialized Scalar of N_s bytes long.
- * "BlindedElement": The blinded value output by Blind(), a serialized Element of N_e bytes long.
- * "EvaluatedElement": The evaluated element output by Evaluate(), a serialized Element of N_e bytes long.
- * "Proof": The serialized Proof output from GenerateProof() (only listed for verifiable mode test vectors), composed of two serialized Scalar values each of N_s bytes long. Only present for VOPRF and POPRF vectors.
- * "ProofRandomScalar": The random scalar r computed in GenerateProof() (only listed for verifiable mode test vectors), a serialized Scalar of N_s bytes long. Only present for VOPRF and POPRF vectors.
- * "Output": The OPRF output, a byte string of length N_h bytes.

Test vectors with batch size $B > 1$ have inputs separated by a comma ",". Applicable test vectors will have B different values for the "Input", "Blind", "BlindedElement", "EvaluationElement", and "Output" fields.

The server key material, pk_{Sm} and sk_{Sm} , are listed under the mode for each ciphersuite. Both pk_{Sm} and sk_{Sm} are the serialized values of pk_S and sk_S , respectively, as used in the protocol. Each key pair is derived from a seed $Seed$ and info string $KeyInfo$, which are listed as well, using the DeriveKeyPair function from Section 3.2.

A.1. OPRF(ristretto255, SHA-512)

A.1.1. OPRF Mode

[illegible]

A.1.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = c604c785ada70d77a5256ae21767de8c3304115237d262134f5e46e512cf
8e03
BlindedElement = 8453ce4f98478a73faf24dd0c2e81d9a5e399171d2687cc258b
9e593623bde4d
EvaluationElement = 22bcfc0930ecddf4ada3f0cb421c8d6669576fc4fbbe24e1
8c94d0f36e767466
Output = 2765a7f9fa7e9d5440bbf1262dc1041277bed5f27fd27ee89662192a408
508bb8711559d5a5390560065b83b946ed7b433d0c1df09bd23871804ae78e4a4d21
5
```

A.1.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = 5ed895206bfc53316d307b23e46ecc6623afb3086da74189a416012be037  
e50b  
BlindedElement = 86ef8baa01dd6cc34a067d2fc56cde51498a54cb0c30f63f083  
53d912164d711  
EvaluationElement = a27d5e498927ca96e493373a04e263115c31b918411df0ce  
d382db4e66388766  
Output = 3d6c9ec7dd6f51b987b46b79128d98323accd7c1561faa50d287c5285ec  
dale660f3ee2929aebd431a7a7d511767cbd1054735a6e19aeelb9423ale6f479535  
e
```

A.1.2. VOPRF Mode

```
Seed = a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3a3  
KeyInfo = 74657374206b6579  
skSm = 24d1aca670a768b99e888c19f9ee3252d17e32689507b2b9b803ae23b1997  
f07  
pkSm = 46919d396c12fbb7a02f4ce8f51a9941ddc1c4335682e1b03b0ca5b3524c6  
619
```

A.1.2.1. Test Vector 1, Batch Size 1

```

Input = 00
Blind = ed8366feb6b1d05d1f46acb727061e43aadfafe9c10e5a64e7518d63e326
3503
BlindedElement = 444550ea064013c569fe63567eb93e7a9496902a573ea1e6654
76fd39d5edc40
EvaluationElement = 7af7a45e4f1e0c6d410d41704e16d980ebff051fd0975fce
cd17f79a6b57a473
Proof = 26982a26b2aa20f1e449be5a858c59d7992f7f4a13b007e3980f5c36e8ae
a7014268883db3094e08e3f493b3d23bae87ac098a33e775172c1027f1b5d025ca08
ProofRandomScalar = 019cbd1d7420292528f8cdd62f339fdabb602f04a95dac9d
bcec831b8c681a09
Output = 453a358544b4e92bbc4625d08ffdde64c0dbc4f9b1501d548e3a6d8094b
a70a993c13a6e65a46880bbd65272ba54cf199577760815098e5e10cb951b1fc5b02
7

```

A.1.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = e6d0f1d89ad552e383d6c6f4e8598cc3037d6e274d22da3089e7afbdc4171ea02  
BlindedElement = 82d9fc20daf67106ae2d2c584c615d103dafda653ac5b2b2c6faafcf06e3f1c0a  
EvaluationElement = 60c468920f4f949be9aaaf9b4fb27dc7bc89daca4a3aaa31e96efae56c02ac75  
Proof = 4a7490fd0a9e13cc66bcdeded002899a3e206364d9bdbaf9998a73dd728c8602a6967f81a4948e6de797d638ee02ca44d933d05f2715fa1618b6a3324f3b2608  
ProofRandomScalar = 74ae06fd50d5f26c2519bd7b184f45dd3ef2cb50197d42df9d013f7d6c312a0b  
Output = a2bacfc82a4cac041edab1e1cd0dc63f46631fb4886f8c395f0b184a9b7cbbef2eee05bbdb3f085552d8c80e77711b2ad9ba2b7574e2531591380e717d29c6f5
```

A.1.2.3. Test Vector 3, Batch Size 2

[illegible]

A.1.3. POPRF Mode

[illegible]

A.1.3.1. Test Vector 1, Batch Size 1

```

Input = 00
Info = 7465737420696e666f
Blind = 7e5bcbf82a46109ee0d24e9bcab41fc830a6ce8b82fc1e9213a043b743b9
5800
BlindedElement = da01485047605a666542d0599ef2fbeed0c2e45a97c6e3d420f
832918e09f535
EvaluationElement = 3015fc16fe179bdb9054da5297c77d1f249dabf32e4fdcc4
937d6ba5e99d7b53
Proof = f10470180fc884a2f51472eddde9ad9a4080b00e13f63c130cece83b93ca
500f956b08e35ed2670ca504c704e0b74687451f5985627c93e2290a5da0dfc1d0b
ProofRandomScalar = 080d0a4d352de92672ab709b1ae1888cb48dfabc2d6ca5b9
14b335512fe70508
Output = 4d04eccb77a29bd8a00fb1e3f391e0601340c3dc874fc7bb16cfd92d961
532dl8b4edfffaec94457cb19111bca1ecd19e46124c6a5d29703d09df5e5ab521b2
8

```

A.1.3.2. Test Vector 2, Batch Size 1

```

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = de2e98f422bf7b99be19f7da7cac62f1599d35a225ec6340149a0aaff310
2003
BlindedElement = 909f8d2d517fa2235f8b35f91220636732541d9f3e309c6988d
6d8c987e5a357
EvaluationElement = 241786b8f9da3e8c28d75dc23b5f8b251ec150ccb453efa7
12f6e9b72e763a0a
Proof = a3748b980aec81add561bcd7ac4fe2b09a93bd8a127991788fd618bf7fb7
93034a6f7f59cdcab538ed3e50d74b31f82dff14e3c8d3a081f744a6bdf93526ed0e
ProofRandomScalar = c4d002aa4cfcf281657cf36fe562bc60d9133e0e72a74432
f685b2b6a4b42a0c
Output = a88ab2bceba2c9c5a0ee0ee45636e65042b5f274af864f8c1560d32ecee
4373c31907f237609d3f164beec32e3270588961c1d19cee467d2a3b0445ebdea215
9

```

A.1.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Info = 7465737420696e666f  
Blind = e79a642b20f4c9118febffaef6b6a31471fe7794aa77ced123f07e56cb8cf  
7c01,0bb106c0e1aac79e92dd2d051e90efe4e2e093bc1e82b80e8cce6afa4f51980  
2  
BlindedElement = 3206271954cce85425971fddfefebe14acad819b9753ffc171815  
7e54a5e56542f,847b21e32855892256a3eee10ea5c512d362b34de1ab278573cf91  
edfcb14a03  
EvaluationElement = 76d3282ac9aabc9b0133df89e680ab0d43f2946c224db25e  
798abdff0ed1d255a,e8483fbacb3e62787a803dd6d688e4db26be5392f529f1dd6a7  
f06e2b28dc52c  
Proof = f5b8d39897f12ddl1f8fc927e2f7f563629b7b45f1e6b5eeb469c043d2143  
7907a0e9236beec240a04e0fb906a7d126a8cb40e22730106446c1fa3a40a5283406  
ProofRandomScalar = 668b3aab5207735beb86c5379228da260159dc24f7c5c248  
3a81aff8fbffcc0d  
Output = 4d04eccb77a29bd8a00fb1e3f391e0601340c3dc874fc7bb16cfd92d961  
532d18b4edfffaec94457cb19111bca1ecd19e46124c6a5d29703d09df5e5ab521b2  
8,a88ab2bceba2c9c5a0ee0ee45636e65042b5f274af864f8c1560d32eceee4373c31  
907f237609d3f164beec32e3270588961c1d19cee467d2a3b0445ebdea2159
```

A.2. OPRF (decaf448, SHAKE-256)

A.2.1. OPRF Mode

[illegible]

A.2.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = d62851d4bc07947c858dc735e9e22aaf0576f161ab555182908dbd7947b1
c988956fa73b17b373b72fd4e3c0264a26aa4cab20fd6193b933
BlindedElement = d078a185d2d8a54b68d6df4e83640192d3659e18fec68d43e48
02998d3c9fd819b32070caa78083c909d68daeb7fd420a73f931452a2b70d
EvaluationElement = 3452e46b6277b032627a7e5d22aa1b25459f8de90dda3137
9ed490bb0078eeec05fc4265fafbb5252d4228f9f1f5453bbd391d6b8589f232
Output = b93d3ed18489c1236cc965d202254de35767ea673560d6c225cec0b30fe
3adc88fee63f8a78d127cd64c7077e1d3ac4a7cc761335c0bcdcd12d6981ad8730285
8
```

A.2.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = ac345e8d755997956ddd1f267a2d86175aeae5e1168932285a6f602b4b20  
a570a697452b3ddb7d0e29363adebbcb5673294396b82931f37  
BlindedElement = 283f0fab2be6ac3a3c8eacf504f3ef63f518892f7b000f1dcc  
1ca2e773aba0fbec48b100886b90d5a08377cbf5ccf69801ae2c23e1adbf2  
EvaluationElement = ae30bab51a34c45a76d00034b29e1c5346fbe3718c386302  
8e47226456880a85a2e5118f274a8c260dae62fcec3cde8624405fc7cddbc867  
Output = aaf99e5a044bbce915bf3ba381e25da62e4b2cea4cee2f47f3662940284  
579cf0f8e1e011062ba010ca4f2c67a8157481c9ae7a458ea035a89e1948bffc5b8323  
b
```

A.2.2. VOPRF Mode

[illegible]

A.2.2.1. Test Vector 1, Batch Size 1

```

Input = 00
Blind = 4bdfc97a75132d92a1da241baff84fada3e7b12d5b712efcac9ba734d54c
2b24bfff0ef6310404b5c05d60d7c258cea6500229ee015149f0f
BlindedElement = 1ceb0a3432ac6b583c31fa70b7c17ac86e0aa425e0593d04b58
021670f725eee6664e6cd2041d90f157bc213a2aa4ed7929630b2d9898a76
EvaluationElement = 3afaa02425294a4810766c68e9e4c3c507b109b9064ed56a
148a419371d5fb158f6ab5f0da62a6ba915bbe431097f5c71854821c1f10889d
Proof = f02f7ab2722508e343b5692078556e7ca9b2d63bf83dff902150b867775b
f375693cc6a0adf33178ba7e72d6179b36ed051065c93619752958746f0d52e2e3a9
89d86df15f458847abdcc23976147b7b10c96452332aa03bfce1b89b7aeed080869d
7ce8c7acb7414e7dbfcda298b532
ProofRandomScalar = 54534add9db9f6df6ce515d1b8017923b65cada199e936a62
3c8eb3bd08e9b3f6584a85e4ff26e9f869d30b6c7c6cc56fd94e306974fbcc3b
Output = b558e37f6435a12fefded196936a4c1d0882bfb4a115002920744ecb3128
43678f396f7d36711cf551750388ddf7a53a3aea7fd0ac60568cd2d4ead16alee106
f

```

A.2.2.2. Test Vector 2, Batch Size 1

```

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = beda1edc786e5fd0033feac1992c53a607d516a46251614940e76a2763b8
0683e5b789398710bdbc774d9221dd33c509b4805fc26f0c8d0b
BlindedElement = 2e04b6883057a5b5ba020d077ae36dee76a07c2f3eb8cc55baf
dfb3da9c7405ffe50802f646ca3c3ef39d195c2d88ee56e73825c7cd2319a
EvaluationElement = 6270b2f73738aa846dca34d7b30b7c4f943e31d4d4fb35c5
98f5d608cf25648b44553d43b158dc2707eda170dc439740c10d7b4355bf0f83
Proof = f731f60aa18d508f07dd3b7851fe9f8cfe6f02c4ea2814cfe8af3203e493
44041e6acf0f09fdffdc02d22728544b9bda8d0604e727f27a1efa16526f169191de
db35a1338bf399d8737d6d1638f6d4b895c0869b4194e66fb0dbb4b3e0437a2af0d7
6dd8c9b0bf38c9de605dc5749603
ProofRandomScalar = 00cc800042a0cff31f865698f8858efa75a1f0faef934317
dd6a10bfbbb39f9f2d97dcd5ff4eae02980b08fc68da7b71d39399dc4eb0400a
Output = eb14608be2f14c25b2c9fdd23690d293d0c6aaac501a3405b626b8699cf
34bb9dd4c2d7987b6391519b9480da453611509ba98098b3e79a35acd00f5e9d8abc
e

```

A.2.2.3. Test Vector 3, Batch Size 2

[illegible]

A.2.3. POPRF Mode

[illegible]

A.2.3.1. Test Vector 1, Batch Size 1

```

Input = 00
Info = 7465737420696e666f
Blind = ee671e4c9b6783bd5e4a55d2e8474fe0ec811b4cca7c0e51a886c4343d83
c4e5228b87399f1dbf033ee131fe52bae62a0cb27eb7abfcab24
BlindedElement = 4c371528ab436b8a6a5bea333cc5702c70cdddb80d12dc2eafa
06b87c15bba8b0b5451bc09f3d07e57c12af4c0398b09ae91b678fdeaf2aa
EvaluationElement = d27f65d6c41880303989752e40748e940add1ad32e7f76cc
bb873b7ffff424d348ec8e43c11402e02934c1fcdadeacbca2d2e5171daaeef90
Proof = bf2f61413c56c0351151c1995007ceb2e197c987056f20a54f0027e544a0
b20a7891b9aa882203f2e09e1a0ca9464e3cdf130eea9e1123023460d3f280dac87d
23b8d2258666d002f57810d8847832b775984819e457c7bbe703947e7aeccdf59d3e
520437edefc26b814f9fa7fa9917
ProofRandomScalar = c4b297c662a87631531aade91c0558d87224d92247bdfa11
9a53af4cbdb352b0a2016e5e5f6c0bee4a642526ef9910289315b71fdee5df1e
Output = 1ffbfb9591b674e6a089279a8319c75e949cc277d7b5c757361412180307
90755e90af009768e1b9240c9734d8886c6121123384140b26c38c7a6c4217a1b3d9
4

```

A.2.3.2. Test Vector 2, Batch Size 1

```

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 1abe4937f28f531b14ac96b844320e7a66810c2d9391cbb877348301ab59
a3a91b4a2129672886ae5da7839f2ac8cf1c5fa92703f5b3fd06
BlindedElement = dea615b00285247715173fc6db40cab1436607bc0eae3d7a1a
1467b70c7ff2f2ce91c05bcaeda2b01952926f254f13e1a763a174caa693a
EvaluationElement = a692017b9e91efbe6641c3ef0cd3b352022ed08bb5ed0c1d
a0838589bebe53c2d2959818359cb0213b94cafa672673608b9a2280671d6c75
Proof = ede122b8ce87d22fa1bb9dd38dc76da1a9ff812a8d2cbf3d2a6e86a10331
a849d203bd925d6f130d80f333aa0443488731769e975b4c900d923d740fec13a61d
3175a0daf9a88d8f66704b36ca2b1b7fef6dcab4ffb50fe998e53ce4743ee9466a56
886f79fd6d5b924553f64130c60a
ProofRandomScalar = a3e896e126d371f6380ca41757f6458b93b049e1b0d73ab5
b8d914b08dff3e52e62ea8898d35b2862d28ff4c5f89353d25d6b5a8dc014d3b
Output = daeb206a0e1fc120ebe4ad885f851f456f7d8908166839b7dc541f71251
4203d9a3589025b4bfad6a79c6d40bfbf217f44a9aa17874a1ec271b23ccd72a44e
f

