CFRG
Internet-Draft
Intended status: Informational
Expires: January 1, 2018

B. Ford
N. Gailly
L. Gasser
P. Jovanovic
EPFL

June 30, 2017

Collective Edwards-Curve Digital Signature Algorithm
draft-ford-cfrg-cosi-00

Abstract

Collective signatures are compact cryptographic proofs showing that several distinct secret key holders, called cosigners, have cooperated to sign a given message. This document describes a collective signature extension to the EdDSA signing schemes for the Ed25519 and Ed448 elliptic curves. A collective EdDSA signature consists of a point R, a scalar s, and a bitmask Z indicating the specific subset of a known group of cosigners that produced this signature. A collective signature produced by n cosigners is of size 64+ceil(n/8) bytes for Ed25519 and 114+ceil(n/8) bytes for Ed448, respectively, instead of 64n and 114n bytes for n individual signatures. Further, collective signature verification requires only one double scalar multiplication rather than n. The verifier learns exactly which subset of the cosigners participated, enabling the verifier to implement flexible acceptance-threshold policies, and preserving transparency and accountability in the event a bad message is collectively signed.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

A conventional digital signature on some statement S is produced by the holder of a secret key k, and may be verified by anyone against the signer's corresponding public key K. An attacker who successfully steals or compromises the secret key k gains unrestricted ability to impersonate and "sign for" the key-holder. In security-critical contexts it is thus often desirable to divide trust and signing capabilities across several parties. For example, some threshold t out of n known parties may be required to sign a message before verifiers consider it acceptable. A cryptographic proof that multiple parties have cooperated to sign a message is generally known as a multisignature.

One form of multisignature is simply a list of individual signatures, which the verifier must check against a given policy. For example, in a 2-of-3 group defined by three public keys, a multisignature is simply a list of two individual signatures, which the verifier must ensure were produced by the holders of any two distinct public keys in the group. Multisignatures of this kind are well-established in many contexts, such as Bitcoin multisignature wallets [BITCOIN], and are practical when the group of signers is small.

Another form of multisignatures is based on threshold cryptography that uses mechanisms like Shamir secret sharing [SHAMIR] enabling any threshold t-of-n group members to create a constant-size signature that reveals nothing about which particular set of t members signed. This approach simplifies verification and is desirable when the specific set of cosigners is irrelevant or privacy-sensitive. Secret sharing based multisignatures are inappropriate when transparency is required, though, because t colluding members can potentially sign a bad message but then (individually) deny involvement once the compromise is discovered. Moreover, threshold signature schemes usually do not scale well for larger numbers of n.

Collective signatures are compact multisignatures that convey the same information as a list of individual signatures and thereby offer the same transparency, but, at the same time, are comparable in size and verification cost to an individual signature. Group members need not coordinate for the creation of their key-pairs beyond selecting a common elliptic curve, and verifiers can apply flexible acceptance policies beyond simple t-of-n thresholds. Generating collective signatures requires cooperation, but can be done efficiently at with thousands of participants using a tree-aggregation mechanisms as done in the collective signing (CoSi) protocol [COSI].

2. Scope

This document does not attempt to describe CoSi in the context of any particular Internet protocol; instead it describes an abstract protocol that can be easily fitted to a particular application. For example, the specific format of messages is not specified. These issues are left to the protocol implementor to decide.

3. Notations and Conventions

The following notation is used throughout the document:

- p: Prime number.
- GF(p): Finite field with p elements.
- a || b: Concatenation of (bit-)string a with (bit-) string b.
- a + b mod p: Addition of integers a and b modulo prime p.
- a * b mod p: Multiplications of integers a and b modulo prime p.
- B: Generator of the group or subgroup of interest.
- L: Order of the group generated by B.
- I: Neutral element of the group generated by B.
- X + Y: Addition of group elements X and Y.
- [a]X: Addition of X to itself a times (scalar multiplication).
- Aggregation either refers to the addition of two group elements X and Y or to the addition of two scalars a and b.

CoSi uses the parameters of the elliptic curves Curve25519 and Curve448 defined in Sections 4.1 and 4.2 of [RFC7748], respectively.

Encoding and decoding of integers is done as specified in Sections 5.1.2 and 5.1.3 of [RFC8032], respectively.

4. Collective Signing

The collective signing (CoSi) algorithm is an aggregate signature scheme based on Schnorr signatures and the EdDSA signing procedure. CoSi signatures are non-deterministic though as they include random participant commitments and a bitmask identifying participants that have not contributed to the signature generation. This section first presents the collective key setup mechanism, the abstract signature generation algorithm and finally the signature verification procedure.

4.1. Collective Public Key Setup

Let N denote the list of participants. First, each participant i of N generates his longterm private-public key pair (a_i, A_i) as in EdDSA, see Section 5.1.5 of RFC8032 [1]. Afterwards, given a list of public keys A_1, ..., A_n, the collective public key is specified as A = A_1 + ... + A_n.

4.2. Signature Generation

This section presents the collective signature generation scheme.

The inputs of the signature process are:

- o A collective public key A generated from the public keys of participants N.
- o A subset of participants M of N who actively participate in the signature creation. The size of M is denoted by m.
- o A statement (or message) S.

The signature is generated as follow:

1. For each participant i in M, generate a random secret r_i by hashing 32 bytes of cryptographically secure random data. For efficiency, reduce each r_i mod L. Each r_i MUST be re-generated until it is different from 0 mod L or 1 mod L.
2. Compute the integer addition r of all r_i: r = SUM_{i in M}(r_i).
3. Compute the encoding of the fixed-base scalar multiplication [r]B and call the result R.

4. Compute SHA512(R || A || S) and interpret the 64-byte digest as an integer c mod L.

5. For each participant i in M, compute the response s_i = (r_i + c * a_i) mod L.

6. Compute the integer addition s of all s_i: s = SUM_{i in M}(s_i).

7. Initialize a bitmask Z of length n to all zero. For each participant i who is present in N but not in M set the i-th bit of Z to 1, i.e., Z[i] = 1.

8. The signature is the concatenation of the encoded point R, the integer s, and the bitmask Z, denoted as sig = R || s || Z.

4.3. Signature Verification

The inputs to the signature verification process are:

- o A list of public keys A_i of all participants i in N.
- o The collective public key A.
- o The statement S.
- o The signature sig = R || s || Z.
- o A signature policy which is a function that takes a bitmask as an input and returns true or false. For example, a basic signature policy might require that a certain threshold of participants took part in the generation of the collective signature.

A signature is considered valid if the verification process finishes each of the steps below successfully.

1. Split sig into two 32-byte sequences R and s and a bitmask Z. Interpret R as a point on the used elliptic curve and check that it fulfills the curve equation. Interpret s as an unsigned integer and verify that it is non-zero and smaller than L. Verify that Z has length n. If any of the mentioned checks fails, abort the verification process and return false.

2. Check Z against the signature policy. If the policy does not hold, abort the verification process and return false.

3. Compute SHA512(R || A || S) and interpret the 64-byte digest as an integer c.

4. Initialize a new elliptic curve point T = I. For each bit i in the bitmask that is equal to 1, add the corresponding public key A_i to the point T. Formally, T = SUM_{i in N, Z[i] == 1}(A_i) for all i set to 1 in the bitmask.

5. Compute the reduced public key A' = A - T.

6. Check if the group equation [8][s]B = [8]R + [8][c]A' holds.

5. Collective Signing Protocol

This section introduces the distributed CoSi protocol with n participants. For simplicity, we assume there is a designated leader who is responsible for collecting the shares and generating the signature. This leader could be any of the signers and is not trusted in any way. All participants are communicating through a reliable channel with the leader.

5.1. Collective Signature

The leader must know the statement S to be signed and the set of public keys of the participants N. The point A is defined as the collective key of the participants N. A collective signature is generated in four steps over two round trips between the leader and the rest of the participants.

5.1.1. Announcement

Upon the request to generate a signature on a statement S, the leader broadcasts an announcement message indicating the start of a signing process. It is up to the implementation to decide whether to send S itself during that phase or not.

5.1.2. Commitment

Upon the receipt of an announcement message or if the participant is the leader, each participant i generates a random secret r_i by hashing 32 bytes of cryptographically secure random data. Each r_i MUST be re-generated until it is different from 0 mod L or 1 mod L. Each participants then constructs the commitment R_i as the encoding of [r_i]B, sends R_i to the leader and stores the generated r_i for usage in the response phase. If the participant is the leader, it executes the challenge step.

5.1.3. Challenge

The leader waits to receive the commitments R_i from the other participants for a certain time frame as defined by the application. After the timeout, the leader constructs the subset M of participants from whom he has received a commitment R_i and computes the sum R = SUM_{i in M}(R_i). The leader then computes SHA512(R || A || M) and interprets the resulting 64-byte digest as an integer c mod L. The leader broadcasts c to all participants.

5.1.4. Response

Upon reception of c or if the participant is the leader, each participant generates his response s_i = (r_i + c * a_i) mod L. Each non-leader participant sends his s_i to the leader. If the participant is the leader, he executes the signature generation step.

5.1.5. Signature Generation

The leader waits to receive the responses s_i from the other participants for a certain time frame as defined by the application. After the timeout, the leader checks if he received responses from all participants in M and if not he MUST abort the protocol. The leader then computes the aggregate response s = SUM{i in M}(s_i) mod L and initializes a bitmask Z of size n to all zero. For each participant i who is present in N but not in M the leader sets the i-th bit of Z to 1, i.e., Z[i] = 1. The leader then forms the signature sig as the concatenation of the byte-encoded point R, the byte-encoded scalar s, and the bitmask Z. The resulting signature is of the form sig = R || s || Z and MUST be of length 32 + 32 + ceil(n/8) bytes.

5.2. Collective Verification

The verification process is the same as defined in the Section "Signature Verification" above.

6. Tree-based CoSi Protocol

This section presents the CoSi protocol using a tree-shaped network communication overlay. While the core protocol stays the same, the tree-shaped communication enables CoSi to handle large numbers of participants during signature generation efficiently.

6.1. CoSi Tree

Any tree used by CoSi SHOULD be a complete tree for performance reasons, i.e., every level except possible the last one of the tree MUST be filled. The leader is the root node of the tree and is responsible for creating the tree. An intermediate node is a node who has one parent node and at least one child node. A leaf node is a node who has only one parent and no child nodes.

We define the BROADCAST operation as:

- o The leader multicasts a message to his direct child nodes.
- o Upon reception of a message, each node stores the message and multicasts it further down to its children node, except if the node is a leaf.

The internal representation of the tree, and its propagation to the participants is left to the application.

6.2. Collective Signature

The leader must know the statement S, the set N of the participants and their public keys, and the subset M of active participants. The actual communication tree T is created from the subset M, and MUST contain all participants of M. The point A is defined as the collective key of the set P.

6.2.1. Announcement

The leader BROADCASTS an announcement message. Upon reception, each leaf node executes the commitment step.

6.2.2. Commitment

Every node must generate a random commitment R_i as described in the previous commitment section [...]. Each leaf node directly sends its commitment R_i to its parent node. Each non-leaf node generates a bit mask Z_i of n bits initialized with all 0 bits and starts waiting for a commitment and a bit mask from each of its children. After the timeout defined by the application, each node aggregates all its children’s commitments R_i received using point addition formulas, adds its own commitment and stores the result in R’. For every absent commitment from a child at index j in N, the node sets the j-th of its bit mask Z_i to 1. The node also performs an OR operation between all the received bitmasks from its children and its own bit mask, and let the result be B’.