```

A.2.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Info = 7465737420696e666f  
Blind = 255d8adc40b8f39f14cd8bd4ade8abbb95166afdc9e922203abe7a853985  
4c64b943b0b46e1e1b47cfb52e9a0867c8cde22bcbdd724d9f09,71bd897c56c86b3  
1b096103b7e2d26d0f4d66be95299379b41668dbbc5ece26cc212d9f2cbfaf479efa  
17b7f6b056dfcfbee5bd7365cea26  
BlindedElement = 361b80bba04ff4b211e38e636a8530531213a44f44738992b18  
eaf0d9759eedcb7e4034e9bda6f8829250aff72343b0d2d1e23d612d94674,c68d79  
dl4614b90e6abl1dc14a982f9fdd423edc94a10d87d45e32935e363079967ad289482  
8b1764cca8dec5e9f919def474b1d03b6c069d  
EvaluationElement = 32629ccfd36787d8d80756f025f6c23c21145dd22c28d974  
f34098e166a300731b691el1faa7e3959c1bb38312c43d1d693cbab4b90fe7d2e,eec  
655ala869b3f0f470f7a0f2cef69eb6539c6c1b9e49d9b380bccc7b510d466f45d88  
fa690b687a8507d1e0b275028d095292fc4aadd2d  
Proof = 2f3501925c81837232ae34e5351518ad35e24f1d32f7459da3c19cae7746  
95e7dc1eca32133dd57cd0e2eb67c75c9edd9cd3ff9c5e1759314ea99a4eca322f6e  
56f4b80795f67d1bf747834d2d7b3049351979ca876ecf28f87b81fba243269e3c09  
ea1889abd968af67c7ca511d0c3d  
ProofRandomScalar = bbbflebe98b192e93cedceb9c0164e95b891bd8bc81721b8  
ea31835d6f9687a36c94592ab76579f42ce1be6961f0700496e71df8c17ab50c  
Output = 1ffbfb951b674e6a089279a8319c75e949cc277d7b5c757361412180307  
90755e90af009768e1b9240c9734d8886c6121123384140b26c38c7a6c4217alb3d9  
4, daeb206a0e1fc120ebe4ad885f851f456f7d8908166839b7dc541f712514203d9a  
3589025b4bfad6a79c6d40bfbf217f44a9aa17874alec271b23ccd72a44ef
```

A.3. OPRF (P-256, SHA-256)

A.3.1. OPRF Mode

[illegible]

A.3.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = f70cf205f782fa11a0d61b2f5a8a2a1143368327f3077c68a1545e9aafbba6aa
BlindedElement = 0372ffe1ebd9273f17b09916d31e7884707e8902f7e3af2a1b3aeldfbfae9b5126
EvaluationElement = 02aa5b346b0375cd734014ffa9ed2135a1b07565c44fe64d5accfe6ab6d8c37f77
Output = 413c5d45657ce515914232ef0bafdbcb1bfa5c272d4b403f2cea0ccf7ca18f9be
```

A.3.1.2. Test Vector 2, Batch Size 1

[illegible]

A.3.2. VOPRF Mode

[illegible]

A.3.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = e74c5078a81806f74dd65065273c5bd886c7f87ff8c5f39f90320718eff7
47e3
BlindedElement = 029d750421c5c726658902c47d3675ebba01ba25d0bd127bf6e
338b801b166f1d2
EvaluationElement = 0291e9890c7418a2fc1ac635d2650bae3f1a25a9ffcd0bc0
1b3c39fcee4b095dca
Proof = 54ec2d8558f5c72ff32489556c3ba1f3087810c5f51cc025f07adc034df2
dcd6d706e7bdae3119b70748cbf76b66d520de87bf90287a091cf6f8d2a465cf2200
ProofRandomScalar = dfc19eb96faba6382ec845097904db87240b9dd47b1e487e
c625f11a7ba2cc3e
Output = a906579bce2c9123e5a105d4bdbcafb513d7d764e4f0937bee95b362527
78424
```

A.3.2.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = dfe89ac0cdd6b74684580de0f95f8e06aa6f6663d48b1a4b998a539380ed  
73cb  
BlindedElement = 02e6085d4017ae0bede4b261977b588349d323414eb5c409e55  
2e2bd4c82df498b  
EvaluationElement = 03a649c5ac48f33a6c6cd82120145e673e17395ca94ea824  
c7d2dda7203ba4159a  
Proof = 1a79f6a52579f7acb0100c916390989a1dca3c1b3078402e102b8dd037f0  
b34d929d38239b34175f1328708ec197bfcc532ef31dafdfdlee85db4ccf8769844fdb  
ProofRandomScalar = 4f9a70536c175f11a827452672b60d4e9f89eba281046e28  
39dd2c7a98309b07  
Output = d13c62d285a71acb534dcebdf312bfec0e2a3fcfb79f4ac32d2dfb0bc9aa  
e3cc7
```

A.3.2.3. Test Vector 3, Batch Size 2

[illegible]

A.3.3. POPRF Mode

[illegible]

A.3.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 4238835743037876080d2e3e27bc3ce7b5fb6a1107ffedeaedb371767432
b68c
BlindedElement = 02cb57f07ba100b93ce1bf8176963c8c7f73a76827f1c1401a9
23d7ca4083e15aa
EvaluationElement = 03059d58ec9a801e33f57525c03241d8fffb61b67a18edd35
222d864ffbb42b5d2f
Proof = 13889d6849850ccd0119981fc053a38a30a57d275091df2887943d1332f7
38204f8a6cf2fb6e57c9b118ec82b9b012f8864561e4cd8866245f9c762b9d45dbf9
ProofRandomScalar = 3d5c65b55a1b8960563b3420d7764097502850c445ccd86e
2d20d7e4ec77617b
Output = 15fce9922a2307349aac2eccc41941283e3c5e938aaf2506f99a6d8b6ee
34ef8
```

A.3.3.2. Test Vector 2, Batch Size 1

[illegible]

A.3.3.3. Test Vector 3, Batch Size 2

[illegible]

A.4. OPRF (P-384, SHA-384)

A.4.1. OPRF Mode

[illegible]

A.4.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = cda63dff3137c959747ec1d27852fce42d79fc710159f349e7da18455479
e27473269d2926fec54d4567adabd7951ad6
BlindedElement = 020db1b05b22ddd8a851792dfb5b10b4f237d69522097cbb012
7ae537e3256f86e35a72554a6ebdb26c28342fee16473dd
EvaluationElement = 031e6e8c82d3284727a724a5854b3e2bf9958b4e5470601f
4ca37d33d26879eca817796cb7e98bbbbb1d1739eeafb33c027
Output = b2e380ca96ea80f7550a6b663e5f7752d7d7772c46169d72308a8425903
1e804ba577ac34e632f535a9519a692734016
```

A.4.1.2. Test Vector 2, Batch Size 1

[illegible]

A.4.2. VOPRF Mode

[illegible]

A.4.2.1. Test Vector 1, Batch Size 1

```

Input = 00
Blind = 61247a74d0c62c98ddff1365bb9b82b279e775b7220c673c782e351691be
a8206a6b6856c044df390ab5683964fc7aac
BlindedElement = 026601d99c313b827a09aad832fcc814ac5257a57bb49d65c05
e247df9518315a66557fc8af56b4521c51900aaab1a2ea9
EvaluationElement = 020b478b9c9b1a5935e07fb532eac2e596b78170a0e755ec
c71829419e63a2119eae23be281e109de205cd85af7e42228a
Proof = 02d0946f1795048bc803171aea5b4a9a5f256bd5fa9414e5fa76dd17a4aa
a94307814d57c2cca239485e29bb76d4ac1b4d3d62dfbb8e43c7135b2ebe50fe923e
30bd99e1e6ec961db18fa6e67c63dd6652284c15860156c08d64d838efbeeb68
ProofRandomScalar = f5685928c72d9dab8ddfe45de734ce0d4ff5823d2e40c4fc
f880e9a8272b46eea593b1095e7d38ba6ff37c42b3c48598
Output = f18884ace2b342f849cea7f2f17de902b9884574fdaa8f507356f482c6b
67013f329e8c899b3c2c154af1defaa11d656

```

A.4.2.2. Test Vector 2, Batch Size 1


```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = ef54a703503046d8272eaea47cfa963b696f07af04cbc6545ca16de56540  
574e2bc92534ac475d6a3649f3e9cdf20a7f  
BlindedElement = 0279e61686e698fdbac5cf484f54846db8cdf6f403fa88209c3  
4c56c584fe4ca600ac81b61aad1lc5e639ffladd3b30de4  
EvaluationElement = 03900e8e3f5b8bf698e7aa0aacb8dbdfaedf80220b1f640d  
e2049615985b19b913569cc2feb90725a3661146fb88ef3755  
Proof = 343536346a1145b81336eafc239f225dc6a154752492707c0465f029aa9a  
f0fe2bf0428285e43b596db633b50f0801b62b0e9c64c62f329b8a84324a415e4a58  
6cbf9477b1285c9b74f614c352e06658a8997486b8177006491e84aa96a3de09  
ProofRandomScalar = 0cdd9475ad6d9e630235fff21b634bc650bf837aaa273530d  
c66aa53bb9adb4f0ed499871eb81ae8claf769a56d4fc42b  
Output = f91d172cdcedea4f8299c8b39426db4c47428b82f8872b8539ad9b019de  
b48b8d3c928c572ed988d5591a4442c060438
```

A.4.2.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Blind = 485cccf5018abbf875b8e81c5ade0def4fe6fa8dfc15388367a60f23616cd1468dae601875f7dd570624d0ae9d7be2e7,b0d53a6f8da29c3cf4f8695135d645424c747bec642bc91375ff142da4687426b0b4f35c14eb2477c52e1ffe177f193b  
BlindedElement = 03dbcb21b211e7b5d2cf0c36d782308af28458539423f67a29336355e55035137eb768b1935b5a825c589a2913f0c2894e,036c8b2fa4dd9cd057561d377b4686cddc82317ad3e5eda08bece2a8616ca724937fff933e340a47fc09bf9b0fc1ef9ab6  
EvaluationElement = 03a1cba477a408162aacdcac43e059309fd61cc14687a107bd492a1ec688a010ffa49c60684e0f973412a7da2e627b1553a5,036fe6df8a99bb7bd4a5020ab4c6d7e71b5abc2d5d5a418f2314b614deb40c7b3acad982951b5f524e56f0e9ac7d8e95  
Proof = 1e26bf1210717b88dafae585008100e9ccaabe93b8605ca168a608cbf1855697b7b87d0b9c6bdca85e43143b3630e87f2fe9ce519dba3d477d2a869bcd0db9dc6239cd11938213f9bfd63d39de090a6fc90cd1f33f164b2c54c38bc31ad98ddf  
ProofRandomScalar = b36f4c2a140b7a3c53dd8efb6171d3bb4d73591be8483a1a38e40c13a04b0f2180dda3c36e3d43c3a8f127158d010945  
Output = f18884ace2e342f849cea7f2f17de902b9884574fdaa8f507356f482c6b67013f329e8c899b3c2c154af1defaa11d656,f91d172cdcedea4f8299c8b39426db4c47428b82f8872b8539ad9b019deb48b8d3c928c572ed988d5591a4442c060438
```

A.4.3. POPRF Mode

[illegible]

A.4.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 9572d3a8a106f875023c9722b2de94efaa02c8e46a9e48f3e2ee00241f9a
75f3f7493200a8a605644334de4987fb60da
BlindedElement = 0252f98f04a956afa469c62ca2850f751b112dc019d4e713c66
2fc0735ef8573f1497cea55b750f27f0efc8330e394a3ab
EvaluationElement = 03fb20c33a7f6f01f2bb388318a6db84f7183bc3bd5e5840
302fe38b6b313649b523238b4c4c625614440dd6ddbcbcc7272
Proof = d33c83c1840a48759659a4d417769ae3bb1adb86326a36fa1ff24f70066b
75d0200e5c1e7d9847e91f7d3d6843efc62101c401a7c952cde32ada6fec848450d8
564e2c778af47ece4f50a88c6d2281bdd858b90fdfad8b093c986bc1e59aaa2e
ProofRandomScalar = 7e82569cb56d97e9c20e59311bac3a50735d573abb787b25
1879b77de4df554c91e25e117919a9db2af19b32ce0d501d
Output = af52cf184180177970be0770e1c7920aa307b767556a13de38a64723d8d
cc7b344af9b6dd8f117ac2cef249ee3acc8fb
```

A.4.3.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a  
Info = 7465737420696e666f  
Blind = 01e6e57b7ec6752a45c74f3ed36a3eb8ad0dafb634f668e415357a04fab5  
01c0f6764e854701129e38071b008286c5fc  
BlindedElement = 0257c264a1016e7a1a8236e46cb3bc11a0f13178b03262e0153  
1da14a05e75a811ba4669fcc41cc9453298f71c23834f91c  
EvaluationElement = 0295529cd99f4255be59966e430bef38c93a5261b0624612  
091327c9aedaaaa40d22b03280ec15620bc91d48970f18c68f  
Proof = b5dfd5fc5ddc61ad8234c544aacbf280193da985d9204d5a30ef9d1a5964  
cle70ffdc3d9c986c93f561ab6f91f012c8ef9b9b6fd21c178f01fe172c37c98fd4ef  
05e5b15c15e810241ed6dda051500165d9f79d4e83580ea4c810a95dddcb593c  
ProofRandomScalar = 6b61028c0ce57aa6729d935ef02e2dd607cb7efcf4ae3bba  
c5ec43774e65a9980f648a5af772f5e7337bbeefbee276ca  
Output = 8bc546462de3087cddafcf81435d5802c0c31f557c791b115a092d5b71e  
a2b6e20986bb624ead85c7a63c976c05dcd
```

A.4.3.3. Test Vector 3, Batch Size 2

[illegible]

A.5. OPRF (P-521, SHA-512)

A.5.1. OPRF Mode

[illegible]

A.5.1.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 00b638b3000884019316267eae9b424f812592e4dc9cd7f7aebfb1d3d2b8
c7fa7904503aef20c694a01d3e1154fe98e7232be9eaec5789a012a559367b1f9965
4ddf
BlindedElement = 02016f3fc7b3c84f673c75b3bb3e00ddd81e734cc84fe3bd4a7
671e0a971879b7678c048f40aae87179614abc2261522303257a92127a195298744c
54094b7b87499c0
EvaluationElement = 0301ddff1ac88acd812a2917cee4917f8a692eaabf9fd052
9981441b83e368175b566657729a8be5ba2573e33e7734c146ef4c8b7d41f4503842
80797318ff3a62d79c
Output = 383e3098d74b43f75d2e1136d7e7c08702d992e6f5f24f2bd438f98b86d
9d143ce87281b2daf7d67c94370903ba81495655d6e9626443a895b37bb74c0276f2
a
```

A.5.1.2. Test Vector 2, Batch Size 1

```
Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Blind = 00219598d5f1544830f9d667b683234c68ef3db95227fe3ebdfd963d0307
0055fef107bfef3c79c86b934061f894227b23a69eb0b53f168a4a2230ef6a7d703a
c4ce
BlindedElement = 0201e75b88d5de2b839f356a05c359ed610601450d83dcc9def
649fdb00a9790161e9333cb07978d1567ecab037c498ae2b00d9181abfd7bcee3fee
dc11de88c54190f
EvaluationElement = 0200b99d9694bbf9b9ea783e18e5fd049fb2fdf169cf386a
be304ccf8cdd633f7a1e25083ec6a6ca3a6e82367b38ee3c991e024097cf6fad928b
023817cdc5dea21751
Output = 5100f12a88477ba993cfe8eb5a82a835892b7fa3bdb47dc1db19725e4c1
138798e0f965df4f649e3a159aacal added fdd07034f7b91c0c9ac3d064b50953bb5c867c
3
```

A.5.2. VOPRF Mode

[illegible]

A.5.2.1. Test Vector 1, Batch Size 1

```
Input = 00
Blind = 01dd6b45efbc57c5f087181c9f03d5b5e51b3a90cc9da17604b2e59a9375
9eb0985d2259c20e3783be009527adf47f8fb5b1437cba7731c34ac70bc6ab1b8c14
ff42
BlindedElement = 0201b93fc5997dc0e8acd5b3ffa3a6f1be1522986c17ed60e5b
ad7b057136b3b7e31b5a7073a744a1304bf9bce4a27b02d77f1caf73a5f72686fa9e
83dbe9f730b4304
EvaluationElement = 0301142205a8fb983efb6d76111db30e2a6e7c54724fe0ee
54f842a477cbf03adcb2cca8df2f165a65694e7a056948f8afd651b32ea8153cc26f
819cc5b1243f383910
Proof = 00fd70ea3e4b7b008de749adb7403ca66f7aa56ae06a587d7d74a06daf6d
5c4dcce8a81acdbee1fcd21ce55db4440383b9fcfc1d584db6fa022a5fd62595c7d9
39820116ad656f5ad470b5bdd208248a0a0b064960c80e180239691c5eb77b4a760e
eda8c27cf3cddc37d6329ad4997a37ffe9aba6f96d45e3e67fad86ec7b1d3a47487f
ProofRandomScalar = 01ce330164821b9b2a108e3ef8964622075015ac9ea0f838
0dcce04b4c70b85f82bd8d1806c3f85daa0e690689a7ed6faa65712283a076c4eaae
988dcf39d6775f40
Output = b3e837431aaafdfa8efbf486d70ca2d4364ef86afc7a8941d9bf1a6adb7
bfd8c5302f91ee5796d956b5d3ea95fd0138d55d3059b1f4febfb8cfd552e31fa2cf9
7
```

A.5.2.2. Test Vector 2, Batch Size 1

[illegible]

A.5.2.3. Test Vector 3, Batch Size 2

[illegible]

A.5.3. POPRF Mode

[illegible]

A.5.3.1. Test Vector 1, Batch Size 1

```
Input = 00
Info = 7465737420696e666f
Blind = 00dc9f04fb076cffe7d179d692a05b0c2210b6c008c1062c1e54514ef654
eeefc0519dd1867571c9d518e305fdf463231b6ec8b7498e2122a7a6033b6261a1696
a773
BlindedElement = 03009a6b363627cbc6ba5f241493a724a69ca7a85f203fb5100
bde9f36ee57e3fe75a5b41d10c6d9a2799fcee9cd1f4bcd730cb8d9be7aa5e8a7a48
8b6ae3004afd2a8
EvaluationElement = 03009ae81470679a5c5733401488cc6648a522a208e698e9
879307e794158ce508e08a50556ec66a055f05f5d5276231258d95d004a49a308037
2f3e9d2075753c010f
Proof = 0122e18e5c3e2242617098cf1d6b5868d66fb4f4816ddd3769e5b7f326f0
ea3d79cd8b8b87be31c1acb9559a2ffdd13f4af7ee143e5081a2db996f3a7d2da839
73e100f559c9dbb7b16df3d5f609d2f8f2184e9e204e6444db72608e4816beee31c8
59dfabfe137bc3bae06947d767cd8cb6ad634134cf6faec24bc8341d51b584872ae1
ProofRandomScalar = 00c07a53a1c70f44466b3861be4f8ef48c2bb1aec2e478e3
41c467fd4a2638aeca63ed6c4bc48d008bca3f36f043e0eb73a44aba77e5e37d5ab1
389e09b80a34cfaa
Output = 70ad5e29de9f6e35f16afab3b97c1b26fdf6be0da60aff48a99980ddb8d
7c2d728a8a5d2837179bfddd612712e014c0c9b9596cbb5a6ee6761c564dbb8921b4
e
```

A.5.3.2. Test Vector 2, Batch Size 1

```

Input = 5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 0085ad3fc8c91caec3bd7699591b10d6da93877a470e128f38030627dffc
bbbf1f576b38677841fc47af778f9d85ac9bce6279388ddf4607e295e64cea6f4f95f
78b8
BlindedElement = 0201eeaedeb3692cc0ecfeacdf9cab61947eb0d23bbfe2e1fbf
8de0907f9410b6089d060d3af63411fd81b9d588fa2c48bf8ec63ec66c14b86d2371
24042ca83fc99e1
EvaluationElement = 0300886138e19945036ebe6f4195cf9f688d9e5a7c89597d
feea6e0e5fcf4b53a9dfa280c8409b6abe8051e3394279d0b669440af8a27aad169d
e10446eb88e09d6801
Proof = 01cb4d8a14eeb472ee3e2fbfe3f6d49f3654cfe6238254bea17ce30848ca
934e20e82c2a33d140de55b24fab047811e20b46f6dcdf3c0945c802e84891316186
17ef001e233aa2c3d674bce7465278faab6300d4f6b5463e1597d74e2a69865bb068
1604f9210edb50bf341d836dc09af85e603b4b2b8b55c90c2efd979a4e312b653e1
ProofRandomScalar = 003a09eed29f2e7f8950d766270d390db7a53b8080b89cb9
e024e1e008d83bd90e94f501281b6b49c351c959348b3a65f24c6f74e77a62905a6d
3e4b0b10600a7cbc
Output = ee2d8e42030da6283ab59a11f41a171c65e208306e00c6f965a56c10f33
bf0942bb38b7e1a33c70bc3542d27220379cbcef8b91898c720be948e9db214a14bb
9

```

A.5.3.3. Test Vector 3, Batch Size 2

```
Input = 00,5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a5a
Info = 7465737420696e666f
Blind = 019dd87ebabccec2627d4006b698d9ba57f6e207c989448d39fe0431e60c
9a9a4110596d5a16fa6cdf3f66467524f295b5dc8f3492c6da02dd7387bd1dc40065
b232,00adaeeed48a6f9a8fb57640c3bff88d3ab3cc52ef969f02beaba2c6e32c2f3
7baaf4ee9c691833dc081e2a0fb6ff636525457a21c1fc56bf3514635ac7fb8618f7
3
BlindedElement = 030170a994d40e8517aed3a7efae01b650dc131clae07158f02
ad70d211348a4b328add9d17e93d2e747dd8bc6960a5ab3bec6a9a29f793bf5663d0
ab108b5f84fa751,03011045c5ea9567626cf6d2baa158830b035e66c249df4967bb
bf917e64bf27e4ec49623704a7c621b32f05elc7bf1b89960c82d4203c4efa6a1056
e083be789d017f
EvaluationElement = 0300fadd09cc84c7e91c2173be0e65ee3c1b6ef98cf0cdd7
57ec432f12f13b5d457edc0311d61ddd831f8becd5231bdc492e92c9d0c103e55ea5
16ee00fe64c10d0e8e,03005941fb7eabc6353a5c2e4a6b284b7b8ee8da6c4435af6
c4c472195bb0deb44e7dc215299c7fe38feafa2b0a1a1db7dd3090c5ce8171247f64
7da20e04acef8164d
Proof = 0186deebd9e2db71ca43bcb57311371390c2d04ac9c3189e155f10c9c548
f6f22c051d38203493176e8392ef405c783759c735cb6f7219636c140cb2dc070a11
58c001bfdb12f34e00a582e22e1eb2fbdebcbffe4b6e0818de5c50451617213e1ab20
fa5392eb3b535206140a2619732c012d5f331a615755f5397feb9e1fb16d1320d20d
ProofRandomScalar = 010a82559ee5e4ba79c390c4033405e3f792bc49daa905c6
94707e7e0191104b34d68c7cc81c2e392da60b838eadf434b693d9b4f7c7beb31e37
008156656c19382b
Output = 70ad5e29de9f6e35f16afab3b97c1b26fdf6be0da60aff48a99980ddb8d
7c2d728a8a5d2837179bfddd612712e014c09b9596cbb5a6ee6761c564dbb8921b4
e,ee2d8e42030da62843ab59a11f41a171c65e208306e00c6f965a56c10f33bf0942b
b38b7e1a33c70bc3542d27220379cbcef8b91898c720be948e9db214a14bb9
```

Authors' Addresses

Alex Davidson
Brave Software

Email: alex.davidson92@gmail.com

Armando Faz-Hernandez
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America

Email: armfazh@cloudflare.com

Nick Sullivan
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America

Email: nick@cloudflare.com

Christopher A. Wood
Cloudflare, Inc.
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net

DRIP
Internet-Draft
Intended status: Standards Track
Expires: 10 October 2022

R. Moskowitz
HTT Consulting
S. Card
A. Wiethuechter
AX Enterprize
8 April 2022

UAS Operator Privacy for RemoteID Messages
draft-moskowitz-drip-operator-privacy-10

Abstract

This document describes a method of providing privacy for UAS Operator/Pilot information specified in the ASTM UAS Remote ID and Tracking messages. This is achieved by encrypting, in place, those fields containing Operator sensitive data using a hybrid ECIES.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 10 October 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	2
2. Terms and Definitions	3
2.1. Requirements Terminology	3
2.2. Definitions	3
3. The Operator - USS Security Relationship	4
3.1. ECIES Shared Secret Generation	4
4. System Message Privacy	5
4.1. Rules for encrypting System Message content	6
4.2. Rules for decrypting System Message content	6
5. Operator ID Message Privacy	6
5.1. Rules for encrypting Operator ID Message content	7
5.2. Rules for decrypting Operator ID Message content	7
6. Cipher choices for Operator PII encryption	7
6.1. Using AES-CFB16	8
6.2. Using a Feistel scheme	8
6.3. Using AES-CTR	8
7. DRIP Requirements addressed	8
8. ASTM Considerations	9
9. IANA Considerations	9
10. Security Considerations	9
10.1. CFB16 Risks	9
10.2. Crypto Agility	9
10.3. Key Derivation vulnerabilities	10
10.4. KMAC Security as a KDF	10
11. Normative References	10
12. Informative References	11
Appendix A. Feistel Scheme	12
Acknowledgments	12
Authors' Addresses	12

1. Introduction

This document defines a mechanism to provide privacy in the ASTM Remote ID and Tracking messages [F3411-19] by encrypting, in place, those fields that contain sensitive UAS Operator/Pilot information. Encrypting in place means that the ciphertext is exactly the same length as the cleartext, and directly replaces it.

An example of and an initial application of this mechanism is the 8 bytes of UAS Operator/Pilot (hereafter called simply Operator) longitude and latitude location in the ASTM System Message (Msg Type 0x4). This meets the Drip Requirements [RFC9153], Priv-01.

It is assumed that the Operator, via the UAS, registers an operation with its USS. During this operation registration, the UAS and USS exchange public keys to use in the hybrid ECIES. The USS key may be

long lived, but the UAS key SHOULD be unique to a specific operation. This provides protection if the ECIES secret is exposed from prior operations.

The actual Tracking message field encryption MUST be an "encrypt in place" cipher. There is rarely any room in the tracking messages for a cipher IV or encryption MAC (AEAD tag). There is rarely any data in the messages that can be used as an IV. The AES-CFB16 mode of operation proposed here can encrypt a multiple of 2 bytes.

The System Message is not a simple, one-time, encrypt the PII with the ECIES derived key. The Operator may move during a operation and these fields change, correspondingly. Further, not all messages will be received by the USS, so each message's encryption must stand on its own and not be at risk of attack by the content of other messages.

Another candidate message is the optional ASTM Operator ID Message (Msg Type 0x5) with its 20 character Operator ID field. The Operator ID does not change during an operation, so this is a one-time encryption operation for the operation. The same cipher SHOULD be used for all messages from the UAS and this will influence the cipher selection.

Future applications of this mechanism may be provided. The content of the System Message may change to meet CAA requirements, requiring encrypting a different amount of data. At that time, they will be added to this document.

Editor note: The Rules for allowing encryption need to be updated to handle the UA operating in Broadcast Remote ID only mode. That is conditions where the USS cannot notify the UAS to stop encrypting.

2. Terms and Definitions

2.1. Requirements Terminology

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.

2.2. Definitions

See Section 2.2 of [RFC9153] for common DRIP terms.

ECIES

Elliptic Curve Integrated Encryption Scheme. A hybrid encryption scheme which provides semantic security against an adversary who is allowed to use chosen-plaintext and chosen-ciphertext attacks.

Keccak (KECCAK Message Authentication Code):

The family of all sponge functions with a KECCAK-f permutation as the underlying function and multi-rate padding as the padding rule. It refers in particular to all the functions referenced from [NIST.FIPS.202] and [NIST.SP.800-185].

KMAC (KECCAK Message Authentication Code):

A PRF and keyed hash function based on KECCAK.

3. The Operator - USS Security Relationship

All CAAs have rules defining which UAS must be registered to operate in their National Airspace. This includes UAS and Operator registration in a USS. Further, operator's are expected to report flight operations to their USS. This operation reporting provides a mechanism for the USS and operator to establish an operation security context. Here it will be used to exchange public keys for use in ECIES.

The operator's ECIES public key SHOULD be unique for each operation. The USS ECIES public key may be unique for each operator and operation, but not required. For best post-compromise security (PCS), the USS ECIES public key should be changed over some operational window.

The public key algorithm should be Curve25519 [RFC7748]. Correspondingly, the ECIES 128 bit shared secret should be generated using KMAC [NIST.SP.800-185].

3.1. ECIES Shared Secret Generation

The KMAC function provides a new, more efficient, key derivation function over HKDF [RFC5869]. This will be referred to as KKDF.

HKDF needs a minimum of 4 hash functions (e.g. SHA256). KKDF does an equivalent shared secret generation in a single Keccak Sponge operation.

When the USS - UAS Operation Security Context is established, the UAS provides its UAS ID (null padded to 20 characters per [F3411-19]) and a 256 bit random nonce to the USS. These are inputs, along with the ECDH keys to produce the shared secret as follows.

A 64 bit UNIX timestamp for the operation time is also included in the Operation Security Context. This will be used in the IV construction.

Per [NIST.SP.800-56Cr1], Section 4.1, Option 3:

$$\text{Shared Secret} = \text{KMAC128}(\text{salt}, \text{IKM}, \text{L}, \text{S})$$

L is the derived key bit length. Since only a single key is needed, L=128.

S is the byte string 01001011 || 01000100 || 01000110, which represents the sequence of characters "K", "D", and "F" in 8-bit ASCII.

$$\text{salt} = \text{Nonce-USS} \mid \text{Nonce-UAS}$$

There are special security considerations for IKM per [RFC7748]. The IKM as follows:

$$\text{IKM} = \text{Diffie-Hellman secret} \mid \text{USS-ID} \mid \text{RID}$$

4. System Message Privacy

The System Message contains 8 bytes of Operator specific information: Longitude and Latitude of the Remote Operator (Pilot in the field description) of the UA. The GCS MAY encrypt these as follows.

Editors Note: The next version of [F3411-19], currently in ballot, is adding a 2 byte Operator Altitude field, thus increasing the Operator specific information to 10 bytes. This change will be delineated via Protocol Version field. It is this future shift from a multiple of 4 bytes to a multiple of 2 bytes that is the reason to change from CFB32 in earlier drafts to CFB16 used now.

The 8 bytes of Operator information are encrypted, using the ECIES derived 128 bit shared secret, with one of the cipher's specified below. The choice of cipher is based on USS policy and is agreed to as part of the operation registration. AES-CFB16 is the recommended default cipher.

ASTM Remote ID and Tracking messages [F3411-19] SHOULD be updated to allow Bit 5 of the Flags byte in the System Message set to "1" to indicate the Operator information is encrypted.

The USS similarly decrypts these 8 bytes and provides the information to authorized entities.

4.1. Rules for encrypting System Message content

If the Operator location is encrypted the encrypted bit flag MUST be set to 1.

The Operator MAY be notified by the USS that the operation has entered a location or time where privacy of Operator location is not allowed. In this case the Operator MUST disable this privacy feature and send the location unencrypted or land the UA or route around the restricted area.

If the UAS loses connectivity to the USS, the privacy feature SHOULD be disabled or land the UA.

If the operation is in an area or time with no Internet Connectivity, the privacy feature MUST NOT be used.

4.2. Rules for decrypting System Message content

An Observer receives a System Message with the encrypt bit set to 1. The Observer sends a query to its USS Display Provider containing the UA's ID and the encrypted fields.

The USS Display Provider MAY deny the request if the Observer does not have the proper authorization.

The USS Display Provider MAY reply to the request with the decrypted fields if the Observer has the proper authorization.

The USS Display Provider MAY reply to the request with the decrypting key if the Observer has the proper authorization.

The Observer MAY notify the USS through its USS Display Provider that content privacy for a UAS in this location/time is not allowed. If the Observer has the proper authorization for this action, the USS notifies the Operator to disable this privacy feature.

5. Operator ID Message Privacy

The Operator ID Message contains the 20 byte Operator ID. The GCS MAY encrypt these as follows.

The 20 bytes Operator ID is encrypted, using the ECIES derived 128 bit shared secret, with one of the cipher's specified below. The choice of cipher is based on USS policy and is agreed to as part of the operation registration. AES-CFB16 is the recommended default cipher.

ASTM Remote ID and Tracking messages [F3411-19] SHOULD be updated to allow Operator ID Type in the Operator ID Message set to "1" to indicate the Operator ID is encrypted.

The USS similarly decrypts these 20 bytes and provides the information to authorized entities.

5.1. Rules for encrypting Operator ID Message content

If the Operator ID is encrypted the Operator ID Type field MUST be set to 1.

The Operator MAY be notified by the USS that the operation has entered a location or time where privacy of Operator ID is not allowed. In this case the Operator MUST disable this privacy feature and send the ID unencrypted or land the UA or route around the restricted area.

If the UAS loses connectivity to the USS, the privacy feature SHOULD be disabled or land the UA.

If the operation is in an area or time with no Internet Connectivity, the privacy feature MUST NOT be used.

5.2. Rules for decrypting Operator ID Message content

An Observer receives a Operator ID Message with the Operator ID Type field set to 1. The Observer sends a query to its USS Display Provider containing the UA's ID and the encrypted fields.

The USS Display Provider MAY deny the request if the Observer does not have the proper authorization.

The USS Display Provider MAY reply to the request with the decrypted fields if the Observer has the proper authorization.

The USS Display Provider MAY reply to the request with the decrypting key if the Observer has the proper authorization.

The Observer MAY notify the USS through its USS Display Provider that content privacy for a UAS in this location/time is not allowed. If the Observer has the proper authorization for this action, the USS notifies the Operator to disable this privacy feature.

6. Cipher choices for Operator PII encryption

6.1. Using AES-CFB16

CFB16 is defined in [NIST.SP.800-38A], Section 6.3. This is the Cipher Feedback (CFB) mode operating on 16 bits at a time. This variant of CFB can be used to encrypt any multiple of 2 bytes of cleartext.

The Operator includes a 64 bit UNIX timestamp for the operation time, along with its operation public key. The Operator also includes the UA MAC address (or multiple addresses if flying multiple UA).

The 128 bit IV for AES-CFB16 is constructed by the Operator and USS as: SHAKE128(MAC|UTCTime|Message_Type, 128). Inclusion of the ASTM Message_Type ensures a unique IV for each Message type that contains PII to encrypt.

AES-CFB16 would then be used to encrypt the Operator information.

6.2. Using a Feistel scheme

If the encryption speed doesn't matter, we can use the following approach based on the Feistel scheme. This approach is already being used in format-preserving encryption (e.g. credit card numbers). The Feistel scheme is explained in Appendix A.