// XXX Should we reject invalid messages, like too-long-bitmask or so? // XXX Bitmasks should be signed and checked? If the node is an intermediate node, it sends the aggregated commitment R’ alongside with the Z’ bitmask to its parents. If the node is the root node, it executes the challenge step.

// XXX What happens when a node does not receive any commitment from a child node. Does it contact the sub-nodes?

6.2.3. Challenge

The leader computes the challenge c = H(R’ || A || S) and BROADCASTS it down the tree. The leader also saves the bitmask Z’ computed in the previous step. Upon reception, each leaf node executes the response step.

6.2.4. Response

Each node generates its response s_i as defined in XXX Response XXX. Each leaf node sends its response to their parent and is allowed to leave the protocol.
Each other node starts waiting for the responses of its children.

XXX HOW to signal / abort? Is it application dependent also? What happens if the root times out?

For each response s received in node i from node’s children j, the node i SHOULD perform a verification of the partial response. Let t be the sub-tree with the node j at the root, and D the aggregation of all the public keys of the participants in t. Let V be the aggregation of all commitments generated by all participants in t. If the equation [8][s]B = [8]V + [8][c]D does not hold, then the node i MUST abort the protocol.

After the timeout occurs, if at least one child’s response is missing, the node MUST signal the leader to abort the protocol. Otherwise, each intermediate node aggregates all its children’s responses, adds its own response s_i, using scalar addition formulas and sends the resulting scalar s’ up to its parent. Each intermediate node can now leave the protocol.

When the root node receives all the responses s’ from its children, it can generate the signature.

6.2.5. Signature Generation

The generation procedure is exactly the same as in the XXX Generation XXX section above.

6.3. Verification

The verification procedure is exactly the same as in the XXX Verify XXX section above.

7. Message Format

All packets exchanged during a CoSi protocol's instance MUST be encoded using Google's Protobuf technology [PROTOBUF]. All packets for a CoSi protocol must be encoded inside the CoSiPacket message format. The "phase" field indicates which message is encoded in the packet. The CoSi packet message contains a "phase" field which is set accordingly to the current phase of the protocol: + Announcement = 1 + Commitment = 2 + Challenge = 3 + Response = 4

```
message CoSiPacket {
  // Announcement = 1, Commitment = 2, Challenge = 3, Response = 4
  required uint32 phase = 1;
  optional Announcement ann = 2;
  optional Commitment comm = 3;
  optional Challenge chal = 4;
  optional Response resp = 5;
}
```

7.1. Announcement

The Announcement message notifies participants of the beginning of a CoSi round. Implementations can extent the message specifications to include the message to sign. That way, participants can refuse to vote at this step by not replying with a commitment. This do not cause any restart of the protocol later.

```
message Announcement {
}
```

7.2. Commitment

The commitment message includes the aggregated commitment as well as the bitmask if the tree based CoSi protocol is used.

```
message Commitment {
  // aggregated commitment R'
  required bytes comm = 1;
  // bitmask B'
  optional bytes mask = 2;
}
```

7.3. Challenge

The challenge message includes the challenge computed by the leader of the CoSi protocol.

```
message Challenge {
  // commputed challenge c
  required bytes chall = 1;
}
```

7.4. Response

The response message includes the aggregated response to be sent to the leader.

```
message Response {
  // aggregated response s'
  required bytes resp = 1;
}
```

8. Security Considerations

8.1. General Implementations Checks

The checks described throughout the different protocols MUST be enforced. Namely that includes: + the random component r MUST conform to r != 0 mod L and r != 1 mod L. + the resulting signature s MUST conform to s != 0 mod L during signature generation + the signature s MUST conform to 0 < s < L + the intermediate signature at each level of the tree MUST be verifiable correctly as described in section the Response step in section XXX

8.2. Random Number Generation

CoSi requires a cryptographically secure pseudorandom number generator (PRNG) for the generation of the private key and the seed to get the random integer r. In most cases, the operating system provides an appropriate facility such as /dev/urandom, which should be used absent other (performance) concerns. It is generally preferable to use an existing PRNG implementation in preference to crafting a new one, and many adequate cryptographic libraries are

already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values. The hashing of the seed provides an additional layer of security regardless of the security of the PRNG.

8.3. Group Membership

Elements should be checked for group membership: failure to properly validate group elements can lead to attacks. In particular it is essential to verify that received points are valid compressions of points on an elliptic curve when using elliptic curves.

8.4. Multiplication by Cofactor in Verification

The given verification formulas multiply points by the cofactor. While this is not strictly necessary for security (in fact, any signature that meets the non-multiplied equation will satisfy the multiplied one), in some applications it is undesirable for implementations to disagree about the exact set of valid signatures.

8.5. Related-Key Attacks

Before any CoSi round happens, all the participants MUST have the list of public keys of the whole set of participants, including a self signature for each public key. This list MUST be generated before any round. If it it not the case, an attacker can craft a special public key which has the effect of eliminating the contribution of a specific participant to the signature.

8.6. Availability

The participating servers should be highly available and should be operated by reputable and competent organizations so the risk a of DDOS attack by un-reliable participants is greatly diminished. In case of failures before the Challenge phase, the leader might abort the protocol if the threshold of present participants is too low.

If a participant detects one of its children in the tree as missing, a simple mechanism is to return an error which propagates back up the tree to the leader. The leader can then restart the round accounting for this missing participant in the bitmask B described in the Commitment section XXX.

9. Discussions

9.1. Hashing the Public Keys in the commitment

Either do H(R || A || msg) with A being the collective public key OR do H(R || SUM(X_i) || msg) where SUM(X_i) is the sum of all public keys that participated in the collective signature,i.e. the aggregation of all keys in the active participant subset Q.

9.2. Hashing the bitmask in the commitment

To truely bind one signature to a set of signers, the bitmask can be included in the challenge computation such like H(R || A || bitmask || msg). The signature verification process could detect any modifications of the original signature before proceeding the computationally expensive process.

9.3. Exception Mechanism

XXX What to do in case a node goes offline, doesn’t sign, or doesn’t relay up etc. in the tree approach.

10. Acknowledgements

Many parts of this document were inspired by RFC8032 on EdDSA.

11. References

11.1. URIs

[1] https://tools.ietf.org/html/rfc8032#page-13

Authors’ Addresses

Bryan Ford
EPFL
BC 210, Station 14
Lausanne CH-1015
Switzerland

Phone: +41 21 693 28 73
Email: bryan.ford@epfl.ch

Nicolas Gailly
EPFL
BC 263, Station 14
Lausanne CH-1015
Switzerland

Phone: +41 21 69 36613
Email: nicolas.gailly@epfl.ch

Linus Gasser
EPFL
BC 208, Station 14
Lausanne CH-1015
Switzerland

Phone: +41 21 69 36770
Email: linus.gasser@epfl.ch

Philipp Jovanovic
EPFL
BC 263, Station 14
Lausanne CH-1015
Switzerland

Phone: +41 21 69 36628
Email: philipp.jovanovic@epfl.ch

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: January 1, 2018

S. Goldberg
Boston University
D. Papadopoulos
University of Maryland
J. Vcelak
NS1

June 30, 2017

Verifiable Random Functions (VRFs)
draft-goldbe-vrf-01

Abstract

A Verifiable Random Function (VRF) is the public-key version of a keyed cryptographic hash. Only the holder of the private key can compute the hash, but anyone with public key can verify the correctness of the hash. VRFs are useful for preventing enumeration of hash-based data structures. This document specifies several VRF constructions that are secure in the cryptographic random oracle model. One VRF uses RSA and the other VRF uses Eliptic Curves (EC).

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 http://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 January 1, 2018.

Copyright Notice

Table of Contents

1. Introduction

1.1. Rationale

A Verifiable Random Function (VRF) [MRV99] is the public-key version of a keyed cryptographic hash. Only the holder of the private VRF key can compute the hash, but anyone with corresponding public key can verify the correctness of the hash.

A key application of the VRF is to provide privacy against offline enumeration (e.g. dictionary attacks) on data stored in a hash-based data structure. In this application, a Prover holds the VRF secret key and uses the VRF hashing to construct a hash-based data structure on the input data. Due to the nature of the VRF, only the Prover can answer queries about whether or not some data is stored in the data structure. Anyone who knows the public VRF key can verify that the Prover has answered the queries correctly. However no offline inferences (i.e. inferences without querying the Prover) can be made about the data stored in the data strucuture.

1.2. Requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.3. Terminology

The following terminology is used through this document:

SK: The private key for the VRF.

PK: The public key for the VRF.

alpha: The input to be hashed by the VRF.

beta: The VRF hash output.

pi: The VRF proof.

Prover: The Prover holds the private VRF key SK and public VRF key PK.

Verifier: The Verifier holds the public VRF key PK.

2. VRF Algorithms

A VRF comes with a key generation algorithm that generates a public VRF key PK and private VRF key SK.

A VRF hashes an input alpha using the private VRF key SK to obtain a VRF hash output beta

```
beta = VRF_hash(SK, alpha)
```

The VRF_hash algorithm is deterministic, in the sense that it always produces the same output beta given a pair of inputs (SK, alpha). The private key SK is also used to construct a proof pi that beta is the correct hash output

```
pi = VRF_prove(SK, alpha)
```

The VRFs defined in this document allow anyone to deterministically obtain the VRF hash output beta directly from the proof value pi as

```
beta = VRF_proof2hash(pi)
```

Notice that this means that

```
VRF_hash(SK, alpha) = VRF_proof2hash(VRF_prove(SK, alpha))
```

The proof pi allows a Verifier holding the public key PK to verify that beta is the correct VRF hash of input alpha under key PK. Thus, the VRF also comes with an algorithm

```
VRF_verify(PK, alpha, pi)
```

that outputs VALID if beta=VRF_proof2hash(pi) is correct VRF hash of alpha under key PK, and outputs INVALID otherwise.

3. VRF Security Properties

VRFs are designed to ensure the following security properties.

3.1. Full Uniqueness or Trusted Uniqueness

Uniqueness means that, for any fixed public VRF key and for any input alpha, there is a unique VRF output beta that can be proved to be valid. Uniqueness must hold even for an adversarial Prover that knows the VRF secret key SK.

"Full uniqueness" states that a computationally-bounded adversary cannot choose a VRF public key PK, a VRF input alpha, two different

VRF hash outputs beta1 and beta2, and two proofs pi1 and pi2 such that VRF_verify(PK, alpha, pi1) and VRF_verify(PK, alpha, pi2) both output VALID.

A slightly weaker security property called "trusted uniquness" sufficies for many applications. Trusted uniqueness is the same as full uniqueness, but it must hold only if the VRF keys PK and SK were generated in a trustworthy manner. In otherwords, uniqueness might not hold if keys were generated in an invalid manner or with bad randomness.

3.2. Full Collison Resistance or Trusted Collision Resistance

Like any cryprographic hash function, VRFs need to be collision resistant. Collison resistance must hold even for an adversarial Prover that knows the VRF secret key SK.

More percisely, "full collision resistance" states that it should be computationally infeasible for an adversary to find two distinct VRF inputs alpha1 and alpha2 that have the same VRF hash beta, even if that adversary knows the secret VRF key SK.

For most applications, a slightly weaker security property called "trusted collision resistance" suffices. Trusted collision resistance is the same as collision resistance, but it holds only if PK and SK were generated in a trustworthy manner.

3.3. Full Pseudorandomness or Selective Pseudorandomness

Pseudorandomness ensures that when an adversarial Verifier sees a VRF hash output beta without its corresponding VRF proof pi, then beta is indistinguishable from a random value.

More percisely, suppose the public and private VRF keys (PK, SK) were generated in a trustworthy manner. Pseudorandomness ensures that the VRF hash output beta (without its corresponding VRF proof pi) on any adversarially-chosen "target" VRF input alpha looks indistinguishable from random for any computationally bounded adversary who does not know the private VRF key SK. This holds even if the adversary also gets to choose other VRF inputs alpha’ and observe their corresponding VRF hash outputs beta’ and proofs pi’.

With "full pseudorandomness", the adversary is allowed to choose the "target" VRF input alpha at any time, even after it observes VRF outputs beta’ and proofs pi’ on a variety of chosen inputs alpha’.

"Selective pseudorandomness" is a weaker security property which suffices in many applications. Here, the adversary must choose the

target VRF input alpha independently of the public VRF key PK, and before it observes VRF outputs beta' and proofs pi' on inputs alpha' of its choice.

It is important to remember that the VRF output beta does not look random to the Prover, or to any other party that knows the private VRF key SK! Such a party can easily distinguish beta from a random value by comparing beta to the result of VRF_hash(SK, alpha).

Also, the VRF output beta does not look random to any party that knows valid VRF proof pi corresponding to the VRF input alpha, even if this party does not know the private VRF key SK. Such a party can easily distinguish beta from a random value by checking whether VRF_verify(PK, alpha, pi) returns "VALID" and beta = VRF_proof2hash(pi).

Also, the VRF output beta may not look random if VRF key generation was not done in a trustworthy fashion. (For example, if VRF keys were generated with bad randomness.)

3.4. An additional pseudorandomness property

[TODO: The following property is not needed for applications that use VRFs to prevent enumeration of hash-based data structures. However, we noticed that some other applications of VRF rely on this property. As we have not yet found a formal definition of this property in the literature, we write it down here.]

Pseudorandomness, as defined in Section 3.3, does not hold if the VRF keys were generated adversarially.

There is, however, a different type of pseudorandomness that could hold even if the VRF keys are generated adversarially, as long as the VRF input alpha is unpredictable. Suppose the VRF keys are generated by an adversary. Then, a VRF hash output beta should look pseudorandom to the adversary as long as (1) its corresponding VRF hash alpha is chosen randomly and independently of the VRF key, (2) alpha is unknown to the adversary, (3) the corresponding proof pi is unknown to the adversary, and (4) the VRF public key chosen by the adversary is valid.

[TODO: It should be possible to get the EC-VRF to satisfy this property, as long as verifiers run an VRF_validate_key() key function upon receipt of VRF public keys. However, we need to work out exactly what properties are needed from the VRF public keys in order for this property to hold. Some additional checks might need to be added to the ECVRF_validate_key() function. Need to work out what are these checks.]

4. RSA Full Domain Hash VRF (RSA-FDH-VRF)

The RSA Full Domain Hash VRF (RSA-FDH-VRF) is a VRF that satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in Section 3. Its security follows from the standard RSA assumption in the random oracle model. Formal security proofs are in [nsec5ecc].

The VRF computes the proof pi as a deterministic RSA signature on input alpha using the RSA Full Domain Hash Algorithm [RFC8017] parametrized with the selected hash algorithm. RSA signature verification is used to verify the correctness of the proof. The VRF hash output beta is simply obtained by hashing the proof pi with the selected hash algorithm.

The key pair for RSA-FDH-VRF MUST be generated in a way that it satisfies the conditions specified in Section 3 of [RFC8017].

In this document, the notation from [RFC8017] is used.

Parameters used:

(n, e) - RSA public key

K - RSA private key

k - length in octets of the RSA modulus n

Fixed options:

Hash - cryptographic hash function

hLen - output length in octets of hash function Hash

Constraints on options:

Cryptographic security of Hash is at least as high as the cryptographic security level of the RSA key

Primitives used:

I2OSP - Coversion of a nonnegative integer to an octet string as defined in Section 4.1 of [RFC8017]

OS2IP - Coversion of an octet string to a nonnegative integer as defined in Section 4.2 of [RFC8017]

RSASP1 - RSA signature primitive as defined in Section 5.2.1 of [RFC8017]

RSAVP1 - RSA verification primitive as defined in Section 5.2.2 of [RFC8017]

MGF1 - Mask Generation Function based on a hash function as defined in Section B.2.1 of [RFC8017]

4.1. RSA-FDH-VRF Proving

RSAFDHVRF_prove(K, alpha)

Input:

K - RSA private key

alpha - VRF hash input, an octet string

Output:

pi - proof, an octet string of length k

Steps:

1. EM = MGF1(alpha, k - 1)

2. m = OS2IP(EM)

3. s = RSASP1(K, m)

4. pi = I2OSP(s, k)

5. Output pi

4.2. RSA-FDH-VRF Proof To Hash

RSAFDHVRF_proof2hash(pi)

Input:

pi - proof, an octet string of length k

Output:

beta - VRF hash output, an octet string of length hLen

Steps:

1. beta = Hash(pi)

2. Output beta

4.3. RSA-FDH-VRF Verifying

RSAFDHVRF_verify((n, e), alpha, pi)

Input:

(n, e) - RSA public key

alpha - VRF hash input, an octet string

pi - proof to be verified, an octet string of length n

Output:

"VALID" or "INVALID"

Steps:

1. s = OS2IP(pi)

2. m = RSAVP1((n, e), s)

3. EM = I2OSP(m, k - 1)

4. EM’ = MGF1(alpha, k - 1)

5. If EM and EM’ are equal, output "VALID"; else output "INVALID".

5. Elliptic Curve VRF (EC-VRF)

The Elliptic Curve Verifiable Random Function (EC-VRF) is a VRF that satisfies the trusted uniqueness, trusted collision resistance, and full pseudorandomness properties defined in Section 3. The security of this VRF follows from the decisional Diffie-Hellman (DDH) assumption in the random oracle model. Formal security proofs are in [nsec5ecc].

Fixed options:

F - finite field

2n - length, in octets, of a field element in F

E - elliptic curve (EC) defined over F

m - length, in octets, of an EC point encoded as an octet string

G - subgroup of E of large prime order

q - prime order of group G

cofactor - number of points on E divided by q

g - generator of group G

Hash - cryptographic hash function

hLen - output length in octets of Hash

Constraints on options:

Field elements in F have bit lengths divisible by 16

hLen is equal to 2n

Parameters used:

y = g^x - VRF public key, an EC point

x - VRF private key, an integer where 0 < x < q [[CREF1: check this with leo --Sharon]]

Notation and primitives used:

p^k - when p is an EC point: point multiplication, i.e. k repetitions of group operation on EC point p. when p is an integer: exponentiation

|| - octet string concatenation

I2OSP - nonnegative integer conversion to octet string as defined in Section 4.1 of [RFC8017]

OS2IP - Coversion of an octet string to a nonnegative integer as defined in Section 4.2 of [RFC8017]

EC2OSP - conversion of EC point to an m-octet string as specified in Section 5.5

OS2ECP - conversion of an m-octet string to EC point as specified in Section 5.5. OS2ECP returns INVALID if the octet string does not convert to a valid EC point.

RS2ECP - conversion of a random 2n-octet string to an EC point as specified in Section 5.5

5.1. EC-VRF Proving

Note: this function is made more efficient by taking in the public VRF key y, as well as the private VRF key x.

ECVRF_prove(y, x, alpha)

Input:

y - public key, an EC point

x - private key, an integer

alpha - VRF input, an octet string

Output:

pi - VRF proof, octet string of length m+3n

Steps:

1. h = ECVRF_hash_to_curve(y, alpha)
2. gamma = h^x
3. choose a random integer nonce k from [0, q-1]
4. c = ECVRF_hash_points(g, h, y, gamma, g^k, h^k)
5. s = k - c*x mod q (where * denotes integer multiplication)
6. pi = EC2OSP(gamma) || I2OSP(c, n) || I2OSP(s, 2n)
7. Output pi

5.2. EC-VRF Proof To Hash

ECVRF_proof2hash(pi)

Input:

pi - VRF proof, octet string of length m+3n

Output:

"INVALID", or

beta - VRF hash output, octet string of length 2n

Steps:

1. D = ECVRF_decode_proof(pi)

2. If D is "INVALID", output "INVALID" and stop

3. (gamma, c, s) = D

4. beta = Hash(EC2OSP(gamma^cofactor))

5. Output beta

5.3. EC-VRF Verifying

ECVRF_verify(y, pi, alpha)

Input:

y - public key, an EC point

pi - VRF proof, octet string of length 5n+1

alpha - VRF input, octet string

Output:

"VALID" or "INVALID"

Steps:

1. D = ECVRF_decode_proof(pi)

2. If D is "INVALID", output "INVALID" and stop

3. (gamma, c, s) = D

4. u = y^c * g^s (where * denotes EC point addition, i.e. a group operation on two EC points)

5. h = ECVRF_hash_to_curve(y, alpha)

6. v = gamma^c * h^s (where * denotes EC point addition)

7. c’ = ECVRF_hash_points(g, h, y, gamma, u, v)

8. If c and c’ are equal, output "VALID"; else output "INVALID"

5.4. EC-VRF Auxiliary Functions

5.4.1. EC-VRF Hash To Curve

The ECVRF_hash_to_curve algorithm takes in an octet string alpha and converts it to h, an EC point in G.

5.4.1.1. ECVRF_hash_to_curve1

The following ECVRF_hash_to_curve1(y, alpha) algorithm implements ECVRF_hash_to_curve in a simple and generic way that works for any elliptic curve.

The running time of this algorithm depends on alpha. For the ciphersuites specified in Section 5.5, this algorithm is expected to find a valid curve point after approximately two attempts (i.e., when ctr=1) on average. See also [Icart09].

However, because the running time of algorithm depends on alpha, this algorithm SHOULD be avoided in applications where it is important that the VRF input alpha remain secret.

ECVRF_hash_to_curve1(y, alpha)

Input:

alpha - value to be hashed, an octet string

y - public key, an EC point

Output:

h - hashed value, a finite EC point in G

Steps:

1. ctr = 0

2. pk = EC2OSP(y)

3. h = "INVALID"

4. While h is "INVALID" or h is EC point at infinity:

   A. CTR = I2OSP(ctr, 4)

B. ctr = ctr + 1

C. attempted_hash = Hash(pk || alpha || CTR)

D. h = RS2ECP(attempted_hash)

E. If h is not "INVALID" and cofactor > 1, set h = h^cofactor

5. Output h

5.4.1.2. ECVRF_hash_to_curve2

For applications where VRF input alpha must be kept secret, the following ECVRF_hash_to_curve algorithm MAY be used to used as generic way to hash an octet string onto any elliptic curve.

[TODO: If there interest, we could look into specifying the generic deterministic time hash_to_curve algorithm from [Icart09]. Note also for the Ed25519 curve (but not the P256 curve), the Elligator algorithm could be used here.]