6.3. Using AES-CTR

If 2 bytes of the Message can be set aside to contain a counter that is incremented each time the Operator information changes, AES-CTR can be used as follows.

The Operator includes a 64 bit UNIX timestamp for the operation time, along with its operation public key. The Operator also includes the UA MAC address (or multiple addresses if flying multiple UA).

The high order bits of an AES-CTR counter is constructed by the Operator and USS as: SHAKE128(MAC|UTCTime|Message_Type, 112). Inclusion of the ASTM Message_Type ensures a unique IV for each Message type that contains PII to encrypt.

AES-CTR would then be used to encrypt the Operator information.

7. DRIP Requirements addressed

This document provides solution to PRIV-1 for PII in the ASTM System Message.

8. ASTM Considerations

ASTM will need to make the following changes to the "Flags" in the System Message (Msg Type 0x4):

Bit 5:

Value 1 for encrypted; 0 for cleartext (see Section 4).

ASTM will need to make the following changes to the "Operator ID Type" in the Operator ID Message (Msg Type 0x5):

Operator ID Type

Value 1 for encrypted Operator ID (see Section 5).

9. IANA Considerations

TBD

10. Security Considerations

An attacker has no known text after decrypting to determine a successful attack. An attacker can make assumptions about the high order byte values for Operator Longitude and Latitude that may substitute for known cleartext. There is no knowledge of where the operator is in relation to the UA. Only if changing location values "make sense" might an attacker assume to have revealed the operator's location.

10.1. CFB16 Risks

Using the same IV for different Operator information values with CFB16 presents a cryptoanalysis risk. Typically only the low order bits would change as the Operators position changes. The risk is mitigated due to the short-term value of the data. Further analysis is need to properly place risk.

10.2. Crypto Agility

The ASTM Remote ID Messages do not provide any space for a crypto suite indicator or any other method to manage crypto agility.

All crypto agility is left to the USS policy and the relation between the USS and operator/UAS. The selection of the ECIES public key algorithm, the shared secret key derivation function, and the actual symmetric cipher used for on the System Message are set by the USS which informs the operator what to do.

10.3. Key Derivation vulnerabilities

[RFC7748] warns about using Curve25519 and Curve448 in Diffie-Hellman for key derivation:

Designers using these curves should be aware that for each public key, there are several publicly computable public keys that are equivalent to it, i.e., they produce the same shared secrets. Thus using a public key as an identifier and knowledge of a shared secret as proof of ownership (without including the public keys in the key derivation) might lead to subtle vulnerabilities.

This applies here, but may have broader consequences. Thus two endpoint IDs are included with the Diffie-Hellman secret.

10.4. KMAC Security as a KDF

Section 4.1 of NIST SP 800-185 [NIST.SP.800-185] states:

"The KECCAK Message Authentication Code (KMAC) algorithm is a PRF and keyed hash function based on KECCAK . It provides variable-length output"

That is, the output of KMAC is indistinguishable from a random string, regardless of the length of the output. As such, the output of KMAC can be divided into multiple substrings, each with the strength of the function (KMAC128 or KMAC256) and provided that a long enough key is used, as discussed in Sec. 8.4.1 of SP 800-185.

For example KMAC128(K, X, 512, S), where K is at least 128 bits, can produce 4 128 bit keys each with a strength of 128 bits. That is a single sponge operation is replacing perhaps 5 HMAC-SHA256 operations (each 2 SHA256 operations) in HKDF.

11. Normative References

[NIST.FIPS.202]

Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.202, July 2015, <<https://doi.org/10.6028/nist.fips.202>>.

[NIST.SP.800-185]

Kelsey, J., Change, S., and R. Perlner, "SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-185, December 2016, <<https://doi.org/10.6028/nist.sp.800-185>>.

[NIST.SP.800-38A]

Dworkin, M., "Recommendation for block cipher modes of operation :: methods and techniques", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-38a, 2001, <<https://doi.org/10.6028/nist.sp.800-38a>>.

[NIST.SP.800-56Cr1]

Barker, E., Chen, L., and R. Davis, "Recommendation for key-derivation methods in key-establishment schemes", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56cr1, April 2018, <<https://doi.org/10.6028/nist.sp.800-56cr1>>.

[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/info/rfc2119>>.

[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/info/rfc8174>>.

12. Informative References

[F3411-19] ASTM International, "Standard Specification for Remote ID and Tracking", February 2020, <<http://www.astm.org/cgi-bin/resolver.cgi?F3411>>.

[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/info/rfc5869>>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.

[RFC9153] Card, S., Ed., Wiethuechter, A., Moskowitz, R., and A. Gurtov, "Drone Remote Identification Protocol (DRIP) Requirements and Terminology", RFC 9153, DOI 10.17487/RFC9153, February 2022, <<https://www.rfc-editor.org/info/rfc9153>>.

Appendix A. Feistel Scheme

This approach is already being used in format-preserving encryption.

According to the theory, to provide CCA security guarantees (CCA = Chosen Ciphertext Attacks) for m-bit encryption $X \rightarrow Y$, we should choose $d \geq 6$. It seems very ineffective that when shortening the block length, we have to use 6 times more block encryptions. On the other hand, we preserve both the block cipher interface and security guarantees in a simple way.

How to encrypt an m-bit plaintext X using an n-bit block cipher
 $E = \{E_K\}$ for $n > m$?

Enc(X, K):

1. $Y \leftarrow X$.
2. Split Y into 2 equal parts: $Y = Y1 \parallel Y2$
 (let us assume for simplicity that m is even).
3. For $i = 1, 2, \dots, d$ do:
 $Y \leftarrow Y2 \parallel (Y1 \wedge \text{first_m/2_bits}(E_K(Y2 \parallel C_i)))$,
 where C_i is a $(n - m/2)$ -bit round constant.
4. $Y \leftarrow Y2 \parallel Y1$.
5. Return Y.

Dec(Y, K):

1. $X \leftarrow Y$.
2. Split X into 2 equal parts: $X = X1 \parallel X2$.
3. For $i = d, \dots, 2, 1$ do:
 $X \leftarrow X2 \parallel (X1 \wedge \text{first_m/2_bits}(E_K(X2 \parallel C_i)))$.
4. $X \leftarrow X2 \parallel X1$.
5. Return X.

Acknowledgments

The recommended ciphers come from discussions on the IRTF CFRG mailing list.

Authors' Addresses

Robert Moskowitz
HTT Consulting
Oak Park, MI 48237
United States of America
Email: rgm@labs.htt-consult.com

Stuart W. Card
AX Enterprize
4947 Commercial Drive
Yorkville, NY 13495
United States of America
Email: stu.card@axenterprize.com

Adam Wiethuechter
AX Enterprize
4947 Commercial Drive
Yorkville, NY 13495
United States of America
Email: adam.wiethuechter@axenterprize.com

HIP
Internet-Draft
Updates: 7401, 7402 (if approved)
Intended status: Standards Track
Expires: 3 February 2022

R. Moskowitz
HTT Consulting
S. Card
A. Wiethuechter
AX Enterprize
2 August 2021

New Cryptographic Algorithms for HIP
draft-moskowitz-hip-new-crypto-10

Abstract

This document provides new cryptographic algorithms to be used with HIP. The Edwards Elliptic Curve and the Keccak sponge functions are the main focus. The HIP parameters and processing instructions impacted by these algorithms are defined.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 3 February 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Terms and Definitions	3
2.1. Requirements Terminology	3
2.2. Definitions	3
3. HIP Parameter values for new Cryptographic Functions	4
3.1. Elliptic Curves for Diffie-Hellman	4
3.1.1. DIFFIE_HELLMAN	5
3.2. Edward Digital Signature Algorithm for HITs	5
3.2.1. HOST_ID	5
3.2.2. HIT_SUITE_LIST	6
3.3. Hashing in HIP	7
3.3.1. Hashing with the Sponge Functions	7
3.3.2. RHASH	8
3.3.3. HIP_MAC and HIP_MAC2	8
3.4. HIP Cipher	9
3.4.1. HIP_CIPHER	9
3.5. ESP Transform	9
3.5.1. ESP_TRANSFORM	10
4. Generating a HIT from an HI	10
5. HIP KEYMAT Generation	10
5.1. The Keccak KEYMAT	11
5.2. The Xoodyak KEYMAT	11
6. Pseudorandom Function (PRF)	12
7. IANA Considerations	12
8. Security Considerations	12
8.1. Keymat vulnerabilities	12
8.2. KMAC Security as a KDF	13
9. Acknowledgments	13
10. References	13
10.1. Normative References	13
10.2. Informative References	15
Authors' Addresses	16

1. Introduction

This document adds new cryptographic algorithms for HIPv2 [RFC7401] and [RFC7402]. This includes:

- * New elliptic curves for ECDH.
- * The Edwards Elliptic Curve Digital Signature Algorithm (EdDSA) used in Host Identities (HI) and for Base Exchange (BEX) signatures.
- * Hashes used in Host Identity Tag (HIT) generation, and wherever else hashes are needed.

- * Keyed hashes used for KEYMAT generation and packet MACing operations.
- * AEAD and stream ciphers to use in HIP and HIP enabled secure communication protocols.

The hashes and encryption are all built on the Keccak [Keccak] sponge function and the Xoodyak [Xoodyak] lightweight scheme.

These additions reflect selection of advances in the field of cryptography that would best benefit HIP, particularly in constrained devices and communications.

Ed Note: The Xoodyak function calls should be considered the 1st best effort. There are a few areas open for discussion, like which of the 3 choices for adding in the nonce to the AEAD mode and when to use counter and Id. Also there may be copy errors from the source specification, nicer function calls, better acronyms.

2. Terms and Definitions

2.1. Requirements Terminology

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.

2.2. Definitions

cSHAKE (The customizable SHAKE function [NIST SP800-185]):
Extends the SHAKE scheme to allow users to customize their use of the function.

DEC function (Doubly-Extendable Cryptographic function):
An extendable output function (XOF) that accepts sequences of strings as input and that supports incremental queries efficiently.

DECK function (Doubly-Extendable Cryptographic Keyed function):
A keyed function that takes a sequence of input strings and returns a pseudorandom string of arbitrary length and that can be computed incrementally.

Keccak:

The family of all sponge functions with a KECCAK-f permutation as the underlying function and multi-rate padding as the padding rule. In particular all the functions referenced from [NIST FIPS-202] and [NIST SP800-185].

KMAC (KECCAK Message Authentication Code [NIST SP800-185]):

A pseudo random function (PRF) and keyed hash function based on KECCAK.

SHAKE (Secure Hash Algorithm KECCAK [NIST FIPS-202]):

A secure hash that allows for an arbitrary output length. SHAKE128 and SHAKE256 are instances of XOFs. SHAKE is shorthand for SHAKE128.

PRF (Pseudorandom Function):

A function that takes as input a key and that it is hard to distinguish from a random oracle by an adversary that does not know the key.

XHASH (Xoodyak Hash Algorithm):

A secure hash, based on Xoodyak, that allows for an arbitrary output length. XHASH is an instance of XOF.

XMAC (Xoodyak Message Authentication Code):

A keyed hash function similar to KMAC, based on Xoodyak, that allows for an arbitrary output length.

XOF (eXtendable-Output Function [NIST FIPS-202]):

A function on bit strings (also called messages) in which the output can be extended to any desired length.

3. HIP Parameter values for new Cryptographic Functions

HIP parameters carry information that is necessary for establishing and maintaining a HIP association. For example, the device's public keys as well as the signaling for negotiating ciphers and payload handling are encapsulated in HIP parameters. Additional information, meaningful for end hosts or middleboxes, may also be included in HIP parameters. The specification of the HIP parameters and their mapping to HIP packets and packet types is flexible to allow HIP extensions to define new parameters and new protocol behavior.

3.1. Elliptic Curves for Diffie-Hellman

Elliptic curves Curve25519 and Curve448 [RFC7748] are specified here for use in the HIP Diffie-Hellman exchange.

Curve25519 and Curve448 are already defined in Section 5.2.1 of [hip-dex], using the HIP-DEX CKDF. Here they are defined for using the new KMAC [NIST SP800-185] or XMAC [Xoodyak] derived KDF in Section 5.

3.1.1. DIFFIE_HELLMAN

The DIFFIE_HELLMAN parameter may be included in selected HIP packets based on the DH Group ID selected. The DIFFIE_HELLMAN parameter is defined in Section 5.2.7 of [RFC7401].

The following Elliptic Curves are defined here:

Group	KDF	Value
Curve25519 [RFC7748]	KMAC	13
Curve448 [RFC7748]	KMAC	14

A new KDF for KEYMAT, Section 6.5 of [RFC7401] using Keccak or Xoodyak is defined in Section 5.

3.2. Edward Digital Signature Algorithm for HITs

This section is extracted from Appendix D of [drip-rid]. It may later be pulled and only maintained there.

Edwards-Curve Digital Signature Algorithm (EdDSA) [RFC8032] are specified here for use as Host Identities (HIs) per HIPv2 [RFC7401]. Further the HIT_SUITE_LIST is specified as used in [RFC7343].

3.2.1. HOST_ID

The HOST_ID parameter specifies the public key algorithm, and for elliptic curves, a name. The HOST_ID parameter is defined in Section 5.2.19 of [RFC7401].

Algorithm profiles	Values
EdDSA	13 [RFC8032]

For hosts that implement EdDSA as the algorithm, the following ECC curves are available:

Algorithm	Curve	Values
EdDSA	RESERVED	0
EdDSA	EdDSA25519	1 [RFC8032]
EdDSA	EdDSA25519ph	2 [RFC8032]
EdDSA	EdDSA448	3 [RFC8032]
EdDSA	EdDSA448ph	4 [RFC8032]

3.2.2. HIT_SUITE_LIST

The HIT_SUITE_LIST parameter contains a list of the supported HIT suite IDs of the Responder. Based on the HIT_SUITE_LIST, the Initiator can determine which source HIT Suite IDs are supported by the Responder. The HIT_SUITE_LIST parameter is defined in Section 5.2.10 of [RFC7401].

The following HIT Suite ID is defined, and the relationship between the four-bit ID value used in the OGA ID field and the eight-bit encoding within the HIT_SUITE_LIST ID field is clarified:

HIT Suite	Four-bit ID	Eight-bit encoding
RESERVED	0	0x00
EdDSA/cSHAKE128	5	0x50
EdDSA/XHASH	6	0x60

The following table provides more detail on the above HIT Suite combinations. The input for each generation algorithm is the encoding of the HI as defined herein.

The output of cSHAKE128 and XHASH are variable per the needs of a specific ORCHID construction. It is at most 96 bits long and is directly used in the ORCHID (without truncation).

Index	Hash function	HMAC	Signature algorithm family	Description
5	cSHAKE128	KMAC128	EdDSA	EdDSA HI hashed with cSHAKE128, output is variable
6	XHASH	XMAC	EdDSA	EdDSA HI hashed with XMAC, output is variable

Table 1: HIT Suites

3.3. Hashing in HIP

Hashing is used in HIP for HIT generation and keyed hashes of HIP payloads. The hash algorithm used is designated as part of the HIT_SUITE_ID. The keyed hash function is the "common" such function used in conjunction with the HIT hash.

3.3.1. Hashing with the Sponge Functions

The XOF function in SHA-3, Secure Hash Algorithm Keccak (SHAKE) [NIST FIPS-202] and the more recent Xoodyak [Xoodyak] algorithm are called sponge functions. Sponge functions have a special feature in which an arbitrary number of output bits are "squeezed" out of the hashing state. This is a significant use change in that hash truncation or multiple "runs" for enough bits are not used with sponge functions.

3.3.1.1. cSHAKE, the customizable SHAKE function

The customizable SHAKE function (cSHAKE) in [NIST SP800-185] will be used as a HIP hash. As a Keccak XOF, it does not use the truncation operation that other hashes need. The invocation of cSHAKE specifies the desired number of bits in the hash output. Further, cSHAKE has a parameter 'S' as a customization bit string. This parameter will be used for including hash specific customization like the ORCHID Context Identifier in a standard fashion.

Hardware implementation of Keccak in VHDL is available from Keccak [Keccak] team website.

3.3.1.2. The Xoodyak Hash

The Xoodyak [Xoodyak] sponge function is a candidate in the NIST Lightweight Cryptography (LWC) Standardization process (see [NISTIR 8369]). Xoodyak has been selected here for use in HIP from the LWC 2nd round candidates as it was developed by the Keccak team, making it more directly in line with Keccak.

Xoodyak has a hash function mode. More specifically, this hash mode is an extendable output function (XOF).

As the Xoodyak specification [Xoodyak_Spec] does not provide high-level function calls, rather a set of primitives to use to construct the various modes, the appropriate primitive calls will be detailed below. Xoodyak as a hash will be called here "XHASH".

To get a n-byte digest of some input x: XHASH(n, x), use the following set of Xoodyak primitives:

```
Cyclist(,,)
Absorb(x)
Squeeze(n)
```

Xoodyak can also naturally implement a DEC function and process a sequence of strings. Here the output depends on the sequence as such and not just on the concatenation of the different strings. To compute a n-byte digest, XHASH(n, {x1, x2, x3}) the Xoodyak primitives are:

```
Cyclist(,,)
Absorb(x1)
Absorb(x2)
Absorb(x3)
Squeeze(n)
```

The equivalent of the parameter 'S' in cSHAKE above can be implemented as the last Absorb primitive call in the DEC function. That is: XHASH(L, {S, N, X}) is equivalent to cSHAKE(X, L, N, S).

3.3.2. RHASH

RHASH is the general term used throughout [RFC7401] to refer to the hash used for a specific HIT suite. For this addendum cSHAKE128 for Keccak or XHASH for Xoodyak is used, even for HITs of EdDSA448.

Unless otherwise specified, L of cSHAKE128 or n of XHASH is 256, resulting in a similar output to SHA256. Any truncation used for, older, fixed output hashes is still used. This is to simplify code integration. One exception to this is in Section 4.

3.3.3. HIP_MAC and HIP_MAC2

The HIP_MAC and HIP_MAC2 parameters in [RFC7401] use HMAC [RFC2104]. This performs two hashes on a string with a key for a keyed hash the length of the underlying hash.

For both HIP_MAC and HIP_MAC2 use, the parameter S below is NULL. It is included for complete function definition.

3.3.3.1. The Keccak Keyed MAC

Here, KMAC from NIST SP 800-185 [NIST SP800-185] is used. This is a single pass using the underlying cSHAKE function. The function call is:

```
KMAC128(Key, Input String, 256, S)
```

3.3.3.2. The Xoodoo Keyed MAC

Here, XMAC is defined as the keyed hash function based on Xoodoo. It is built with primitives from [Xoodoo_Spec] as a DEC function.

To get a n-byte keyed MAC of some input x: XMAC(Key, n, {x, S}). Where n=256, use the following set of Xoodoo primitives:

```
Cyclist(Key, Id,)
Absorb(S)           Only if S is non-null
Absorb(Input String)
Squeeze(32)
```

Id is "HIP_MAC" and "HIP_MAC2" respectively. Note since S is null in this XMAC usage, the first Absorb call is not performed.

3.4. HIP Cipher

HIP encrypted parameters use the HIP_CIPHER, Section 5.2.8 of [RFC7401]. The Xoodoo cipher, [Xoodoo], is recommended. Here Xoodoo is used in encrypt only mode.

3.4.1. HIP_CIPHER

The HIP_CIPHER parameter value for Xoodoo is:

hip_cipher Suite ID	Value
Xoodoo	6 (Xoodoo)

The Xoodoo primitive calls for encrypt only are:

```
Cyclist(Key, Id,)
Absorb(IV)
C Encrypt(P)
```

Where Id is HIP parameter name (e.g. "ENCRYPTED").
 IV is from the encrypted HIP parameter.
 P is the plain-text per the specific HIP encrypted parameter.
 C is the ciphertext.

3.5. ESP Transform

The ESP_TRANSFORM parameter is used during ESP SA establishment, Section 5.1.2 of [RFC7402]. The Xoodoo cipher, [Xoodoo], is recommended. Here Xoodoo is used in AEAD mode.

Further, it is recommended to use Implicit IV ESP [RFC8750] to match its lightweight over-the-air format with the lightweight Xoodyak AEAD cipher.

3.5.1. ESP_TRANSFORM

The ESP_TRANSFORM Suite IDs for Xoodyak are:

hip_cipher		
Suite ID	Value	
Xoodyak-96	16	(Xoodyak)
Xoodyak	17	(Xoodyak)
Implicit IV	18	[8750]

The Implicit IV Suite ID is unique in that it is an AND condition with ciphers that can use it. That is AES-GCM and Xoodyak can both use 'regular' ESP [RFC4303] or [RFC8750].

The Xoodyak primitive calls for AEAD encrypt are:

```
Cyclist(Key, Id,)
Absorb(IV)
Absorb(A)
C   Encrypt(P)
T   Squeeze(t)
```

Where Id is "ESP_TRANSFORM". The IV is either a 32 bit ESP IV per [RFC4303] or the ESP Seq Number per [RFC8750]. P is the plain-text and A is the associated data. t is either 12 or 16. T is the ESP ICV of length t.

4. Generating a HIT from an HI

The EdDSA/cSHAKE based HITs require a new ORCHID generation method than that described in section 3.2 of [RFC7401]. The XOF functionality of cSHAKE produces an output of L bits. This replaces the Encode_96 function in the ORCHID generation.

For identities that are EdDSA public keys, ORCHIDs will be generated per the process defined in Appendix C.2.1 of [drip-rid].

5. HIP KEYMAT Generation

For either the Keccak or Xoodyak KEYMAT generation, the inputs are consistent. The only practical difference is that cSHAKE allows for 128 or 256 bits of strength, whereas Xoodyak only provides 128 bits.

L is the derived key bit length. Since 4 HIP keys are "drawn" from this output, the length is $4 * \text{HIP_key_size}$. Per ASIACRYPT 2017, pp. 606-637 [ASIACRYPT-2017] each of these derived keys will have the same strength as the Diffie-Hellman shared secret.

S is the byte string 01001011 || 01000100 || 01000110, which represents the sequence of characters "K", "D", and "F" in 8-bit ASCII.

Salt and info are derived as defined in sec 6.5 of [RFC7401]. There are special security considerations for IKM per [RFC7748].

5.1. The Keccak KEYMAT

The KMAC function provides a new, more efficient, key derivation function over HKDF [RFC5869]. KMAC as a KDF is defined below.

The two HIs MUST be used in constructing IKM as follows:

$$\text{IKM} = \text{Diffie-Hellman secret} \mid \text{sort}(\text{HI-I} \mid \text{HI-R})$$

The two HIs are separately DER encoded per [RFC7401]

The choice of KMAC128 or KMAC256 is based on the strength of the output key material. For 256 bits of strength equivalent to HMAC-SHA256, use KMAC256. Per [NIST SP800-56Cr1], Section 4.1, Option 3:

$$\text{OKM} = \text{KMAC}[128|256](\text{salt} \mid \text{info}, \text{IKM}, \text{L}, \text{S})$$

5.2. The Xoodyak KEYMAT

Here, XMAC from Section 3.3.3.2 is used. The DEC function $\text{XMAC}("", \text{L}, \{\text{DH}, \text{sort}(\text{HI-I}, \text{HI-R}), \text{info}, \text{Salt}, \text{S}\})$ primitives are:

```
Cyclist( , , )
Absorb(S)
Absorb(salt)
Absorb(info)
Absorb(max(HI-I , HI-R))
Absorb(min(HI-I , HI-R))
Absorb(Diffie-Hellman secret)
Squeeze(L)   Where L is bytes
```

Ed Note: Need to check that all above are well defined bytestrings per 7401. I think they are.

6. Pseudorandom Function (PRF)

Appendix B of NIST SP 800-185 [NIST SP800-185] defines how to use SHAKE, cSHAKE, or KMAC as a PRF.

For Xoodyak, XMAC from Section 3.3.3.2 is used in the same manner as KMAC above.

7. IANA Considerations

IANA will need to make the following changes to the "Host Identity Protocol (HIP) Parameters" registries:

Diffie Hellman:

This document defines the new Curve25519 and Curve448 for the Diffie-Hellman exchange (see Section 3.1.1).

Host ID:

This document defines the new EdDSA Host ID (see Section 3.2.1).

HIT Suite ID:

This document defines the new HIT Suite of EdDSA/cSHAKE and EdDSA/XHASH (see Section 3.2.2).

HIP Cipher:

This document defines the new Xoodyak cipher for HIP encrypted parameters (see Section 3.4.1).

ESP Transform:

This document defines the new Xoodyak cipher and use of [RFC8750] for the ESP Transform parameter (see Section 3.5).

8. Security Considerations

8.1. Keymat vulnerabilities

[RFC7748] warns about using Curve25519 and Curve448 in Diffie-Hellman for key derivation:

Designers using these curves should be aware that for each public key, there are several publicly computable public keys that are equivalent to it, i.e., they produce the same shared secrets. Thus using a public key as an identifier and knowledge of a shared secret as proof of ownership (without including the public keys in the key derivation) might lead to subtle vulnerabilities.

Thus the two Host IDs are included with the Diffie-Hellman secret in the KEYMAT generation.

8.2. KMAC Security as a KDF

Section 4.1 of NIST SP 800-185 [NIST SP800-185] states:

"The KECCAK Message Authentication Code (KMAC) algorithm is a PRF and keyed hash function based on KECCAK . It provides variable-length output"

That is, the output of KMAC is indistinguishable from a random string, regardless of the length of the output. As such, the output of KMAC can be divided into multiple substrings, each with the strength of the function (KMAC128 or KMAC256) and provided that a long enough key is used, as discussed in Sec. 8.4.1 of SP 800-185.

For example KMAC128(K, X, 512, S), where K is at least 128 bits, can produce 4 128 bit keys each with a strength of 128 bits. That is a single sponge operation is replacing perhaps 5 HMAC-SHA256 operations (each 2 SHA256 operations) in HKDF.

9. Acknowledgments

Quynh Dang of NIST gave considerable guidance on using Keccak and the NIST supporting documents. Joan Deamen of the Keccak team was especially helpful in many aspects of using Keccak and Xoodyak, particularly with the KEYMAT section and the strength of the derived keys.

NIST is entering round 3 (final) of its Lightweight Crypto Competition with anticipated selection the end of 2021 or early in 2022. Events in this process will impact selections in this document.

10. References

10.1. Normative References

[NIST FIPS-202]

Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", National Institute of Standards and Technology report, DOI 10.6028/nist.fips.202, July 2015, <<https://doi.org/10.6028/nist.fips.202>>.

[NIST SP800-185]

Kelsey, J., Change, S., and R. Perlner, "SHA-3 derived functions: cSHAKE, KMAC, TupleHash and ParallelHash", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-185, December 2016, <<https://doi.org/10.6028/nist.sp.800-185>>.

[NIST SP800-56Cr1]

Barker, E., Chen, L., and R. Davis, "Recommendation for key-derivation methods in key-establishment schemes", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56cr1, April 2018, <<https://doi.org/10.6028/nist.sp.800-56cr1>>.

[NISTIR 8369]

Sonmez Turan, M., McKay, K., Chang, D., Calik, C., Bassham, L., Kang, J., and J. Kelsey, "Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process", National Institute of Standards and Technology report, DOI 10.6028/nist.ir.8369, July 2021, <<https://doi.org/10.6028/nist.ir.8369>>.

[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/info/rfc2119>>.

[RFC7401] Moskowitz, R., Ed., Heer, T., Jokela, P., and T. Henderson, "Host Identity Protocol Version 2 (HIPv2)", RFC 7401, DOI 10.17487/RFC7401, April 2015, <<https://www.rfc-editor.org/info/rfc7401>>.

[RFC7402] Jokela, P., Moskowitz, R., and J. Melen, "Using the Encapsulating Security Payload (ESP) Transport Format with the Host Identity Protocol (HIP)", RFC 7402, DOI 10.17487/RFC7402, April 2015, <<https://www.rfc-editor.org/info/rfc7402>>.

[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/info/rfc8174>>.

[Xoodyak] Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., and R. Van Keer, "The Xoodyak Cipher and Hash", <<https://keccak.team/xoodyak.html>>.

[Xoodyak_Spec]

Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., and R. Van Keer, "Xoodoo cookbook", 2019, <<https://eprint.iacr.org/2018/767.pdf>>.

10.2. Informative References

[ASIACRYPT-2017]

Daemen, J., Mennink, B., and G. Van Assche, "Full-State Keyed Duplex with Built-In Multi-user Support", DOI 10.1007/978-3-319-70697-9_21, Advances in Cryptology - ASIACRYPT 2017 pp. 606-637, 2017, <https://doi.org/10.1007/978-3-319-70697-9_21>.

[drip-rid] Moskowitz, R., Card, S. W., Wiethuechter, A., and A. Gurtov, "Unmanned Aircraft System Remote Identification (UAS RID)", Work in Progress, Internet-Draft, draft-ietf-drip-rid-08, 25 July 2021, <<https://tools.ietf.org/html/draft-ietf-drip-rid-08>>.

[hip-dex] Moskowitz, R., Hummen, R., and M. Komu, "HIP Diet EXchange (DEX)", Work in Progress, Internet-Draft, draft-ietf-hip-dex-24, 19 January 2021, <<https://tools.ietf.org/html/draft-ietf-hip-dex-24>>.

[Keccak] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "The Keccak Function", <<https://keccak.team/index.html>>.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.

[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/info/rfc5869>>.

[RFC7343] Laganier, J. and F. Dupont, "An IPv6 Prefix for Overlay Routable Cryptographic Hash Identifiers Version 2 (ORCHIDv2)", RFC 7343, DOI 10.17487/RFC7343, September 2014, <<https://www.rfc-editor.org/info/rfc7343>>.

- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [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/info/rfc8032>>.
- [RFC8750] Migault, D., Guggemos, T., and Y. Nir, "Implicit Initialization Vector (IV) for Counter-Based Ciphers in Encapsulating Security Payload (ESP)", RFC 8750, DOI 10.17487/RFC8750, March 2020, <<https://www.rfc-editor.org/info/rfc8750>>.

Authors' Addresses

Robert Moskowitz
HTT Consulting
Oak Park, MI 48237
United States of America

Email: rgm@labs.htt-consult.com

Stuart W. Card
AX Enterprize
4947 Commercial Drive
Yorkville, NY 13495
United States of America

Email: stu.card@axenterprize.com

Adam Wiethuechter
AX Enterprize
4947 Commercial Drive
Yorkville, NY 13495
United States of America

Email: adam.wiethuechter@axenterprize.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 28 April 2022

S. Hendrickson
Google LLC
J. Iyengar
Fastly
T. Pauly
Apple Inc.
S. Valdez
Google LLC
C.A. Wood
Cloudflare
25 October 2021

Private Access Tokens
draft-private-access-tokens-01

Abstract

This document defines a protocol for issuing and redeeming privacy-preserving access tokens. These tokens can adhere to an issuance policy, allowing a service to limit access according to the policy without tracking client identity.

Discussion Venues

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

Source for this draft and an issue tracker can be found at <https://github.com/tfpauly/privacy-proxy>.

Status of This Memo

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

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 28 April 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Motivation	4
1.1.1. Rate-limited Access	4
1.1.2. Client Geo-Location	5
1.1.3. Private Client Authentication	5
1.2. Architecture	5
1.3. Properties and Requirements	7
1.4. Client Identity	9
1.5. User Interaction	10
2. Notation and Terminology	10
3. Configuration	12
4. Token Challenge and Redemption Protocol	13
4.1. Token Challenge	14
4.2. Token Redemption	15
5. Issuance Protocol	16
5.1. State Requirements	17
5.1.1. Client State	17
5.1.2. Mediator State	18
5.1.3. Issuer State	19
5.2. Issuance HTTP Headers	19
5.3. Client-to-Mediator Request	20
5.4. Mediator-to-Issuer Request	23
5.5. Issuer-to-Mediator Response	24
5.6. Mediator-to-Client Response	25
5.7. Encrypting Origin Names	26
5.8. Non-Interactive Schnorr Proof of Knowledge	27
6. Instantiating Uses Cases	28
6.1. Rate-limited Access	28
6.2. Client Geo-Location	29
6.3. Private Client Authentication	29
7. Security Considerations	29
7.1. Client Identity	30

7.2.	Denial of Service	30
7.3.	Channel Security	30
8.	Privacy Considerations	30
8.1.	Client Token State and Origin Tracking	30
8.2.	Origin Verification	31
8.3.	Client Identification with Unique Keys	31
8.4.	Collusion Among Different Entities	32
9.	Deployment Considerations	32
9.1.	Origin Key Rollout	32
10.	IANA Considerations	32
10.1.	Authentication Scheme	32
10.2.	HTTP Headers	33
10.3.	Media Types	33
10.3.1.	"message/access-token-request" media type	33
10.3.2.	"message/access-token-response" media type	34
11.	References	35
11.1.	Normative References	35
11.2.	Informative References	36
Appendix A.	Related Work: Privacy Pass	36
Authors' Addresses	37

1. Introduction

Servers commonly use passive and persistent identifiers associated with clients, such as IP addresses or device identifiers, for enforcing access and usage policies. For example, a server might limit the amount of content an IP address can access over a given time period (referred to as a "metered paywall"), or a server might rate-limit access from an IP address to prevent fraud and abuse. Servers also commonly use the client's IP address as a strong indicator of the client's geographic location to limit access to services or content to a specific geographic area (referred to as "geofencing").

However, passive and persistent client identifiers can be used by any entity that has access to it without the client's express consent. A server can use a client's IP address or its device identifier to track client activity. A client's IP address, and therefore its location, is visible to all entities on the path between the client and the server. These entities can trivially track a client, its location, and servers that the client visits.

A client that wishes to keep its IP address private can hide its IP address using a proxy service or a VPN. However, doing so severely limits the client's ability to access services and content, since servers might not be able to enforce their policies without a stable and unique client identifier.

This document describes an architecture for Private Access Tokens (PATs), using RSA Blind Signatures as defined in [BLINDSIG], as an explicit replacement for these passive client identifiers. These tokens are privately issued to clients upon request and then redeemed by servers in such a way that the issuance and redemption events for a given token are unlinkable.

At first glance, using PATs in lieu of passive identifiers for policy enforcement suggests that some entity needs to know both the client's identity and the server's policy, and such an entity would be trivially able to track a client and its activities. However, with appropriate mediation and separation between the parties involved in the issuance and the redemption protocols, it is possible to eliminate this information concentration without any functional regressions. This document describes such a protocol.