5.4.2. EC-VRF Hash Points

ECVRF_hash_points(p_1, p_2, ..., p_j)

Input:

p_i - EC point in G

Output:

h - hash value, integer between 0 and 2^(8n)-1

Steps:

1. P = empty octet string

2. for p_i in [p_1, p_2, ... p_j]:
   P = P || EC2OSP(p_i)

3. h1 = Hash(P)

4. h2 = first n octets of h1

5. h = OS2IP(h2)

6. Output h

5.4.3. EC-VRF Decode Proof

ECVRF_decode_proof(pi)

Input:

pi - VRF proof, octet string (m+3n octets)

Output:

"INVALID", or

gamma - EC point

c - integer between 0 and 2^(8n)-1

s - integer between 0 and 2^(16n)-1

Steps:

1. let gamma’, c’, s’ be pi split after m-th and m+n-th octet
2. gamma = OS2ECP(gamma’)
3. if gamma = "INVALID" output "INVALID" and stop.
4. c = OS2IP(c’)
5. s = OS2IP(s’)
6. Output gamma, c, and s

5.5. EC-VRF Ciphersuites

This document defines EC-VRF-P256-SHA256 as follows:

- o The EC group G is the NIST-P256 elliptic curve, with curve parameters as specified in [FIPS-186-3] (Section D.1.2.3) and [RFC5114] (Section 2.6). For this group, 2n = 32 and cofactor = 1.
- o The key pair generation primitive is specified in Section 3.2.1 of [SECG1].
- o EC2OSP is specified in Section 2.3.3 of [SECG1] with point compression on. This implies m = 2n + 1 = 33.
- o OS2ECP is specified in Section 2.3.4 of [SECG1].

- o RS2ECP(h) = OS2ECP(0x02 || h). The input h is a 32-octet string and the output is either an EC point or "INVALID".

- o The hash function Hash is SHA-256 as specified in [RFC6234].

- o The ECVRF_hash_to_curve function is as specified in Section 5.4.1.1.

This document defines EC-VRF-ED25519-SHA256 as follows:

- o The EC group G is the Ed25519 elliptic curve with parameters defined in Table 1 of [RFC8032]. For this group, 2n = 32 and cofactor = 8.

- o The key pair generation primitive is specified in Section 5.1.5 of [RFC8032]

- o EC2OSP is specified in Section 5.1.2 of [RFC8032]. This implies m = 2n = 32.

- o OS2ECP is specified in Section 5.1.3 of [RFC8032].

- o RS2ECP is equivalent to OS2ECP.

- o The hash function Hash is SHA-256 as specified in [RFC6234].

- o The ECVRF_hash_to_curve function is as specified in Section 5.4.1.1.

[TODO: Should we add an EC-VRF-ED25519-SHA256-Elligator ciphersuite where the Elligator hash function is used for ECVRF_hash-to-curve?]

[TODO: Add an Ed448 ciphersuite?]

[NOTE: In the unlikely case that future versions of this spec use a elliptic curve group G that does not also come with a specification of the group generator g, then we can still have full uniqueness and full collision resistance by adding an check to ECVRF_validate_key(PK) that ensures that g is a point on the elliptic curve and g^cofactor is not the EC point at infinity.]

5.6. When the EC-VRF Keys are Untrusted

The EC-VRF as specified above is a VRF that satisfies the "trusted uniqueness", "trusted collision resistance", and "full pseudorandomness" properties defined in Section 3. If the elliptic curve parameters (including the generator g) are trusted, but the VRF public key PK is not trusted, this VRF can be modified to

additionally satisfy "full uniqueness", and "full collision resistance". This is done by additionally requiring the Verifier to perform the following validation procedure upon receipt of the public VRF key.

The Verifier MUST perform this validation procedure when the entity that generated the public VRF key is untrusted. The public key MUST NOT be used if this procedure returns "INVALID". Note well that this procedure is not sufficient if the elliptic curve E or if g, the generator of group G, is untrusted.

This procedure supposes that the public key provided to the Verifier is an octet string. The procedure returns "INVALID" if the public key in invalid. Otherwise, it returns y, the public key as an EC point.

5.6.1. EC-VRF Validate Key

ECVRF_validate_key(PK)

Input:

PK - public key, an octet string

Output:

"INVALID", or

y - public key, an EC point

Steps:

1. y = OS2ECP(PK)

2. If y is "INVALID", output "INVALID" and stop

3. If y^cofactor is the EC point at infinty, output "INVALID" and stop

4. Output y

6. Implementation Status

An implementation of the RSA-FDH-VRF (SHA-256) and EC-VRF-P256-SHA256 was first developed as a part of the NSEC5 project [I-D.vcelak-nsec5] and is available at <http://github.com/fcelda/nsec5-crypto>. The EC-VRF implementation may be out of date as this spec has evolved.

The Key Transparency project at Google uses a VRF implemention that is similar to the EC-VRF-P256-SHA256, with a few minor changes including the use of SHA-512 instead of SHA-256. Its implementation is available <https://github.com/google/keytransparency/blob/master/core/vrf/vrf.go>

An implementation by Yahoo! similar to the EC-VRF is available at <https://github.com/r2ishiguro/vrf>.

An implementation similar to EC-VRF is available as part of the CONIKS implementation in Golang at <https://github.com/coniks-sys/coniks-go/tree/master/crypto/vrf>.

Open Whisper Systems also uses a VRF very similar to EC-VRF-ED25519-SHA512-Elligator, called VXEdDSA, and specified here: <https://whispersystems.org/docs/specifications/xeddsa/>

7. Security Considerations

7.1. Key Generation

Applications that use the VRFs defined in this document MUST ensure that that the VRF key is generated correctly, using good randomness.

7.1.1. Uniqueness and collision resistance with untrusted keys

The EC-VRF as specified in Section 5.1-Section 5.5 statisfies the "trusted uniqueness" and "trusted collision resistance" properties as long as the VRF keys are generated correctly, with good randomness. If the Verifier trusts the VRF keys are generated correctly, it MAY use the public key y as is.

However, if the EC-VRF uses keys that could be generated adversarially, then the the Verfier MUST first perform the validation procedure ECVRF_validate_key(PK) (specified in Section 5.6) upon receipt of the public key PK as an octet string. If the validation procedure outputs "INVALID", then the public key MUST not be used. Otherwise, the procedure will output a valid public key y, and the EC-VRF with public key y satisfies the "full uniqueness" and "full collision resistance" properties.

The RSA-FDH-VRF statisfies the "trusted uniqueness" and "trusted collision resistance" properties as long as the VRF keys are generated correctly, with good randomness. These properties may not hold if the keys are generated adversarially (e.g., if RSA is not permutation). Meanwhile, the "full uniqueness" and "full collision resistance" are properties that hold even if VRF keys are generated

by an adversary. The RSA-FDH-VRF defined in this document does not have these properties. However, if adversarial key generation is a concern, the RSA-FDH-VRF may be modifed to have these properties by adding additional cryptographic checks that its public key has the right form. These modifications are left for future specification.

7.1.2. Pseudorandomness with untrusted keys

Without good randomness, the "pseudorandomness" properties of the VRF may not hold. Note that it is not possible to guarantee pseudorandomness in the face of adversarially generated VRF keys. This is because an adversary can always use bad randomness to generate the VRF keys, and thus, the VRF output may not be pseudorandom.

7.2. Selective vs Full Pseudorandomness

[nsec5ecc] presents cryptographic reductions to an underlying hard problem (e.g. Decisional Diffie Hellman for the EC-VRF, or the standard RSA assumption for RSA-FDH-VRF) that prove the VRFs specificied in this document possess full pseudorandomness as well as selective pseudorandomness. However, the cryptographic reductions are tighter for selective pseudorandomness than for full pseudorandomness. This means the the VRFs have quantitavely stronger security guarentees for selective pseudorandomness.

Applications that are concerned about tightness of cryptographic reductions therefore have two options.

- o They may choose to ensure that selective pseudorandomness is sufficient for the application. That is, that pseudorandomness of outputs matters only for inputs that are chosen independently of the VRF key.

- o If full pseudorandomness is required for the application, the application may increase security parameters to make up for the loose security reduction. For RSA-FDH-VRF, this means increasing the RSA key length. For EC-VRF, this means increasing the cryptographic strength of the EC group G. For both RSA-FDH-VRF and EC-VRF the cryptographic strength of the hash function Hash may also potentially need to be increased.

7.3. Proper randomness for EC-VRF

Applications that use the EC-VRF defined in this document MUST ensure that the random nonce k used in the ECVRF_prove algorithm is chosen with proper randomness. Otherwise, an adversary may be able to

recover the private VRF key x (and thus break pseudorandomness of the VRF) after observing several valid VRF proofs pi.

7.4. Timing attacks

The EC-VRF_hash_to_curve algorithm defined in Section 5.4.1.1 SHOULD NOT be used in applications where the VRF input alpha is secret and is hashed by the VRF on-the-fly. This is because the EC-VRF_hash_to_curve algorithm's running time depends on the VRF input alpha, and thus creates a timing channel that can be used to learn information about alpha. That said, for most inputs the amount of information obtained from such a timing attack is likely to be small (1 bit, on average), since the algorithm is expected to find a valid curve point after only two attempts. However, there might be inputs which cause the algorithm to make many attempts before it finds a valid curve point; for such inputs, the information leaked in a timing attack will be more than 1 bit.

8. Change Log

Note to RFC Editor: if this document does not obsolete an existing RFC, please remove this appendix before publication as an RFC.

00 - Forked this document from draft-vcelak-nsec5-04. Cleaned up the definitions of VRF algorithms. Added security definitions for VRF and security considerations. Parameterized EC-VRF so it could support curves other than P-256 and Ed25519.

01 - Fixed ECVRF to work when cofactor > 1. Changed ECVRF_proof2hash(pi) so that it outputs a value raised to the cofactor and then processed by the cryptographic hash function Hash. Included the VRF public key y as input to the hash function ECVRF_hash_to_curve1. Cleaned up ciphersuites and ECVRF description so that it works with EC point encodings for both P256 and Ed25519 curves. Added ECVRF_validate_key so that EC-VRF can satisfy "full uniqueness" and "full collision" resistance. Updated implementation status. Added "an additional pseudorandomness property" to security definitions.

9. Contributors

Leonid Reyzin (Boston University) is a major contributor to this document.

This document also would not be possible without the work of Moni Naor (Weizmann Institute), Sachin Vasant (Cisco Systems), and Asaf Ziv (Facebook). Shumon Huque (Salesforce) and David C. Lawerence (Akamai) provided valuable input to this draft.

10. References

10.1. Normative References

[FIPS-186-3]
National Institute for Standards and Technology, "Digital Signature Standard (DSS)", FIPS PUB 186-3, June 2009.

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

[RFC5114] Lepinski, M. and S. Kent, "Additional Diffie-Hellman Groups for Use with IETF Standards", RFC 5114, DOI 10.17487/RFC5114, January 2008, <http://www.rfc-editor.org/info/rfc5114>.

[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, <http://www.rfc-editor.org/info/rfc6234>.

[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, <http://www.rfc-editor.org/info/rfc8017>.

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

[SECG1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", Version 2.0, May 2009, <http://www.secg.org/sec1-v2.pdf>.

10.2. Informative References

[I-D.vcelak-nsec5]
Vcelak, J., Goldberg, S., Papadopoulos, D., Huque, S., and D. Lawrence, "NSEC5, DNSSEC Authenticated Denial of Existence", draft-vcelak-nsec5-04 (work in progress), March 2017.

[Icart09] Icart, T., "How to Hash into Elliptic Curves", in CRYPTO, 2009.

[MRV99] Michali, S., Rabin, M., and S. Vadhan, "Verifiable Random Functions", in FOCS, 1999.

[nsec5ecc] Papadopoulos, D., Wessels, D., Huque, S., Vcelak, J., Naor, M., Reyzin, L., and S. Goldberg, "Making NSEC5 Practical for DNSSEC Deployments", in ePrint Cryptology Archive 2017/099, February 2017, <https://eprint.iacr.org/2017/099.pdf>.