The relationship of this work to Privacy Pass ([I-D.ietf-privacypass-protocol]) is discussed in Appendix A.

1.1. Motivation

This section describes classes of use cases where an origin would traditionally use a stable and unique client identifier for enforcing attribute-based policy. Hiding these identifiers from origins would therefore require an alternative for origins to continue enforcing their policies. Using the Privacy Address Token architecture for addressing these use cases is described in Section 6.

1.1.1. Rate-limited Access

An origin provides rate-limited access to content to a client over a fixed period of time. The origin does not need to know the client's identity, but needs to know that a requesting client has not exceeded the maximum rate set by the origin.

One example of this use case is a metered paywall, where an origin limits the number of page requests to each unique user over a period of time before the user is required to pay for access. The origin typically resets this state periodically, say, once per month. For example, an origin may serve ten (major content) requests in a month before a paywall is enacted. Origins may want to differentiate quick refreshes from distinct accesses.

Another example of this use case is rate-limiting page accesses to a client to help prevent fraud. Operations that are sensitive to fraud, such as account creation on a website, often employ rate-limiting as a defense in depth strategy. Captchas or additional verification can be required by these pages when a client exceeds a set rate-limit.

Origins routinely use client IP addresses for this purpose.

1.1.2. Client Geo-Location

An origin provides access to or customizes content based on the geo-location of the client. The origin does not need to know the client's identity, but needs to know the geo-location, with some level of accuracy, for providing service.

A specific example of this use case is "geo-fencing", where an origin restricts the available content it can serve based on the client's geographical region.

Origins almost exclusively use client IP addresses for this purpose.

1.1.3. Private Client Authentication

An origin provides access to content for clients that have been authorized by a delegated or known mediator. The origin does not need to know the client's identity.

A specific example of this use case is a federated service that authorizes users for access to specific sites, such as a federated news service or a federated video streaming service. The origin trusts the federator to authorize users and needs proof that the federator authorized a particular user, but it does not need the user's identity to provide access to content.

Origins could currently redirect clients to a federator for authentication, but origins could then track the client's federator user ID or the client's IP address across accesses.

1.2. Architecture

At a high level, the PAT architecture seeks to solve the following problem: in the absence of a stable Client identifier, an Origin needs to verify the identity of a connecting Client and enforce access policies for the incoming Client. To accomplish this, the PAT architecture employs four functional components:

1. Client: requests a PAT from an Issuer and presents it to a Origin for access to the Origin's service.
2. Mediator: authenticates a Client, using information such as its IP address, an account name, or a device identifier. Anonymizes a Client to an Issuer and relays information between an anonymized Client and an Issuer.
3. Issuer: issues PATs to an anonymized Client on behalf of an Origin. Anonymizes an Origin to a Mediator and enforces the Origin's policy.
4. Origin: directs a Client to an Issuer with a challenge and enables access to content or services to the Client upon verification of any PAT sent in response by the Client.

In the PAT architecture, these four components interact as follows.

An Origin designates a trusted Issuer to issue tokens for it. The Origin then redirects any incoming Clients to the Issuer for policy enforcement, expecting the Client to return with a proof from the Issuer that the Origin's policy has been enforced for this Client.

The Client employs a trusted Mediator through which it communicates with the Issuer for this proof. The Mediator performs three important functions:

- * authenticate and associate the Client with a stable identifier;
- * maintain issuance state for the Client and relay it to the Issuer;
and
- * anonymize the Client and mediate communication between the Client and the Issuer.

When a Mediator-anonymized Client requests a token from an Issuer, the Issuer enforces the Origin's policies based on the received Client issuance state and Origin policy. Issuers know the Origin's policies and enforce them on behalf of the Origin. An example policy is: "Limit 10 accesses per Client". More examples and their use cases are discussed in Section 6. The Issuer does not learn the Client's true identity.

Finally, the Origin provides access to content or services to a Client upon verifying a PAT presented by the Client. Verification of this token serves as proof that the Client meets the Origin's policies as enforced by the delegated Issuer with the help of a Mediator. The Origin can then provide any services or content gated behind these policies to the Client.

Figure 1 shows the components of the PAT architecture described in this document. Protocol details follow in Section 4 and Section 5.

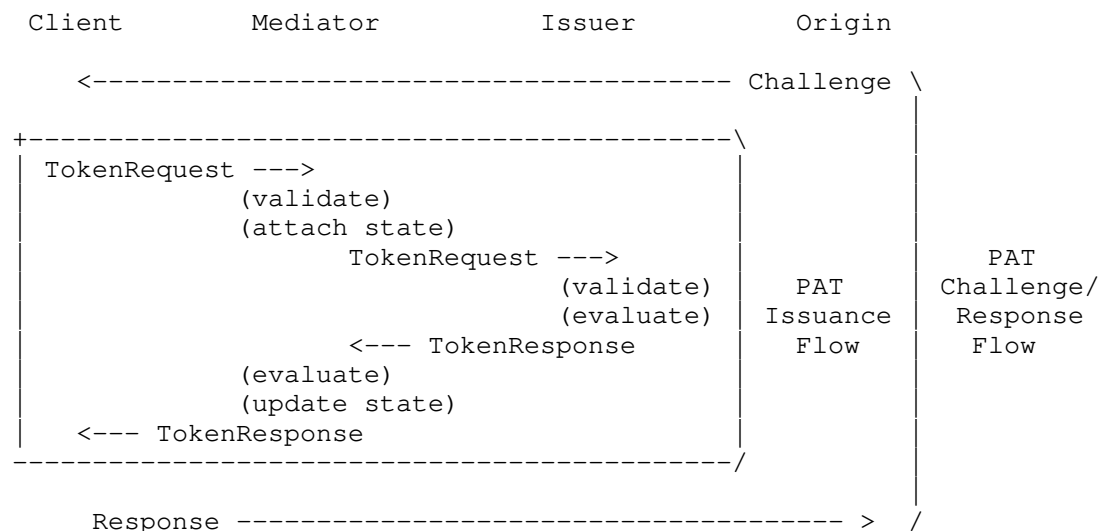


Figure 1: PAT Architectural Components

1.3. Properties and Requirements

In this architecture, the Mediator, Issuer, and Origin each have partial knowledge of the Client's identity and actions, and each entity only knows enough to serve its function (see Section 2 for more about the pieces of information):

- * The Mediator knows the Client's identity and learns the Client's public key (CLIENT_KEY), the Issuer being targeted (ISSUER_NAME), the period of time for which the Issuer's policy is valid (ISSUER_POLICY_WINDOW), and the number of tokens issued to a given Client for the claimed Origin in the given policy window. The Mediator does not know the identity of the Origin the Client is trying to access (ORIGIN_ID), but knows a Client-anonymized identifier for it (ANON_ORIGIN_ID).

- * The Issuer knows the Origin's secret (ORIGIN_SECRET) and policy about client access, and learns the Origin's identity (ORIGIN_NAME) and the number of previous tokens issued to the Client (as communicated by the Mediator) during issuance. The Issuer does not learn the Client's identity.
- * The Origin knows the Issuer to which it will delegate an incoming Client (ISSUER_NAME), and can verify that any tokens presented by the Client were signed by the Issuer. The Origin does not learn which Mediator was used by a Client for issuance.

Since an Issuer enforces policies on behalf of Origins, a Client is required to reveal the Origin's identity to the delegated Issuer. It is a requirement of this architecture that the Mediator not learn the Origin's identity so that, despite knowing the Client's identity, a Mediator cannot track and concentrate information about Client activity.

An Issuer expects a Mediator to verify its Clients' identities correctly, but an Issuer cannot confirm a Mediator's efficacy or the Mediator-Client relationship directly without learning the Client's identity. Similarly, an Origin does not know the Mediator's identity, but ultimately relies on the Mediator to correctly verify or authenticate a Client for the Origin's policies to be correctly enforced. An Issuer therefore chooses to issue tokens to only known and reputable Mediators; the Issuer can employ its own methods to determine the reputation of a Mediator.

A Mediator is expected to employ a stable Client identifier, such as an IP address, a device identifier, or an account at the Mediator, that can serve as a reasonable proxy for a user with some creation and maintenance cost on the user.

For the Issuance protocol, a Client is expected to create and maintain stable and explicit secrets for time periods that are on the scale of Issuer policy windows. Changing these secrets arbitrarily during a policy window can result in token issuance failure for the rest of the policy window; see Section 5.1.1 for more details. A Client can use a service offered by its Mediator or a third-party to store these secrets, but it is a requirement of the PAT architecture that the Mediator not be able to learn these secrets.

The privacy guarantees of the PAT architecture, specifically those around separating the identity of the Client from the names of the Origins that it accesses, are based on the expectation that there is not collusion between the entities that know about Client identity and those that know about Origin identity. Clients choose and share information with Mediators, and Origins choose and share policy with

Issuers; however, the Mediator is generally expected to not be colluding with Issuers or Origins. If this occurs, it can become possible for a Mediator to learn or infer which Origins a Client is accessing, or for an Origin to learn or infer the Client identity. For further discussion, see Section 8.4.

1.4. Client Identity

The PAT architecture does not enforce strong constraints around the definition of a Client identity and allows it to be defined entirely by a Mediator. If a user can create an arbitrary number of Client identities that are accepted by one or more Mediators, a malicious user can easily abuse the system to defeat the Issuer's ability to enforce per-Client policies.

These multiple identities could be fake or true identities.

A Mediator alone is responsible for detecting and weeding out fake Client identities in the PAT architecture. An Issuer relies on a Mediator's reputation; as explained in Section 1.3, the correctness of the architecture hinges on Issuers issuing tokens to only known and reputable Mediators.

Users have multiple true identities on the Internet however, and as a result, it seems possible for a user to abuse the system without having to create fake identities. For instance, a user could use multiple Mediators, authenticating with each one using a different true identity.

The PAT architecture offers no panacea against this potential abuse. We note however that the usages of PATs will cause the ecosystem to evolve and offer practical mitigations, such as:

- * An Issuer can learn the properties of a Mediator - specifically, which stable Client identifier is authenticated by the Mediator - to determine whether the Mediator is acceptable for an Origin.
- * An Origin can choose an Issuer based on the types of Mediators accepted by the Issuer, or the Origin can communicate its constraints to the designated Issuer.
- * An Origin can direct a user to a specific Issuer based on client properties that are visible. For instance, properties that are observable in the HTTP User Agent string.
- * The number of true Mediator-authenticated identities for a user is expected to be small, and therefore likely to be small enough to not matter for certain use cases. For instance, when PATs are

used to prevent fraud by rate-limiting Clients (as described in Section 1.1.1), an Origin might be tolerant of the potential amplification caused by an attacking user's access to multiple true identities with Issuer-trusted Mediators.

1.5. User Interaction

When used in contexts like websites, origin servers that challenge clients for Private Access Tokens need to consider how to optimize their interaction model to ensure a good user experience.

Private Access Tokens are designed to be used without explicit user involvement. Since tokens are only valid for a single origin and in response to a specific challenge, there is no need for a user to manage a limited pool of tokens across origins. The information that is available to an origin upon token redemption is limited to the fact that this is a client that passed a Mediator's checks and has not exceeded the per-origin limit defined by an Issuer. Generally, if a user is willing to use Private Access Tokens with a particular origin (or all origins), there is no need for per-challenge user interaction. Note that the Issuance flow may separately involve user interaction if the Mediator needs to authenticate the Client.

Since tokens are issued using a separate connection through a Mediator to an Issuer, the process of issuance can add user-perceivable latency. Origins SHOULD NOT block useful work on token authentication. Instead, token authentication can be used in similar ways to CAPTCHA validation today, but without the need for user interaction. If issuance is taking a long time, a website could show an indicator that it is waiting, or fall back to another method of user validation.

If an origin is requesting an unexpected number of tokens, such as requesting token authentication more than once for a single website load, it can indicate that the server is not functioning correctly, or is trying to attack or overload the client or issuance servers. In such cases, the client SHOULD ignore redundant token challengers, or else alert the user.

2. Notation and Terminology

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.

Unless said otherwise, this document encodes protocol messages in TLS notation from [TLS13], Section 3.

This draft includes pseudocode that uses the functions and conventions defined in [HPKE].

Encoding an integer to a sequence of bytes in network byte order is described using the function "encode(*n*, *v*)", where "*n*" is the number of bytes and "*v*" is the integer value. The function "len()" returns the length of a sequence of bytes.

The following terms are defined to refer to the different pieces of information passed through the system:

ISSUER_NAME: The Issuer Name identifies which Issuer is able to provide tokens for a Client. The Client sends the Issuer Name to the Mediator so the Mediator know where to forward requests. Each Issuer is associated with a specific **ISSUER_POLICY_WINDOW**.

ISSUER_POLICY_WINDOW: The period over which an Issuer will track access policy, defined in terms of seconds and represented as a uint64. The state that the Mediator keeps for a Client is specific to a policy window. The effective policy window for a specific Client starts when the Client first sends a request associated with an Issuer.

ORIGIN_TOKEN_KEY: The public key used when generating and verifying Private Access Tokens. Each Origin Token Key is unique to a single Origin. The corresponding private key is held by the Issuer.

ISSUER_KEY: The public key used to encrypt values such as **ORIGIN_NAME** in requests from Clients to the Issuer, so that Mediators cannot learn the **ORIGIN_NAME** value. Each **ISSUER_KEY** is used across all requests on the Issuer, for different Origins.

ORIGIN_NAME: The name of the Origin that requests and verifies Private Access Tokens.

ANON_ORIGIN_ID: An identifier that is generated by the Client and marked on requests to the Mediator, which represents a specific Origin anonymously. The Client generates a stable **ANON_ORIGIN_ID** for each **ORIGIN_NAME**, to allow the Mediator to count token access without learning the **ORIGIN_NAME**.

CLIENT_KEY: A public key chosen by the Client and shared only with the Mediator.

CLIENT_SECRET: The secret key used by the Client during token issuance, whose public key (CLIENT_KEY) is shared with the Mediator.

ORIGIN_SECRET: The secret key used by the Issuer during token issuance, whose public key is not shared with anyone.

ANON_ISSUER_ORIGIN_ID: An identifier that is generated by Issuer based on an ORIGIN_SECRET that is per-Client and per-Origin. See Section 5.6 for details of derivation.

3. Configuration

Issuers MUST provide three parameters for configuration:

1. **ISSUER_KEY:** a KeyConfig as defined in [OHTTP] to use when encrypting the ORIGIN_NAME in issuance requests. This parameter uses resource media type "application/ohttp-keys".
2. **ISSUER_POLICY_WINDOW:** a uint64 of seconds as defined in Section 2.
3. **ISSUER_REQUEST_URI:** a Private Access Token request URL for generating access tokens. For example, an Issuer URL might be <https://issuer.example.net/access-token-request>. This parameter uses resource media type "text/plain".

These parameters can be obtained from an Issuer via a directory object, which is a JSON object whose field names and values are raw values and URLs for the parameters.

Field Name	Value
issuer-key	ISSUER_KEY resource URL as a JSON string
issuer-policy-window	ISSUER_POLICY_WINDOW as a JSON number
issuer-request-uri	ISSUER_REQUEST_URI resource URL as a JSON string

Table 1

As an example, the Issuer's JSON directory could look like:

```
{
  "issuer-key": "https://issuer.example.net/key",
  "issuer-token-window": 86400,
  "issuer-request-uri": "https://issuer.example.net/access-token-request"
}
```

Mediators MUST provide a single parameter for configuration, `MEDIATOR_REQUEST_URI`, which is Private Access Token request URL for proxying protocol messages to Issuers. For example, a Mediator URL might be `https://mediator.example.net/relay-access-token-request`. Similar to Issuers, Mediators make this parameter available by a directory object with the following contents:

Field Name	Value
mediator-request-uri	MEDIATOR_REQUEST_URI resource URL

Table 2

As an example, the Mediator's JSON dictionary could look like:

```
{
  "mediator-request-uri": "https://mediator.example.net/relay-access-token-request."
}
```

Issuer and Mediator directory resources have the media type `"application/json"` and are located at the well-known location `/.well-known/private-access-tokens-directory`.

4. Token Challenge and Redemption Protocol

This section describes the interactive protocol for the token challenge and redemption flow between a Client and an Origin.

Token redemption is performed using HTTP Authentication ([RFC7235]), with the scheme `"PrivateAccessToken"`. Origins challenge Clients to present a unique, single-use token from a specific Issuer. Once a Client has received a token from that Issuer, it presents the token to the Origin.

Token redemption only requires Origins to verify token signatures computed using the Blind Signature protocol from [BLINDSIG]. Origins are not required to implement the complete Blind Signature protocol. (In contrast, token issuance requires Clients and Issuers to implement the Blind Signature protocol, as described in Section 5.)

4.1. Token Challenge

Origins send a token challenge to Clients in an "WWW-Authenticate" header with the "PrivateAccessToken" scheme. This challenge includes a TokenChallenge message, along with information about what keys to use when requesting a token from the Issuer.

The TokenChallenge message has the following structure:

```
struct {  
    uint8_t version;  
    opaque origin_name<1..2^16-1>;  
    opaque issuer_name<1..2^16-1>;  
    opaque redemption_nonce[32];  
} TokenChallenge;
```

The structure fields are defined as follows:

- * "version" is a 1-octet integer. This document defines version 1.
- * "origin_name" is a string containing the name of the Origin (ORIGIN_NAME).
- * "issuer_name" is a string containing the name of the Issuer (ISSUER_NAME).
- * "redemption_nonce" is a fresh 32-byte nonce generated for each redemption request.

When used in an authentication challenge, the "PrivateAccessToken" scheme uses the following attributes:

- * "challenge", which contains a base64url-encoded [RFC4648] TokenChallenge value. This MUST be unique for every 401 HTTP response to prevent replay attacks.
- * "token-key", which contains a base64url encoding of the SubjectPublicKeyInfo object for use with the RSA Blind Signature protocol (ORIGIN_TOKEN_KEY).
- * "issuer-key", which contains a base64url encoding of a KeyConfig as defined in [OHTTP] to use when encrypting the ORIGIN_NAME in issuance requests (ISSUER_KEY).
- * "max-age", an optional attribute that consists of the number of seconds for which the challenge will be accepted by the Origin.

Origins MAY also include the standard "realm" attribute, if desired.

As an example, the WWW-Authenticate header could look like this:

```
WWW-Authenticate: PrivateAccessToken challenge=abc..., token-key=123...,  
issuer-key=456...
```

Upon receipt of this challenge, a Client uses the message and keys in the Issuance protocol (see Section 5). If the TokenChallenge has a version field the Client does not recognize or support, it MUST NOT parse or respond to the challenge. This document defines version 1, which indicates use of private tokens based on RSA Blind Signatures [BLINDSIG], and determines the rest of the structure contents.

Note that it is possible for the WWW-Authenticate header to include multiple challenges, in order to allow the Client to fetch a batch of multiple tokens for future use.

For example, the WWW-Authenticate header could look like this:

```
WWW-Authenticate: PrivateAccessToken challenge=abc..., token-key=123...,  
issuer-key=456..., PrivateAccessToken challenge=def..., token-key=234...,  
issuer-key=567...
```

4.2. Token Redemption

The output of the issuance protocol is a token that corresponds to the Origin's challenge (see Section 4.1). A token is a structure that begins with a single byte that indicates a version, which MUST match the version in the TokenChallenge structure.

```
struct {  
    uint8_t version;  
    uint8_t token_key_id[32];  
    uint8_t message[32];  
    uint8_t signature[Nk];  
} Token;
```

The structure fields are defined as follows:

- * "version" is a 1-octet integer. This document defines version 1.
- * "token_key_id" is a collision-resistant hash that identifies the ORIGIN_TOKEN_KEY used to produce the signature. This is generated as SHA256(public_key), where public_key is a DER-encoded SubjectPublicKeyInfo object carrying the public key.
- * "message" is a 32-octet message containing the hash of the original TokenChallenge, SHA256(TokenChallenge). This message is signed by the signature,

- * "signature" is a Nk-octet RSA Blind Signature that covers the message. For version 1, Nk is indicated by size of the Token structure and may be 256, 384, or 512. These correspond to RSA 2048, 3072, and 4096 bit keys. Clients implementing version 1 MUST support signature sizes with Nk of 512 and 256.

When used for client authorization, the "PrivateAccessToken" authentication scheme defines one parameter, "token", which contains the base64url-encoded Token struct. All unknown or unsupported parameters to "PrivateAccessToken" authentication credentials MUST be ignored.

Clients present this Token structure to Origins in a new HTTP request using the Authorization header as follows:

Authorization: PrivateAccessToken token=abc...

Origins verify the token signature using the corresponding policy verification key from the Issuer, and validate that the message matches the hash of a TokenChallenge it previously issued and is still valid, SHA256(TokenChallenge), and that the version of the Token matches the version in the TokenChallenge. The TokenChallenge MAY be bound to a specific HTTP session with Client, but Origins can also accept tokens for valid challenges in new sessions.

If a Client's issuance request fails with a 401 error, as described in Section 5.4, the Client MUST react to the challenge as if it could not produce a valid Authorization response.

5. Issuance Protocol

This section describes the Issuance protocol for a Client to request and receive a token from an Issuer. Token issuance involves a Client, Mediator, and Issuer, with the following steps:

1. The Client sends a token request to the Mediator, encrypted using an Issuer-specific key
2. The Mediator validates the request and proxies the request to the Issuer
3. The Issuer decrypts the request and sends a response back to the Mediator
4. The Mediator verifies the response and proxies the response to the Client

The Issuance protocol has a number of underlying cryptographic dependencies for operation:

- * [HPKE], for encrypting information in transit between Client and Issuer across the Mediator.
- * RSA Blind Signatures [BLINDSIG], for issuing and constructing Tokens as described in Section 4.2.
- * Prime Order Groups (POGs), for computing stable mappings between (Client, Origin) pairs. This document uses notation described in [VOPRF], Section 2.1, and, in particular, the functions `RandomScalar()`, `Generator()`, `SerializeScalar()`, `SerializeElement()`, and `HashToScalar()`.
- * Non-Interactive proof-of-knowledge (POK), as described in Section 5.8, for verifying correctness of Client requests.

Clients and Issuers are required to implement all of these dependencies, whereas Mediators are required to implement POG and POK support.

5.1. State Requirements

The Issuance protocol requires each participating endpoint to maintain some necessary state, as described in this section.

5.1.1. Client State

A Client is required to have the following information, derived from a given `TokenChallenge`:

- * Origin name (`ORIGIN_NAME`), a URI referring to the Origin [RFC6454]. This is the value of `TokenChallenge.origin_name`.
- * Origin token public key (`ORIGIN_TOKEN_KEY`), a blind signature public key corresponding to the Origin identified by `TokenChallenge.origin_name`.
- * Issuer public key (`ISSUER_KEY`), a public key used to encrypt requests corresponding to the Issuer identified by `TokenChallenge.issuer_name`.

Clients maintain a stable `CLIENT_ID` that they use for all communication with a specific Mediator. `CLIENT_ID` is a public key, where the corresponding private key `CLIENT_SECRET` is known only to the client.

If the client loses this (CLIENT_ID, CLIENT_SECRET), they may generate a new tuple. The mediator will enforce if a client is allowed to use this new CLIENT_ID. See #mediator-state for details on this enforcement.

Clients also need to be able to generate an ANON_ORIGIN_ID value that corresponds to the ORIGIN_NAME, to send in requests to the Mediator.

ANON_ORIGIN_ID MUST be a stable and unpredictable 32-byte value computed by the Client. Clients MUST NOT change this value across token requests for the same ORIGIN_NAME. Doing so will result in token issuance failure (specifically, when a Mediator rejects a request upon detecting two ANON_ORIGIN_ID values that map to the same Origin).

One possible mechanism for implementing this identifier is for the Client to store a mapping between the ORIGIN_NAME and a randomly generated ANON_ORIGIN_ID for future requests. Alternatively, the Client can compute a PRF keyed by a per-client secret (CLIENT_SECRET) over the ORIGIN_NAME, e.g., ANON_ORIGIN_ID = HKDF(secret=CLIENT_SECRET, salt="", info=ORIGIN_NAME).

5.1.2. Mediator State

A Mediator is required to maintain state for every authenticated Client. The mechanism of identifying a Client is specific to each Mediator, and is not defined in this document. As examples, the Mediator could use device-specific certificates or account authentication to identify a Client.

Mediators must enforce that Clients don't change their CLIENT_ID frequently, to ensure Clients can't regularly evade the per-client policy as seen by the issuer. Mediators MUST NOT allow Clients to change their CLIENT_ID more than once within a policy window, or in the subsequent policy window after a previous CLIENT_ID change. Alternative schemes where the mediator stores the encrypted (CLIENT_ID, CLIENT_SECRET) tuple on behalf of the client are possible but not described here.

Mediators are expected to know the ISSUER_POLICY_WINDOW for any ISSUER_NAME to which they allow access. This information can be retrieved using the URIs defined in Section 3.

For each Client-Issuer pair, a Mediator maintains a policy window start and end time for each Issuer from which a Client requests a token.

For each tuple of (CLIENT_ID, ANON_ORIGIN_ID, policy window), the Mediator maintains the following state:

- * A counter of successful tokens issued
- * Whether or not a previous request was rejected by the Issuer
- * The last received ANON_ISSUER_ORIGIN_ID value for this ANON_ORIGIN_ID, if any

5.1.3. Issuer State

Issuers maintain a stable ORIGIN_SECRET that they use in calculating values returned to the Mediator for each origin. If this value changes, it will open up a possibility for Clients to request extra tokens for an Origin without being limited, within a policy window.

Issuers are expected to have the private key that corresponds to ISSUER_KEY, which allows them to decrypt the ORIGIN_NAME values in requests.

Issuers also need to know the set of valid ORIGIN_TOKEN_KEY public keys and corresponding private key, for each ORIGIN_NAME that is served by the Issuer. Origins SHOULD update their view of the ORIGIN_TOKEN_KEY regularly to ensure that Client requests do not fail after ORIGIN_TOKEN_KEY rotation.

5.2. Issuance HTTP Headers

The Issuance protocol defines four new HTTP headers that are used in requests and responses between Clients, Mediators, and Issuers (see Section 10.2).

The "Sec-Token-Origin" is an Item Structured Header [RFC8941]. Its value MUST be a Byte Sequence. This header is sent both on Client-to-Mediator requests (Section 5.3) and on Issuer-to-Mediator responses (Section 5.5). Its ABNF is:

Sec-Token-Origin = sf-binary

The "Sec-Token-Client" is an Item Structured Header [RFC8941]. Its value MUST be a Byte Sequence. This header is sent on Client-to-Mediator requests (Section 5.3), and contains the bytes of CLIENT_KEY. Its ABNF is:

Sec-Token-Client = sf-binary

The "Sec-Token-Nonce" is an Item Structured Header [RFC8941]. Its value MUST be a Byte Sequence. This header is sent on Client-to-Mediator requests (Section 5.3), and contains a per-request nonce value. Its ABNF is:

```
Sec-Token-Nonce = sf-binary
```

The "Sec-Token-Count" is an Item Structured Header [RFC8941]. Its value MUST be an Integer. This header is sent on Mediator-to-Issuer requests (Section 5.3), and contains the number of times a Client has previously received a token for an Origin. Its ABNF is:

```
Sec-Token-Count = sf-integer
```

5.3. Client-to-Mediator Request

The Client and Mediator MUST use a secure and Mediator-authenticated HTTPS connection. They MAY use mutual authentication or mechanisms such as TLS certificate pinning, to mitigate the risk of channel compromise; see Section 7 for additional about this channel.

Issuance begins by Clients hashing the TokenChallenge to produce a token input as `message = SHA256(challenge)`, and then blinding message as follows:

```
blinded_req, blind_inv = rsabssa_blind(ORIGIN_TOKEN_KEY, message)
```

The Client MUST use a randomized variant of RSABSSA in producing this signature with a salt length of at least 32 bytes.

The Client uses CLIENT_SECRET to generate proof of its request.

```
blind = RandomScalar()
blind_key = blind * CLIENT_SECRET
blind_generator = blind * Generator()
key_proof = SchnorrProof(CLIENT_SECRET, blind_key, blind_generator)
```

The Client then transforms this proof into "mapping_nonce", "mapping_key", "mapping_generator", and "mapping_proof".

```
mapping_nonce = SerializeScalar(blind)
mapping_key = SerializeElement(blind_key)
mapping_generator = SerializeElement(blind_generator)
mapping_proof = SerializeProof(key_proof)
```

The Client then constructs a Private Access Token request using mapping_key, mapping_generator, mapping_proof, blinded_req, and origin information.

```
struct {  
    uint8_t version;  
    uint8_t mapping_generator[Ne];  
    uint8_t mapping_key[Ne];  
    uint8_t mapping_proof[Np];  
    uint8_t token_key_id;  
    uint8_t blinded_req[Nk];  
    uint8_t name_key_id[32];  
    uint8_t encrypted_origin_name<1..2^16-1>;  
} AccessTokenRequest;
```

The structure fields are defined as follows:

- * "version" is a 1-octet integer, which matches the version in the TokenChallenge. This document defines version 1.
- * "mapping_generator", "mapping_key", and "mapping_proof" are computed as described above.
- * "token_key_id" is the least significant byte of the ORIGIN_TOKEN_KEY key ID, which is generated as $\text{SHA256}(\text{public_key})$, where `public_key` is a DER-encoded SubjectPublicKeyInfo object carrying ORIGIN_TOKEN_KEY.
- * "blinded_req" is the Nk-octet request defined above.
- * "name_key_id" is a collision-resistant hash that identifies the ISSUER_KEY public key, generated as $\text{SHA256}(\text{KeyConfig})$.
- * "encrypted_origin_name" is an encrypted structure that contains ORIGIN_NAME, calculated as described in Section 5.7.

The Client then generates an HTTP POST request to send through the Mediator to the Issuer, with the AccessTokenRequest as the body. The media type for this request is "message/access-token-request". The Client includes the "Sec-Token-Origin" header, whose value is ANON_ORIGIN_ID; the "Sec-Token-Client" header, whose value is CLIENT_KEY; and the "Sec-Token-Nonce" header, whose value is mapping_nonce. The Client sends this request to the Mediator's proxy URI. An example request is shown below, where Nk = 512.

```
:method = POST
:scheme = https
:authority = issuer.net
:path = /access-token-request
accept = message/access-token-response
cache-control = no-cache, no-store
content-type = message/access-token-request
content-length = 512
sec-token-origin = ANON_ORIGIN_ID
sec-token-client = CLIENT_KEY
sec-token-nonce = mapping_nonce
```

<Bytes containing the AccessTokenRequest>

If the Mediator detects a version in the AccessTokenRequest that it does not recognize or support, it MUST reject the request with an HTTP 400 error.

The Mediator also checks to validate that the name_key_id in the client's AccessTokenRequest matches a known ISSUER_KEY public key for the Issuer. For example, the Mediator can fetch this key using the API defined in Section 3. This check is done to help ensure that the Client has not been given a unique key that could allow the Issuer to fingerprint or target the Client. If the key does not match, the Mediator rejects the request with an HTTP 400 error. Note that Mediators need to be careful in cases of key rotation; see Section 8.

The Mediator finally checks to ensure that the AccessTokenRequest.mapping_proof is valid for the given CLIENT_KEY; see Section 5.8 for verification details. If the index is invalid, the Mediator rejects the request with an HTTP 400 error.

If the Mediator accepts the request, it will look up the state stored for this Client. It will look up the count of previously generate tokens for this Client using the same ANON_ORIGIN_ID. See Section 5.1.2 for more details.

If the Mediator has stored state that a previous request for this ANON_ORIGIN_ID was rejected by the Issuer in the current policy window, it SHOULD reject the request without forwarding it to the Issuer.

If the Mediator detects this Client has changed their CLIENT_ID more frequently than allowed as described in #mediator-state, it SHOULD reject the request without forwarding it to the Issuer.

5.4. Mediator-to-Issuer Request

The Mediator and the Issuer MUST use a secure and Issuer-authenticated HTTPS connection. Also, Issuers MUST authenticate Mediators, either via mutual TLS or another form of application-layer authentication. They MAY additionally use mechanisms such as TLS certificate pinning, to mitigate the risk of channel compromise; see Section 7 for additional about this channel.

Before copying and forwarding the Client's AccessTokenRequest request to the Issuer, the Mediator adds a header that includes the count of previous tokens as "Sec-Token-Count". The Mediator MAY also add additional context information, but MUST NOT add information that will uniquely identify a Client.

```
:method = POST
:scheme = https
:authority = issuer.net
:path = /access-token-request
accept = message/access-token-response
cache-control = no-cache, no-store
content-type = message/access-token-request
content-length = 512
sec-token-count = 3
```

<Bytes containing the AccessTokenRequest>

Upon receipt of the forwarded request, the Issuer validates the following conditions:

- * The "Sec-Token-Count" header is present
- * The AccessTokenRequest contains a supported version
- * For version 1, the AccessTokenRequest.name_key_id corresponds to the ID of the ISSUER_KEY held by the Issuer
- * For version 1, the AccessTokenRequest.encrypted_origin_name can be decrypted using the Issuer's private key (the private key associated with ISSUER_KEY), and matches an ORIGIN_NAME that is served by the Issuer
- * For version 1, the AccessTokenRequest.blinded_req is of the correct size
- * For version 1, the AccessTokenRequest.token_key_id corresponds to an ID of an ORIGIN_TOKEN_KEY for the corresponding ORIGIN_NAME

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Mediator, which will forward the error to the client.

If the request is valid, the Issuer then can use the value from "Sec-Token-Count" to determine if the Client is allowed to receive a token for this Origin during the current policy window. If the Issuer refuses to issue more tokens, it responds with an HTTP 429 (Too Many Requests) error to the Mediator, which will forward the error to the client.

The Issuer determines the correct ORIGIN_TOKEN_KEY by using the decrypted ORIGIN_NAME value and AccessTokenRequest.token_key_id. If there is no ORIGIN_TOKEN_KEY whose truncated key ID matches AccessTokenRequest.token_key_id, the Issuer MUST return an HTTP 401 error to Mediator, which will forward the error to the client. The Mediator learns that the client's view of the Origin key was invalid in the process.