Appendix A. Open Issues

Note to RFC Editor: please remove this appendix before publication as an RFC.

1. Open issue.

Authors' Addresses

Sharon Goldberg
Boston University
111 Cummington St, MCS135
Boston, MA 02215
USA

EMail: goldbe@cs.bu.edu

Dimitrios Papadopoulos
University of Maryland
8223 Paint Branch Dr
College Park, MD 20740
USA

EMail: dipapado@bu.edu

Jan Vcelak
NS1
16 Beaver St
New York, NY 10004
USA

EMail: jvcelak@ns1.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 3, 2018

P. Hoffman
ICANN

July 2, 2017

The Transition from Classical to Post-Quantum Cryptography
draft-hoffman-c2pq-01

Abstract

Quantum computing is the study of computers that use quantum features in calculations. For over 20 years, it has been known that if large-scale quantum computers could be built, they could have a devastating effect on classical cryptographic algorithms such as RSA and elliptic curve signatures and key exchange, as well as on encryption algorithms. There has already been a great deal of study on how to create algorithms that will resist large-scale quantum computers, but so far, the properties of those algorithms make them onerous to adopt before they are needed.

Small-scale quantum computers are being built today, but it is still far from clear when large-scale quantum computers that can be used to break classical algorithms with key sizes commonly used today will be available. It is important to be able to predict when large-scale quantum computers usable for cryptanalysis will be possible so that organization can change to post-quantum cryptographic algorithms well before they are needed.

This document describes quantum computing, how it can be used to attack classical cryptographic algorithms, and possibly how to predict when large-scale quantum computers will become feasible.

Status of This Memo

This Internet-Draft will expire on January 3, 2018.

Copyright Notice

Table of Contents

1. Introduction

Early drafts of this document use "@@@@@" to indicate where the editor particularly want input from reviewers. The editor welcomes all types of review, but the areas marked with "@@@@@" are in the most noticeable need of new material. (The editor particularly appreciates new material that comes with references that can be included in this document as well.)

1.1. Disclaimer

**** This is an early version of this draft. **** As such, it has had little in-depth review in the cryptography community. Statements in this document might be wrong; given that the entire document is about cryptography, those wrong statements might have significant security problems associated with them.

Readers of this document should not rely on any statements in this version of this draft. As the draft gets more input from the cryptography community over time, this disclaimer will be softened and eventually eliminated.

1.2. Executive Summary

The development of quantum computers that can break classical cryptographic keys is at a very early stage. None of the published examples of such quantum computers is useful in breaking keys that are in use today. There is a great amount of interest in this development, and researchers expect large strides in this development in the coming decade.

Because the world does not know when large-scale quantum computers that can break cryptographic keys will be available, organizations should be watching this area so that they have plenty of time to either change to larger key sizes for classical cryptography or to change to post-quantum algorithms. See Section 5 for a fuller discussion of determining how to predict when large-scale quantum computers might become feasible.

1.3. Terminology

The term "classical cryptography" is used to indicate the cryptographic algorithms that are in common use today. In particular, signature and key exchange algorithms that are based on the difficulty of factoring numbers into two large prime numbers, or

are based on the difficulty of determining the discrete log of a large composite number, are considered classical cryptography.

The term "post-quantum cryptography" is the invention and study of encryption, signature and key exchange algorithms that are not based on the difficulty of factoring numbers into two large prime numbers, nor on the difficulty of determining the discrete log of a large composite number.

Note that these definitions apply to only one aspect of quantum computing as it relates to cryptography. It is expected that quantum computing will also be able to be used against symmetric key cryptography to make it possible to search for a secret symmetric key using far fewer operations than are needed using classical computers (see Section 4 for more detail). However, using longer keys to thwart that possibility is not normally called "post-quantum cryptography".

There are many terms that are only used in the field of quantum computing, such as "qubit", "quantum algorithm", and so on. Chapter 1 of [NielsenChuang] has good definitions of such terms.

The "^" symbol is used to indicate "the power of". The term "log" always means "logarithm base 2".

1.4. Not Covered: Post-Quantum Cryptographic Algorithms

This document discusses when an organization would want to consider using post-quantum cryptographic algorithms, but definitely does not delve into which of those algorithms would be best to use. Post-quantum cryptography is an active field of research; in fact, it is much more active than the study of when we might want to transition from classical to post-quantum cryptography.

Readers interested in post-quantum cryptographic algorithms will have no problem finding many articles proposing such algorithms, comparing the many current proposals, and so on. An excellent starting point is the web site <http://pqcrypto.org/>. Another is the article on post-quantum cryptography at Wikipedia: <https://en.wikipedia.org/wiki/Post-quantum_cryptography>.

In addition, various organizations are working on standardizing the algorithms for post-quantum cryptography. For example, the US National Institute of Standards and Technology (commonly just called "NIST") is holding a competition to evaluate post-quantum cryptographic algorithms. NIST’s description of that effort is currently at <http://csrc.nist.gov/groups/ST/post-quantum-crypto/>.

1.5. Not Covered: Quantum Cryptography

Outside of this section, this document does not cover "quantum cryptography". The field of quantum cryptography is related to quantum computers, but not to cryptanalysis. Quantum cryptography is used to share random values that cannot be observed by outside parties without discovery.

1.6. Where to Read More

There are many reasonably accessible articles on Wikipedia, notably <https://en.wikipedia.org/wiki/Quantum_computing>.

@@@@@ Note to the CFRG: please review the various pages at Wikipedia and update them if they are wrong or out of date. Doing so is incredibly helpful to the world.

[NielsenChuang] is a well-regarded college textbook on quantum computers. Prerequisites for understanding the book include linear algebra and some quantum physics; however, even without those, a reader can probably get value from the introductory material in the book.

@@@@@ Maybe add more references that might be useful to non-experts.

2. Brief Introduction to Quantum Computers

A quantum computer is a computer that uses quantum bits (qubits) in quantum circuits to perform calculations. Quantum computers also use classical bits and regular circuits: most calculations in a quantum computer are a mix of classical and quantum bits and circuits.

@@@@@ This can be expanded and made less hand-wavy.

Qubits are valuable in quantum computers when they are combined in calculations. Combining qubits in a calculation requires that the qubits are correlated. Correlating qubits requires much more effort than correlating classical bits (such as in registers or volatile memory), which is one of the main reasons that developing quantum computers has proven more difficult than early development of classical computers.

@@@@@ Discuss measurements and how they have to be done with correlated qubits.

2.1. Quantum Computers that Discover Cryptographic Keys

Quantum computers are expected to be useful in the future for some problems that take up too many resources on a large classical computer. However, this document only discusses how they might be used to discover cryptographic keys faster than classical computers. In order to discover cryptographic keys, a quantum computer needs to have a quantum circuit specifically designed for the type of key it is attempting to break.

A quantum computer will need to have a circuit with thousands of qubits to be useful to discover the type and size keys that are in common use today. Smaller quantum computers (those with fewer qubits and simpler circuits) are not useful for using Shor’s algorithm (as discussed in Section 3.1) at all. That is, no one has devised a way to combine a bunch of smaller quantum computers to perform the same attacks on cryptographic keys via Shor’s algorithm as a properly-sized quantum computer.

This is why this document uses the term "large-scale quantum computer" when describing ones that can be used to break keys: there will certainly be small-scale quantum computers built first, but those computers cannot be used to discover the type and size keys that are in common use today.

A straight-forward application of Shor’s algorithm may not be the only way for large-scale quantum computers to attack RSA keys. [LowResource] describes how to combine quantum computers with classical methods for breaking RSA keys at speeds faster than just using the classical methods.

2.2. Physical Designs for Quantum Computers

Quantum computers can be built using many different physical technologies. Deciding which physical technologies are best to pursue is an extremely active research topic. A few physical technologies (particularly trapped ions, super-conduction using Josephson junctions, and nuclear magnetic resonance) are currently getting the most press, but other technologies are also showing promise.

@@@@@ It would be useful to have maybe two paragraphs about each physical design that is being actively pursued.

2.3. Challenges for Physical Designs

Different designs have different challenges to overcome before the physical technology can be scaled enough to build a useful large-scale quantum computer. Some of those challenges include the following. (Note that some items on this list apply only to some of the physical technologies

Temperature: Getting stable operation without extreme cooling is difficult for many of the proposed technologies. The definition of "extreme" is different for different low-temperature technologies.

Stabilization: The length of time every qubit in a circuit holds is value

Quantum control: Coherence and reproducibility of qubits

Error detection and correction: Getting accurate results through simultaneous detection of bit-flip and phase-flip. See Section 2.4 for a longer description of this.

Substrate: The material on which the qubit circuits are built. This has a large effect on the stability of the qubits.

Particles: The atoms or sub-atomic particles used to make the qubits

Scalability: The ability to handle the number of physical qubits needed for the desired the circuit

Architecture: Ability to change quantum gates in a circuit

2.4. Qubits, Error Detection, and Error Correction

Researchers building small-scale quantum computers have discovered that correlating qubits often has a large rate of error, and that error increases rapidly over time. Performing quantum calculations such as those needed to break cryptographic keys is not feasible with the current state of physical qubits.

Researchers have also discovered that they do not need to rely only on the properties of physical qubits. Instead, they can build "logical qubits" from multiple physical qubits, and these logical qubits have much lower error rates over much longer lifetimes. Currently, it is estimated that it takes hundreds or thousands of physical qubits to make a logical qubit.

@@@@@ Lots more material should goe here. We will need recent references for how many physical qubits are needed for each corrected qubit. It’s OK if this section has lots of references, but hopefully they don’t contradict each other.

3. Quantum Computers and Public Key Cryptography

The area of quantum computing that has generated the most interest in the cryptographic community is the ability of quantum computers to find the secret keys in the RSA and Diffie-Hellman algorithms using many fewer operations than classical computers would need to use. It is widely believed that factoring large numbers and finding discrete logs using classical computers increases with the exponential size of the key. [RFC3766] describes in detail how classical computers can be used to determine keys; even though that RFC is over a decade old, no significant changes have been made to the process of classical attacks on RSA and Diffie-Hellman. @@@@@ CFRG: is that true? Does RFC 3766 need to be updated?

Shor’s algorithm shows that these problems can be solved on quantum computers in polynomial time, meaning that the speed of finding the keys is a polynomial function based on the size of the keys, which would require significantly fewer steps than a classical computer. The definitive paper on Shor’s algorithm is [Shor97].

3.1. Explanation of Shor’s Algorithm

@@@@@ Pointers to understandable articles would be good here.

@@@@@ Describe period-finding and why it applies to finding prime factors and discrete logs.

@@@@@ Give the steps for applying Shor’s algorithm to 2048-bit RSA. Describe how many rounds of the quantum subroutine would likely be needed. Describe how many rounds of the classical loop would likely be needed.

@@@@@ Give the steps for applying Shor’s algorithm to 256-bit elliptic curves. Describe how many rounds of the quantum subroutine would likely be needed. Describe how many rounds of the classical loop would likely be needed.

3.2. Properties of Large-Scale Quantum Computers Needed for Discovering Public Keys

Researchers have built small-scale quantum computers that implement Shor’s algorithm, factoring numbers with four or five bits. These

are used to show that Shor’s algorithm is possible to realize in actual hardware.

@@@@@ References are needed here. Did they implement all of Shor’s algorithm, including the looping logic in the classical part and the looping logic in the quantum part?