5.5. Issuer-to-Mediator Response

If the Issuer is willing to give a token to the Client, the Issuer verifies the token request using "mapping_generator", "mapping_key", and "mapping_proof":

```
valid = SchnorrVerify(mapping_generator, mapping_key, mapping_proof)
```

If this fails, the Issuer rejects the request with a 400 error. Otherwise, the Issuer decrypts AccessTokenRequest.encrypted_origin_name to discover "origin". If this fails, the Issuer rejects the request with a 400 error. The Issuer then evaluates the mapping over the ORIGIN_SECRET pertaining to the origin for this issuer:

```
mapping_input = DeserializeElement(AccessTokenRequest.mapping_key)
index = ORIGIN_SECRET * mapping_input
mapping_index = SerializeElement(index)
```

If DeserializeElement fails, or if AccessTokenRequest.mapping_key is the identity element, the Issuer rejects the request with a 400 error.

The Issuer completes the issuance flow by computing a blinded response as follows:

```
blind_sig = rsabssa_blind_sign(skP, AccessTokenRequest.blinded_req)
```

skP is the private key corresponding to ORIGIN_TOKEN_KEY, known only to the Issuer.

The Issuer generates an HTTP response with status code 200 whose body consists of blind_sig, with the content type set as "message/access-token-response" and the mapping_tag set in the "Sec-Token-Origin" header.

```
:status = 200
content-type = message/access-token-response
content-length = 512
sec-token-origin = mapping_index
```

<Bytes containing the blind_sig>

5.6. Mediator-to-Client Response

Upon receipt of a successful response from the Issuer, the Mediator extracts the "Sec-Token-Origin" header, and uses the value to determine ANON_ISSUER_ORIGIN_ID.

```
index = DeserializeElement(mapping_index)
nonce = DeserializeScalar(mapping_nonce)
ANON_ISSUER_ORIGIN_ID = (nonce^(-1)) * index
```

If the "Sec-Token-Origin" is missing, or if the same ANON_ISSUER_ORIGIN_ID was previously received in a response for a different ANON_ORIGIN_ID within the same policy window, the Mediator MUST drop the token and respond to the client with an HTTP 400 status. If there is not an error, the ANON_ISSUER_ORIGIN_ID is stored alongside the state for the ANON_ORIGIN_ID.

For all other cases, the Mediator forwards all HTTP responses unmodified to the Client as the response to the original request for this issuance.

When the Mediator detects successful token issuance, it MUST increment the counter in its state for the number of tokens issued to the Client for the ANON_ORIGIN_ID.

Upon receipt, the Client handles the response and, if successful, processes the body as follows:

```
sig = rsabssa_finalize(ORIGIN_TOKEN_KEY, nonce, blind_sig, blind_inv)
```

If this succeeds, the Client then constructs a Private Access Token as described in Section 4.1 using the token input message and output sig.

5.7. Encrypting Origin Names

Given a KeyConfig (ISSUER_KEY), Clients produce `encrypted_origin_name` and authenticate all other contents of the AccessTokenRequest using the following values:

- * the key identifier from the configuration, `keyID`, with the corresponding KEM identified by `kemID`, the public key from the configuration, `pkI`, and;
- * a selected combination of KDF, identified by `kdfID`, and AEAD, identified by `aeadID`.

Beyond the key configuration inputs, Clients also require the AccessTokenRequest inputs. Together, these are used to encapsulate `ORIGIN_NAME` (`origin_name`) and produce `ENCRYPTED_ORIGIN_NAME` (`encrypted_origin`) as follows:

1. Compute an [HPKE] context using `pkI`, yielding context and encapsulation key `enc`.
2. Construct associated data, `aad`, by concatenating the values of `keyID`, `kemID`, `kdfID`, `aeadID`, and all other values of the AccessTokenRequest structure.
3. Encrypt (seal) request with `aad` as associated data using context, yielding ciphertext `ct`.
4. Concatenate the values of `aad`, `enc`, and `ct`, yielding an Encapsulated Request `enc_request`.

Note that `enc` is of fixed-length, so there is no ambiguity in parsing this structure.

In pseudocode, this procedure is as follows:


```

enc, context = SetupBaseS(pkI, "AccessTokenRequest")
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID),
             encode(1, version),
             encode(Ne, mapping_generator),
             encode(Ne, mapping_key),
             encode(Np, mapping_proof),
             encode(1, token_key_id),
             encode(Nk, blinded_req),
             encode(32, name_key_id))
ct = context.Seal(aad, origin_name)
encrypted_origin_name = concat(enc, ct)

```

Issuers reverse this procedure to recover `ORIGIN_NAME` by computing the AAD as described above and decrypting `encrypted_origin_name` with their private key `skI`, the private key corresponding to `pkI`. In pseudocode, this procedure is as follows:

```

enc, ct = parse(encrypted_origin_name)
aad = concat(encode(1, keyID),
             encode(2, kemID),
             encode(2, kdfID),
             encode(2, aeadID),
             encode(1, version),
             encode(Ne, mapping_generator),
             encode(Ne, mapping_key),
             encode(Np, mapping_proof),
             encode(1, token_key_id),
             encode(Nk, blinded_req),
             encode(32, name_key_id))
enc, context = SetupBaseR(enc, skI, "AccessTokenRequest")
origin_name, error = context.Open(aad, ct)

```

5.8. Non-Interactive Schnorr Proof of Knowledge

Each Issuance request requires evaluation and verification of a Schnorr proof-of-knowledge. Given input secret "`secret`" and two elements, "`base`" and "`target`", generation of this proof (`u`, `c`, `z`), denoted `SchnorrProof(secret, base, target)`, works as follows:

```

r = RandomScalar()
u = r * base
c = HashToScalar(SerializeElement(base) ||
                  SerializeElement(target) ||
                  SerializeElement(mask),
                  dst = "PrivateAccessTokensProof")
z = r + (c * secret)

```

The proof is encoded by serializing (u, c, z) as follows:

```

struct {
    uint8_t u[Ne];
    uint8_t c[Ns];
    uint8_t z[Ns];
} Proof;

```

The size of this structure is $N_p = N_e + 2*N_s$ bytes.

Verification of a proof (u, c, z), denoted SchnorrVerify(base, target, proof), works as follows:

```

c = HashToScalar(SerializeElement(base) ||
                  SerializeElement(target) ||
                  SerializeElement(mask),
                  dst = "PrivateAccessTokensProof")
expected_left = base * z
expected_right = u + (target * c)

```

The proof is considered valid if expected_left is the same as expected_right.

6. Instantiating Uses Cases

This section describes various instantiations of this protocol to address use cases described in Section 1.1.

6.1. Rate-limited Access

To instantiate this case, the site acts as an Origin and registers a "bounded token" policy with the Issuer. In this policy, the Issuer enforces a fixed number of tokens that it will allow a Client to request for a single ORIGIN_NAME.

Origins request tokens from Clients and, upon successful redemption, the Origin knows the Client was able to request a token for the given ORIGIN_NAME within its budget. Failure to present a token can be interpreted as a signal that the client's token budget was exceeded.

Clients can redeem a token from a specific challenge up to the max-age in the challenge. Servers can choose to issue many challenges in a single HTTP 401 response, providing the client with many challenge nonces which can be used to redeem tokens over a longer period of time.

6.2. Client Geo-Location

To instantiate this use case, the Issuer has an issuing key pair per geographic region, i.e., each region has a unique policy key. When verifying the key for the Client request, the Mediator obtains the per-region key from the Issuer based on the Client's perceived location. During issuance, the Mediator checks that this key matches that of the Client's request. If it matches, the Mediator forwards the request to complete issuance. The number of key pairs is then the cross product of the number of Origins that require per-region keys and the number of regions.

During redemption, Clients present their geographic location to Origins in a "Sec-CH-Geohash" header. Origins use this to obtain the appropriate policy verification key. Origins request tokens from Clients and, upon successful redemption, the Origin knows the Client obtained a token for the given ORIGIN_NAME in the specified region.

6.3. Private Client Authentication

To instantiate this case, the site acts as an Origin and registers an "unlimited token" policy with the Issuer. In this policy, the Issuer does not enforce any limit on the number of tokens a given user will obtain.

Origins request tokens from Clients and, upon successful redemption, the Origin knows the Client was able to request a token for the given ORIGIN_NAME tuple. As a result, the Origin knows this is an authentic client.

7. Security Considerations

This section discusses security considerations for the protocol.

[OPEN ISSUE: discuss trust model]

7.1. Client Identity

The HTTPS connection between Client and Mediator is minimally Mediator-authenticated. Mediators can also require Client authentication if they wish to restrict Private Access Token proxying to trusted or otherwise authenticated Clients. Absent some form of Client authentication, Mediators can use other per-Client information for the client identifier mapping, such as IP addresses.

7.2. Denial of Service

Requesting and verifying a Private Access Token is more expensive than checking an implicit signal, such as an IP address, especially since malicious clients can generate garbage Private Access Tokens and for Origins to work. However, similar DoS vectors already exist for Origins, e.g., at the underlying TLS layer.

7.3. Channel Security

An attacker that can act as an intermediate between Mediator and Issuer communication can influence or disrupt the ability for the Issuer to correctly rate-limit token issuance. All communication channels use server-authenticated HTTPS. Some connections, e.g., between a Mediator and an Issuer, require mutual authentication between both endpoints. Where appropriate, endpoints MAY use further enhancements such as TLS certificate pinning to mitigate the risk of channel compromise.

An attacker that can intermediate the channel between Client and Origin can observe a TokenChallenge, and can view a Token being presented for authentication to an Origin. Scoping the TokenChallenge nonce to the Client HTTP session prevents Tokens being collected in one session and then presented to the Origin in another. Note that an Origin cannot distinguish between a connection to a single Client and a connection to an attacker intermediating multiple Clients. Thus, it is possible for an attacker to collect and later present Tokens from multiple clients over the same Origin session.

8. Privacy Considerations

8.1. Client Token State and Origin Tracking

Origins SHOULD only generate token challenges based on client action, such as when a user loads a website. Clients SHOULD ignore token challenges if an Origin tries to force the client to present tokens multiple times without any new client-initiated action. Failure to do so can allow malicious origins to track clients across contexts. Specifically, an origin can abuse per-user token limits for tracking

by assigning each new client a random token count and observing whether or not the client can successfully redeem that many tokens in a given context. If any token redemption fails, then the origin learns information about how many tokens that client had previously been issued.

By rejecting repeated or duplicative challenges within a single context, the origin only learns a single bit of information: whether or not the client had any token quota left in the given policy window.

8.2. Origin Verification

Private Access Tokens are defined in terms of a Client authenticating to an Origin, where the "origin" is used as defined in [RFC6454]. In order to limit cross-origin correlation, Clients MUST verify that the origin_name presented in the TokenChallenge structure (Section 4.1) matches the origin that is providing the HTTP authentication challenge, where the matching logic is defined for same-origin policies in [RFC6454]. Clients MAY further limit which authentication challenges they are willing to respond to, for example by only accepting challenges when the origin is a web site to which the user navigated.

8.3. Client Identification with Unique Keys

Client activity could be linked if an Origin and Issuer collude to have unique keys targeted at specific Clients or sets of Clients.

To mitigate the risk of a targeted ISSUER_KEY, the Mediator can observe and validate the name_key_id presented by the Client to the Issuer. As described in Section 5, Mediators MUST validate that the name_key_id in the Client's AccessTokenRequest matches a known public key for the Issuer. The Mediator needs to support key rotation, but ought to disallow very rapid key changes, which could indicate that an Origin is colluding with an Issuer to try to rotate the key for each new Client in order to link the client activity.

To mitigate the risk of a targeted ORIGIN_TOKEN_KEY, the protocol expects that an Issuer has only a single valid public key for signing tokens at a time. The Client does not present the name_key_id of the token public key to the Issuer, but instead expects the Issuer to infer the correct key based on the information the Issuer knows, specifically the origin_name itself.

8.4. Collusion Among Different Entities

Collusion among the different entities in the PAT architecture can result in violation of the Client's privacy.

Issuers and Mediators should be run by mutually distinct organizations to limit information sharing. A single entity running an issuer and mediator for a single redemption can view the origins being accessed by a given client. Running the issuer and mediator in this 'single issuer/mediator' fashion reduces the privacy promises to those of the [I-D.ietf-privacypass-protocol]; see Appendix A for more discussion. This may be desirable for a redemption flow that is limited to specific issuers and mediators, but should be avoided where hiding origins from the mediator is desirable.

If a Mediator and Origin are able to collude, they can correlate a client's identity and origin access patterns through timestamp correlation. The timing of a request to an Origin and subsequent token issuance to a Mediator can reveal the Client identity (as known to the Mediator) to the Origin, especially if repeated over multiple accesses.

9. Deployment Considerations

9.1. Origin Key Rollout

Issuers SHOULD generate a new (ORIGIN_TOKEN_KEY, ORIGIN_SECRET) regularly, and SHOULD maintain old and new secrets to allow for graceful updates. The RECOMMENDED rotation interval is two times the length of the policy window for that information. During generation, issuers must ensure the token_key_id (the 8-bit prefix of SHA256(ORIGIN_TOKEN_KEY)) is different from all other token_key_id values for that origin currently in rotation. One way to ensure this uniqueness is via rejection sampling, where a new key is generated until its token_key_id is unique among all currently in rotation for the origin.

10. IANA Considerations

10.1. Authentication Scheme

This document registers the "PrivateAccessToken" authentication scheme in the "Hypertext Transfer Protocol (HTTP) Authentication Scheme Registry" established by [RFC7235].

Authentication Scheme Name: PrivateAccessToken

Pointer to specification text: Section 4.1 of this document

10.2. HTTP Headers

This document registers four new headers for use on the token issuance path in the "Permanent Message Header Field Names" <<https://www.iana.org/assignments/message-headers>>.

Header Field Name	Protocol	Status	Reference
Sec-Token-Origin	http	std	This document
Sec-Token-Client	http	std	This document
Sec-Token-Nonce	http	std	This document
Sec-Token-Count	http	std	This document

Figure 2: Registered HTTP Header

10.3. Media Types

This specification defines the following protocol messages, along with their corresponding media types:

* AccessTokenRequest Section 5: "message/access-token-request"

* AccessTokenResponse Section 5: "message/access-token-response"

The definition for each media type is in the following subsections.

10.3.1. "message/access-token-request" media type

Type name: message

Subtype name: access-token-request

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

10.3.2. "message/access-token-response" media type

Type name: message

Subtype name: access-token-response

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 5

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information: Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

11. References

11.1. Normative References

- [BLINDSIG] Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", Work in Progress, Internet-Draft, draft-irtf-cfrg-rsa-blind-signatures-02, 2 August 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-rsa-blind-signatures-02>>.
- [HPKE] Barnes, R. L., Bhargavan, K., Lipp, B., and C. A. Wood, "Hybrid Public Key Encryption", Work in Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2 September 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hpke-12>>.
- [OHTTP] Thomson, M. and C. A. Wood, "Oblivious HTTP", Work in Progress, Internet-Draft, draft-thomson-http-oblivious-02, 24 August 2021, <<https://datatracker.ietf.org/doc/html/draft-thomson-http-oblivious-02>>.
- [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>>.
- [RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <<https://www.rfc-editor.org/rfc/rfc4648>>.
- [RFC6454] Barth, A., "The Web Origin Concept", RFC 6454, DOI 10.17487/RFC6454, December 2011, <<https://www.rfc-editor.org/rfc/rfc6454>>.

- [RFC7235] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Authentication", RFC 7235, DOI 10.17487/RFC7235, June 2014, <<https://www.rfc-editor.org/rfc/rfc7235>>.
- [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>>.
- [RFC8941] Nottingham, M. and P-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <<https://www.rfc-editor.org/rfc/rfc8941>>.
- [TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/rfc/rfc8446>>.
- [VOPRF] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-08, 25 October 2021, <<https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-08>>.

11.2. Informative References

- [I-D.ietf-privacypass-protocol] Celi, S., Davidson, A., and A. Faz-Hernandez, "Privacy Pass Protocol Specification", Work in Progress, Internet-Draft, draft-ietf-privacypass-protocol-01, 22 February 2021, <<https://datatracker.ietf.org/doc/html/draft-ietf-privacypass-protocol-01>>.

Appendix A. Related Work: Privacy Pass

Private Access Tokens has many similarities to the existing Privacy Pass protocol ([I-D.ietf-privacypass-protocol]). Both protocols allow clients to redeem signed tokens while not allowing linking between token issuance and token redemption.

There are several important differences between the protocols, however:

- * Private Access Tokens uses per-origin tokens that support rate-limiting policies. Each token can only be used with a specific origin in accordance with a policy defined for that origin. This allows origins to implement metered paywalls or mechanisms that limit the actions a single client can perform. Per-origin

tokens also ensure that one origin cannot consume all of a client's tokens, so there is less need for clients to manage when they are willing to present tokens to origins.

- * Private Access Tokens employ an online challenge (Section 4.1) during token redemption. This ensures that tokens cannot be harvested and stored for use later. This also removes the need for preventing double spending or employing token expiry techniques, such as frequent signer rotation or expiry-encoded public metadata.
- * Private Access Tokens use a publically verifiable signature [BLINDSIG] to optimize token verification at the origin by avoiding a round trip to the issuer/mediator.

Authors' Addresses

Scott Hendrickson
Google LLC

Email: scott@shendrickson.com

Jana Iyengar
Fastly

Email: jri@fastly.com

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014,
United States of America

Email: tpauly@apple.com

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net