@@@@@ Numbers and explanation is needed below:

A quantum computer that can determine the secret keys for 2048-bit RSA would require SOME NUMBER GOES HERE correlated qubits and SOME NUMBER GOES HERE circuit elements. A quantum computer that can determine the secret keys for 256-bt elliptic curves would require SOME NUMBER GOES HERE correlated qubits and SOME NUMBER GOES HERE circuit elements.

4. Quantum Computers and Symmetric Key Cryptography

Section 3 is about Shor’s algorithm and compromises to public key cryptography. There is a second quantum computing algorithm, Grover’s algorithm, that is often mentioned at the same time as Shor’s algorithm but, with respect to cryptanalysis, only applies to symmetric ciphers such as AES. The definitive paper on Grover’s algorithm is by Grover: [Grover96]. Grover later wrote a more accessible paper about the algorithm in [QuantumSearch].

Grover’s algorithm gives a way to search for keys to symmetric algorithms in the square root of the time that a normal exhaustive search would take. Thus, a large-scale quantum computer that implemented Grover’s algorithm could find a secret AES-128 key in about 2^64 steps instead of the 2^128 steps that would be required for a classical computer.

When it appears that it is feasible to build a large-scale quantum computer that can defeat a particular symmetric algorithm at a particular key size, the proper response would be to use keys with twice as many bits. That is, if one is using the AES-128 algorithm and there is a concern that an adversary might be able to build a large-scale quantum computer that is designed to attack AES-128 keys, move to an algorithm that has keys twice as long as AES-128, namely AES-256.

It is currently expected that large-scale quantum computers that implement Grover’s algorithm are expected to be built long before ones that implement Shor’s algorithm are. There are two primary reasons for this:

- o Grover’s algorithm is likely to be useful in areas other than cryptography. For example, a large-scale quantum computer that implements Grover’s algorithm might be used to help create medicines by speeding up complex problems that involve how proteins fold. @@@@@ Add more likely examples and references here.

- o A large-scale quantum computer that can be used to break AES-128 will likely much smaller (and thus easier to build) than one that implements Shor’s algorithm for 256-bit elliptic curves or 2048-bit RSA/DSA keys.

4.1. Explanation of Grover’s Algorithm

@@@@@ Give the steps for applying Grover’s algorithm to AES-128.

4.2. Properties of Large-Scale Quantum Computers Needed for Discovering Symmetric Keys

@@@@@ Numbers and explanation is needed below:

A quantum computer that can determine the secret keys for AES-128 would require SOME NUMBER GOES HERE correlated qubits and SOME NUMBER GOES HERE circuit elements.

@@@@@ <https://arxiv.org/abs/1512.04965> indicates that the quantum part of the computer would have more than 2^80 quantum gates, which might be prohibitive for physical hardware.

5. Predicting When Useful Cryptographic Attacks Will Be Feasible

If quantum computers that perform useful cryptographic attacks can be built in the future, many organizations will want to start using post-quantum algorithms well before those computers can be built. However, given how few implementations of such quantum computers exist (even for tiny keys), it is impossible to predict with any accuracy when quantum computers that perform useful cryptographic attacks will be feasible.

The term "useful" above is relative to the value of the material being protected by the cryptographic algorithm to the attacker. For example, if the quantum computer attacking a particular key costs US$100 billion to build, costs US$1 billion a year to run, and can extract only one key a year, it is possibly useful to some governments, but probably not useful for attacking the TLS key used to protect a small mail server. On the other hand, if later a similar computer costs US$1 billion to build, costs US$10 million a year to run, and can extract ten keys a year, many more keys become vulnerable.

[BeReady] gives a simple way to approach the calculation of when one needs to deploy post-quantum algorithms. In short, if the sum of how long you need your keys to be secure plus how long it takes to deploy new algorithms is longer than the length of time it will take for an attacker to create a large-scale quantum computer and use it against your keys, then you waited too long.

@@@@@ If the following is wrong, it would be great to have references to replace this with

To date, few people have done systematic research that would give estimates for when useful quantum-based cryptographic attacks might be feasible, and at what cost. Without such research, it is easy to make wild guesses but those are not of much value to people having to decide when to start using post-quantum cryptography.

For example, in [NIST8105], NIST says "researchers working on building a quantum computer have estimated that it is likely that a quantum computer capable of breaking 2000-bit RSA in a matter of hours could be built by 2030 for a budget of about a billion dollars". However, the referenced link is to a YouTube video [MariantoniYoutube] where the researcher, Matteo Mariantoni, says "maybe you should not quote me on that". [NIST8105] gives no other references for predictions on cost and availability of useful cryptographic attacks with quantum computers.

5.1. Proposal: Public Measurements of Various Quantum Technologies

In order to get a rough idea of when useful cryptographic attacks with quantum computers may be feasible, researchers creating such computers can demonstrate them when they can break keys an eighth the size of those in common use. That is, given that 2048-bit RSA, 256-bit elliptic curve, and AES-128 are common today, when a research team has a computer than can break 256-bit RSA, 32-bit elliptic curve, or AES-128 where only 16 bits are unknown, they should demonstrate it.

Such a demonstration could easily be made fair with trusted representatives from the cryptographic community using verifiable means to pick the keys to break and verifying the time that it takes to break each key. It might be interesting to run the same tests in classical computers at the same time to give perspective.

Note that this proposal would only give an idea of how public progress is being made on quantum computers. Well-funded military agencies (and possibly even criminal enterprises) could be way ahead of the publicly-visible computers. No one should rely on just the

public measurements when deciding how safe their keys are against quantum computers.

6. IANA Considerations

None, and thus this section can be removed at final publication.

7. Security Considerations

This entire document is about cryptography, and thus about security.

See Section 1.1 for an important disclaimer about this document and security.

This document is meant to help the reader predict when to transition from using classical cryptographic algorithms to post-quantum algorithms. That decision is ultimately up to the reader, and must be made not only based on predictions of how quantum computing is progressing but also the value of every key that the user handles. For example, a financial institution using TLS to protect its customers' transactions will probably consider its keys more valuable than a small online store, and will thus be likely to begin the transition earlier.

8. Acknowledgements

The list here is meant to acknowledge input to this document. The people listed here do not necessarily agree with ideas presented.

Some of the ideas in this document come from Denis Butin and Tomofumi Okubo.

9. References

9.1. Normative References

[Grover96] Grover, L., "A fast quantum mechanical algorithm for database search", 1996, <https://arxiv.org/abs/quant-ph/9605043>.

[Shor97] Shor, P., "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", 1997, <http://epubs.siam.org/doi/pdf/10.1137/S0097539795293172>.

9.2. Informative References

[BeReady] Mosca, M., "Cybersecurity in an era with quantum computers: will we be ready?", 2015, <http://eprint.iacr.org/2015/1075.pdf>.

[LowResource] Bernstein, D., Fiassse, J., and M. Mosca, "A low-resource quantum factoring algorithm", 2017, <https://eprint.iacr.org/2017/352.pdf>.

[MariantoniYoutube] Mariantoni, M., "Building a Superconducting Quantum Computer", 2014, <https://www.youtube.com/watch?v=wWHAs--HA1c>.

[NielsenChuang] Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information, 10th Anniversary Edition", ISBN 97801-107-00217-3 , 2010.

[NIST8105] Chen, L. and et. al, "Report on Post-Quantum Cryptography", 2016, <http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf>.

[QuantumSearch] Grover, L., "From Schrodinger’s Equation to the Quantum Search Algorithm", 2001, <https://arxiv.org/abs/quant-ph/0109116>.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, DOI 10.17487/RFC3766, April 2004, <http://www.rfc-editor.org/info/rfc3766>.

Author’s Address

Paul Hoffman
ICANN

Email: paul.hoffman@icann.org

Network Working Group
Internet-Draft
Intended status: Informational
Expires: November 27, 2020

P. Hoffman
ICANN

May 26, 2020

The Transition from Classical to Post-Quantum Cryptography
draft-hoffman-c2pq-07

Abstract

Quantum computing is the study of computers that use quantum features in calculations. For over 20 years, it has been known that if very large, specialized quantum computers could be built, they could have a devastating effect on asymmetric classical cryptographic algorithms such as RSA and elliptic curve signatures and key exchange, as well as (but in smaller scale) on symmetric cryptographic algorithms such as block ciphers, MACs, and hash functions. There has already been a great deal of study on how to create algorithms that will resist large, specialized quantum computers, but so far, the properties of those algorithms make them onerous to adopt before they are needed.

Small quantum computers are being built today, but it is still far from clear when large, specialized quantum computers will be built that can recover private or secret keys in classical algorithms at the key sizes commonly used today. It is important to be able to predict when large, specialized quantum computers usable for cryptanalysis will be possible so that organization can change to post-quantum cryptographic algorithms well before they are needed.

This document describes quantum computing, how it might be used to attack classical cryptographic algorithms, and possibly how to predict when large, specialized quantum computers will become feasible.

Status of This Memo

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 November 27, 2020.

Copyright Notice

Table of Contents

1. Introduction

Early drafts of this document use "@@@@@" to indicate where the editor particularly want input from reviewers. The editor welcomes all types of review, but the areas marked with "@@@@@" are in the most noticeable need of new material. (The editor particularly appreciates new material that comes with references that can be included in this document as well.)

1.1. Disclaimer

**** This is still an early version of this draft. **** As such, it has had only some review in the cryptography community. Statements in this document might be wrong; given that the entire document is about cryptography, those wrong statements might have significant security problems associated with them.

Readers of this document should not rely on any statements in this version of this draft. As the draft gets more input from the cryptography community over time, this disclaimer will be softened and eventually eliminated.

1.2. Executive Summary

The development of quantum computers that can recover private or secret keys in classical algorithms at the key sizes commonly used today is at a very early stage. None of the published examples of such quantum computers is useful in recovering keys that are in use today. There is a great amount of interest in this development, and researchers expect large strides in this development in the coming decade.

There is active research in standardizing signing and key exchange algorithms that will withstand attacks from large, specialized quantum computers. However, all those algorithms to date have very large keys, very large signatures, or both. Thus, there is a large sustained cost in using those algorithms. Similarly, there is a large cost in being surprised about when quantum computers can cause damage to current cryptographic keys and signatures.

Because the world does not know when large, specialized quantum computers that can recover cryptographic keys will be available, organizations should be watching this area so that they have plenty of time to either change to larger key sizes for classical cryptography or to change to post-quantum algorithms. See Section 6 for a fuller discussion of determining how to predict when quantum computers that can harm current cryptography might become feasible.

1.3. Terminology

The term "classical cryptography" is used to indicate the cryptographic algorithms that are in common use today. In particular, signature and key exchange algorithms that are based on the difficulty of factoring numbers into two large prime numbers, or are based on the difficulty of determining the discrete log of a large composite number, are considered classical cryptography.

The term "post-quantum cryptography" refers to the invention and study of cryptographic mechanisms in which the security does not rely on computationally hard problems that can be efficiently solved on quantum computers. This excludes systems whose security relies on factoring numbers, or the difficulty of determining the discrete log of one group element with respect to another.

Note that these definitions apply to only one aspect of quantum computing as it relates to cryptography. It is expected that quantum computing will also be able to be used against symmetric key cryptography to make it possible to search for a secret symmetric key using far fewer operations than are needed using classical computers (see Section 5 for more detail). However, using longer keys to thwart that possibility is not normally called "post-quantum cryptography".

There are many terms that are only used in the field of quantum computing, such as "qubit", "quantum algorithm", and so on. Chapter 1 of [NielsenChuang] has good definitions of such terms.

Some papers discussing quantum computers and cryptanalysis say that large, specialized quantum computers "break" algorithms in classical cryptography. This paper does not use that terminology because the algorithms' strength will be reduced when large, specialized quantum computers exist, but not to the point where there is an immediate need to change algorithms.

The "^" symbol is used to indicate "the power of". The term "log" always means "logarithm base 2".

1.4. Where to Read More

There are many reasonably accessible articles on Wikipedia, notably the overview article at <https://en.wikipedia.org/wiki/Quantum_computing> and the timeline of quantum computing developments at <https://en.wikipedia.org/wiki/Timeline_of_quantum_computing>.

[NielsenChuang] is a well-regarded college textbook on quantum computers. Prerequisites for understanding the book include linear algebra and some quantum physics; however, even without those, a reader can probably get value from the introductory material in the book.

A good overview of the current status of quantum computing in general is [ProgressProspects].

[QCPolicy] describes how the development of quantum computing affects encryption policies.

[Turing50Youtube] is a good overview of the near-term and longer-term prospects for designing and building quantum computers; it is a video of a panel discussion by quantum hardware and software experts given at the ACM's Turing 50 lecture.

@@@@@ Maybe add more references that might be useful to non-experts.

1.5. Not Covered: Post-Quantum Cryptographic Algorithms

This document discusses when an organization would want to consider using post-quantum cryptographic algorithms, but definitely does not delve into which of those algorithms would be best to use. Post-quantum cryptography is an active field of research; in fact, it is much more active than the study of when we might want to transition from classical to post-quantum cryptography.

Readers interested in post-quantum cryptographic algorithms will have no problem finding many articles proposing such algorithms, comparing the many current proposals, and so on. An excellent starting point is the web site <http://pqcrypto.org/>. The Open Quantum Safe (OQS) project <https://openquantumsafe.org/> is developing and prototyping quantum-resistant cryptography. Another is the article on post-quantum cryptography at Wikipedia: <https://en.wikipedia.org/wiki/Post-quantum_cryptography>.

Various organizations are working on standardizing the algorithms for post-quantum cryptography. For example, the US National Institute of Standards and Technology (commonly just called "NIST") is holding a competition to evaluate post-quantum cryptographic algorithms.

NIST's description of that effort is currently at <http://csrc.nist.gov/groups/ST/post-quantum-crypto/>. Until recently, ETSI (the European Telecommunications Standards Institute) had a Quantum-Safe Cryptography (QSC) Industry Specification Group (ISG) that worked on specifying post-quantum algorithms; see <http://www.etsi.org/technologies-clusters/technologies/quantum-safe-cryptography> for results from this work.

1.6. Not Covered: Quantum Key Exchange

Other than in this section, this document does not cover "quantum key exchange", also called "quantum cryptography". The field of quantum key exchange uses quantum effects in order to secure communication between users. Quantum key exchange is not related to cryptanalysis.

2. Brief Introduction to Quantum Computers

A quantum computer is a computer that uses quantum bits (qubits) in quantum circuits to perform calculations. Quantum computers also use classical bits and regular circuits: most calculations in a quantum computer are a mix of classical and quantum bits and circuits. For example, classical bits could be used for error correction or controlling the behavior of physical components of the quantum computer.

A basic principle that makes it possible to speed up calculations on qubits in quantum computers is quantum superposition. Informally, similarly to waves in classical physics, arbitrary number of quantum states can be added together and result will be another valid quantum state. That means that, for example, two qubits could be in any quantum superposition of four states, three qubits in quantum superposition of eight states, and so on. Generally n qubits can be in quantum superposition of 2^n states.

The main challenge for quantum computing is to create and maintain a significantly large number of superposed qubits while performing quantum computations. Physical components of quantum computers that are non-ideal results in the destruction of qubit state over time; this is the source of errors in quantum computation. See Section 3.1 for a description of how to overcome this problem.

A good description of different aspects of calculations on quantum computer could be found in [EstimatingPreimage].

A separate question is a measurement of a quantum state. Due to uncertainty of the state, the measurement process is stochastic. That means that in order to get the correct measurement one should run several consequent calculations and corresponding measurement in

order to the expected value which is considered as a result of measurement.

@@@@@ Discuss measurements and how they have to be done with correlated qubits.

2.1. Quantum Computers that Recover Cryptographic Keys

Quantum computers are expected to be useful in the future for some problems that take up too many resources on a large classical computer. However, this document only discusses how they might recover cryptographic keys faster than classical computers. In order to recover cryptographic keys, a quantum computer needs to have a quantum circuit specifically designed for the type of key it is attempting to recover.

A quantum computer will need to have a circuit with thousands of qubits to be useful to recover the type and size keys that are in common use today. Smaller quantum computers (those with fewer qubits in superposition) are not useful for using Shor's algorithm (as discussed in Section 4.1) at all. That is, no one has devised a way to combine a bunch of smaller quantum computers to perform the same attacks on cryptographic keys via Shor's algorithm as a properly-sized quantum computer.

This is why this document uses the term "large, specialized quantum computer" when describing ones that can recover keys: there will certainly be small quantum computers built first, but those computers cannot recover the type and size keys that are in common use today. Further, there are already quantum computers that have many qubits but without the circuits needed to make those qubits useful for recovering cryptographic keys.

A straight-forward application of Shor's algorithm may not be the only way for large, specialized quantum computers to attack RSA keys. [LowResource] describes how to combine quantum computers with classical methods for recovering RSA keys at speeds faster than just using the classical methods.

3. Physical Designs for Quantum Computers

Quantum computers can be built using many different physical technologies. Deciding which physical technologies are best to pursue is an extremely active research topic. A few physical technologies (particularly trapped ions and neutral atoms, super-conduction using Josephson junctions, and nuclear magnetic resonance) are currently getting the most press, but other technologies are also showing promise.

One factor that is important to quantum computers that can be used for cryptanalysis is the speed of the operations (transformations) on qubits. Most of the estimates of speeds of these quantum computers assume that qubit operations will take about the same amount of time as operations in circuits that consist of classical gates and classical memory. Current quantum circuits are currently slower than classical circuits, but will certainly become faster as quantum computers are developed in the future.

Note that some current quantum computer research uses bits that are not fully entangled, and this will greatly affect their ability to make useful quantum calculations.

3.1. Qubits, Error Detection, and Error Correction

Researchers building small quantum computers have discovered that calculating the superposition of qubits often has a large rate of error, and that error rate increases rapidly over time. Performing quantum calculations such as those needed to recover cryptographic keys is not feasible with the current state of quantum computers.

In the future, actual quantum calculations will be performed on "logical qubits", that is, after the application of error correction codes on physical qubits. Thus, the number of physical qubits will be higher than the number of logical qubits, depending on the parameters of the error correction code, which in turn depends on the parameters of a technology used for a physical implementation of qubits. Currently, it is estimated that it takes hundreds or thousands of physical qubits to make a logical qubit. @@@@@ Need reference for this statement.

@@@@@ Lots more material should go here. We will need recent references for how many physical qubits are needed for each corrected qubit. It’s OK if this section has lots of references, but hopefully they don’t contradict each other.

3.2. Promising Physical Designs for Quantum Computers

@@@@@ It would be useful to have maybe two paragraphs about each physical design that is being actively pursued.

3.3. Challenges for Physical Designs

Different designs have different challenges to overcome before the physical technology can be scaled enough to build a useful large, specialized quantum computer. Some of those challenges include the following. (Note that some items on this list apply only to some of the physical technologies.)

Temperature: Getting stable operation without extreme cooling is difficult for many of the proposed technologies. The definition of "extreme" is different for different low-temperature technologies.

Stabilization: The length of time every qubit in a circuit holds is value

Quantum control: Coherence and reproducibility of qubits

Error detection and correction: Getting accurate results through simultaneous detection of bit-flip and phase-flip. See Section 3.1 for a longer description of this.

Substrate: The material on which the qubit circuits are built. This has a large effect on the stability of the qubits.

Particles: The atoms or sub-atomic particles used to make the qubits

Scalability: The ability to handle the number of physical qubits needed for the desired the circuit

Architecture: Ability to change quantum gates in a circuit

4. Quantum Computers and Public Key Cryptography

The area of quantum computing that has generated the most interest in the cryptographic community is the ability of quantum computers to find the private keys in encryption and signature algorithms based on discrete logarithms using exponentially fewer operations than classical computers would need to use.

As described in [RFC3766], it is widely believed that factoring large numbers and finding discrete logs using classical computers increases with the exponential size of the key. [RFC3766] describes in detail how classical computers can be used to determine keys; even though that RFC is over a decade old, no significant changes have been made to the process of classical attacks on RSA and Diffie-Hellman. @@@@@ CFRG: is that true? Does RFC 3766 need to be updated?

Shor’s algorithm shows that these problems can be solved on quantum computers in polynomial time, meaning that the speed of finding the keys is a polynomial function (with reasonable-sized coefficients) based on the size of the keys, which would require significantly fewer steps than a classical computer. The definitive paper on Shor’s algorithm is [Shor97].

4.1. Explanation of Shor's Algorithm

@@@@@ Pointers to understandable articles would be good here.

@@@@@ Describe period-finding and why it applies to finding prime factors and discrete logs.

@@@@@ Give the steps for applying Shor's algorithm to 2048-bit RSA. Describe how many rounds of the quantum subroutine would likely be needed. Describe how many rounds of the classical loop would likely be needed.

[ResourceElliptic] gives concrete estimates of the resources needed to build a quantum computer to compute elliptic curve discrete logarithms. It shows that for the common P-256 elliptic curve, 2330 logical qubits and over 10^11 Toffoli gates.

[PrimeFactAnneal] describes a method of converting the integer factorization problem to one that can be executed on an adiabatic quantum computer. Adiabatic quantum computers are already available today, such as those from D-Wave Systems. Note that this method is not a way to run Shor's algorithm on an adiabatic quantum computer.

4.2. Properties of Large, Specialized Quantum Computers Needed for Recovering RSA Public Keys

Researchers have built small quantum computers that implement Shor's algorithm, factoring numbers with four or five bits. These are used to show that Shor's algorithm is possible to realize in actual hardware. (Note, however, that [PretendingFactor] indicates that these experiments may have taken shortcuts that prevent them from indicating real Shor designs.)

@@@@@ References are needed here. Did they implement all of Shor's algorithm, including the looping logic in the classical part and the looping logic in the quantum part?

According to [GidneyEkera], a quantum computer that can determine the private keys for 2048-bit RSA would require 20 million correlated qubits. The paper estimates a similar order of size for a quantum computer that could determine the private key for 256-bit elliptic curve algorithms.

5. Quantum Computers and Symmetric Key Cryptography

Section 4 is about Shor's algorithm and compromises to public key cryptography. There is a second quantum computing algorithm, Grover's algorithm, that is often mentioned at the same time as

Shor's algorithm. With respect to cryptanalysis, however, Grover's algorithm applies to tasks of finding a preimage, including tasks of finding a secret key of a symmetric algorithm such as AES if there is knowledge of plaintext-ciphertext pairs. The definitive paper on Grover's algorithm is by Grover: [Grover96]. Grover later wrote a more accessible paper about the algorithm in [QuantumSearch].

Grover's algorithm gives a way to search for keys to symmetric algorithms in the square root of the time that a normal exhaustive search would take. Thus, a large, specialized quantum computer that implements Grover's algorithm could find a secret AES-128 key in about 2^64 steps instead of the 2^128 steps that would be required for a classical computer.

When it appears that it is feasible to build a large, specialized quantum computer that can defeat a particular symmetric algorithm at a particular key size, the proper response would be to use keys with twice as many bits. That is, if one is using the AES-128 algorithm and there is a concern that an adversary might be able to build a large, specialized quantum computer that is designed to attack AES-128 keys, move to an algorithm that has keys twice as long as AES-128, namely AES-256 (the block size used is not significant here).

By some estimates, large specialized quantum computers that implement Grover's algorithm might be built before ones that implement Shor's algorithm are. There are two primary reasons for this:

- Grover's algorithm is likely to be useful in areas other than cryptography. For example, a large, specialized quantum computer that implements Grover's algorithm might help create medicines by speeding up complex problems that involve how proteins fold. @@@@@ Add more likely examples and references here.

- A large, specialized quantum computer that can recover AES-128 keys will likely be much smaller (and thus easier to build) than one that implements Shor's algorithm for 256-bit elliptic curves or 2048-bit RSA/DSA keys.

There are arguments against the likelihood of building computers using Grover's algorithm to break AES-128. As described in [FindCollisions]:

- Breaking AES-128 with Grover's method could be infeasible due to inherent inefficiencies in the algorithm. For example, the overhead of the quantum operations in the algorithm might be huge when compared to non-quantum operations.

- o Grover's algorithm has not been parallelized in a quantum computer, so the 2^64 steps must be done serially. Unless the speed of quantum computations become as fast as current classical computers, this will make doing all the calculations needed to break an AES-128 key take so long as to be infeasible.

5.1. Properties of Large, Specialized Quantum Computers Needed for Recovering Symmetric Keys

[ApplyingGrover] estimates that a quantum computer that can determine the secret keys for AES-128 would require 2953 correlated qubits and 2.74 * 2^86 gates.

[GoverSDES] shows how to use Grover's algorithm to search for keys in SDES, a simplifed version of the DES encryption algorithm.

5.2. Properties of Large, Specialized Quantum Computers for Computing Hash Collisions

@@@@@ More goes here. Also, discuss how Grover's algorithm does not appear to be useful for computing preimages (or say how it might be used).

6. Predicting When Useful Cryptographic Attacks Will Be Feasible

If quantum computers that perform useful cryptographic attacks can be built in the future, many organizations will want to start using post-quantum algorithms well before those computers can be built. However, given how few implementations of such quantum computers exist (even for tiny keys), it is impossible to predict with any accuracy when quantum computers that perform useful cryptographic attacks will be feasible.

The term "useful" above is relative to the value of the material being protected by the cryptographic algorithm to the attacker. For example, if the quantum computer attacking a particular key costs US$100 billion to build, costs US$1 billion a year to run, and can extract only one key a year, it is possibly useful to some governments, but probably not useful for attacking the TLS key used to protect a small mail server. On the other hand, if later a similar computer costs US$1 billion to build, costs US$10 million a year to run, and can extract ten keys a year, many more keys become vulnerable.

[BeReady] gives a simple way to approach the calculation of when one needs to deploy post-quantum algorithms. In short, if the sum of how long you need your keys to be secure plus how long it takes to deploy new algorithms is longer than the length of time it will take for an

attacker to create a large, specialized quantum computer and use it against your keys, then you waited too long.

To date, few people have done systematic research that would give estimates for when useful quantum-based cryptographic attacks might be feasible, and at what cost. Without such research, it is easy to make wild guesses but those are not of much value to people having to decide when to start using post-quantum cryptography.

For example, in [NIST8105], NIST says "researchers working on building a quantum computer have estimated that it is likely that a quantum computer capable of recovering 2000-bit RSA in a matter of hours could be built by 2030 for a budget of about a billion dollars". However, the referenced link is to a YouTube video [MariantoniYoutube] where the researcher, Matteo Mariantoni, says "maybe you should not quote me on that". [NIST8105] gives no other references for predictions on cost and availability of useful cryptographic attacks with quantum computers.

6.1. Proposal: Public Measurements of Various Quantum Technologies

In order to get a rough idea of when useful cryptographic attacks with quantum computers may be feasible, researchers creating such computers can demonstrate them when they can recover keys an eighth the size of those in common use. That is, given that 2048-bit RSA, 256-bit elliptic curve, and AES-128 are common today, when a research team has a computer than can recover 256-bit RSA, 32-bit elliptic curve, or AES-128 where only 16 bits are unknown, they should demonstrate it.

Such a demonstration could easily be made fair with trusted representatives from the cryptographic community using verifiable means to pick the keys to recover, and verifying the time that it takes to recover each key. It might be interesting to run the same tests in classical computers at the same time to give perspective.

These demonstrations will have many benefits to those who have to decide when post-quantum algorithms should be deployed in various environments.

- Demonstrations will likely use designs that are considered most efficient. This in turn will cause greater focus research on choosing good design candidates.

- The results of the demonstrations will help focus on issues important to cryptanalysis, namely the cost of building the systems and the speed of breaking a single key.

- o Competing demonstrations will reveal where different research teams have made different optimizations from well-known designs.

- o Public demonstrations could expose designs that work only in limited cases that are uncommon in normal cryptographic practice. (For example, [PretendingFactor] claims that all current factorization experiments have taken advantage of using a classical computer that already knows the answer to design the quantum circuits.)

Note that this proposal would only give an idea of how public progress is being made on quantum computers. Well-funded military agencies (and possibly even criminal enterprises) could be way ahead of the publicly-visible computers. No one should rely on just the public measurements when deciding how safe their keys are against quantum computers.

7. IANA Considerations

None, and thus this section can be removed at final publication.

8. Security Considerations

This entire document is about cryptography, and thus about security.

See Section 1.1 for an important disclaimer about this document and security.

This document is meant to help the reader predict when to transition from using classical cryptographic algorithms to post-quantum algorithms. That decision is ultimately up to the reader, and must be made not only based on predictions of how quantum computing is progressing but also the value of every key that the user handles. For example, a financial institution using TLS to protect its customers' transactions will probably consider its keys more valuable than a small online store, and will thus be likely to begin the transition earlier.

9. Acknowledgements

The list here is meant to acknowledge input to this document. The people listed here do not necessarily agree with ideas presented.

Many sections of text were contributed by Grigory Marshalko and Stanislav Smyshlyaev.

Some of the ideas in this document come from Denis Butin, Philip Lafrance, Hilarie Orman, and Tomofumi Okubo.

10. References

10.1. Normative References

[Grover96]
Grover, L., "A fast quantum mechanical algorithm for database search", 1996, <https://arxiv.org/abs/quant-ph/9605043>.

[Shor97] Shor, P., "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", 1997, <http://epubs.siam.org/doi/pdf/10.1137/S0097539795293172>.

10.2. Informative References

[ApplyingGrover]
Grassl, M., Langenberg, B., Roetteler, M., and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates", 2015, <https://arxiv.org/abs/1512.04965>.

[BeReady] Mosca, M., "Cybersecurity in an era with quantum computers: will we be ready?", 2015, <http://eprint.iacr.org/2015/1075>.

[EstimatingPreimage]
Amy, M., Di Matteo, O., Gheorghiu, V., Mosca, M., Parent, A., and J. Schanck, "Estimating the cost of generic quantum pre-image attacks on SHA-2 and SHA-3", 2016, <https://eprint.iacr.org/2016/992>.

[FindCollisions]
Bernstein, D., "Quantum algorithms to find collisions", 2017, <https://blog.cr.yp.to/20171017-collisions.html>.

[GidneyEkera]
Gidney, C. and M. Ekera, "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits", 2019, <https://arxiv.org/abs/1905.09749>.

[GoverSDES]
Denisenko, D. and M. Nikitenkova, "Application of Grover's Quantum Algorithm for SDES Key Searching", 2018.

[LowResource]
Bernstein, D., Fiassse, J., and M. Mosca, "A low-resource quantum factoring algorithm", 2017, <https://eprint.iacr.org/2017/352.pdf>.

[MariantoniYoutube]
Mariantoni, M., "Building a Superconducting Quantum Computer", 2014, <https://www.youtube.com/watch?v=wWHAs--HA1c>.

[NielsenChuang]
Nielsen, M. and I. Chuang, "Quantum Computation and Quantum Information, 10th Anniversary Edition", ISBN 97801-107-00217-3 , 2010.

[NIST8105]
Chen, L. and et. al, "Report on Post-Quantum Cryptography", 2016, <http://nvlpubs.nist.gov/nistpubs/ir/2016/NIST.IR.8105.pdf>.

[PretendingFactor]
Smolin, J., Vargo, A., and J. Smolin, "Pretending to factor large numbers on a quantum computer", 2013, <https://arxiv.org/abs/1301.7007>.

[PrimeFactAnneal]
Jiang, S., Britt, K., McCaskey, A., Humble, T., and S. Kais, "Quantum Annealing for Prime Factorization", 2018, <https://www.nature.com/articles/s41598-018-36058-z>.

[ProgressProspects]
The National Acadamies Sciences Engineering Medicine, "Quantum Computing: Progress and Prospects (2019)", 2019, <https://www.nap.edu/catalog/25196/quantum-computing-progress-and-prospects>.

[QCPolicy]
Princeton University Center for Information Technology Policy, "Implications of Quantum Computing for Encryption Policy", 2019, <https://carnegieendowment.org/2019/04/25/implications-of-quantum-computing-for-encryption-policy-pub-78985>.

[QuantumSearch]
Grover, L., "From Schrodinger’s Equation to the Quantum Search Algorithm", 2001, <https://arxiv.org/abs/quant-ph/0109116>.

[ResourceElliptic]
Roetteler, M., Naehrig, M., Svore, K., and K. Lauter, "Quantum Resource Estimates for Computing Elliptic Curve Discrete Logarithms", 2017, <https://eprint.iacr.org/2017/598>.

[RFC3766] Orman, H. and P. Hoffman, "Determining Strengths For Public Keys Used For Exchanging Symmetric Keys", BCP 86, RFC 3766, DOI 10.17487/RFC3766, April 2004, <https://www.rfc-editor.org/info/rfc3766>.

[Turing50Youtube]
Vazirani, U., Aharonov, D., Gambetta, J., Martinis, J., and A. Yao, "Quantum Computing: Far Away? Around the Corner?", 2017, <https://www.youtube.com/watch?v=SzfJRR5JrgQ>.

Author's Address

Paul Hoffman
ICANN

Email: paul.hoffman@icann.org

CFRG
Internet-Draft
Intended status: Informational
Expires: December 2, 2019

S. Smyshlyaev, Ed.
CryptoPro

May 31, 2019

Re-keying Mechanisms for Symmetric Keys
draft-irtf-cfrg-re-keying-17

Abstract

A certain maximum amount of data can be safely encrypted when encryption is performed under a single key. This amount is called "key lifetime". This specification describes a variety of methods to increase the lifetime of symmetric keys. It provides two types of re-keying mechanisms based on hash functions and on block ciphers, that can be used with modes of operations such as CTR, GCM, CBC, CFB and OMAC.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

A certain maximum amount of data can be safely encrypted when encryption is performed under a single key. Hereinafter this amount will be referred to as "key lifetime". The need for such a limitation is dictated by the following methods of cryptanalysis:

1. Methods based on the combinatorial properties of the used block cipher mode of operation

   These methods do not depend on the underlying block cipher. Common modes restrictions derived from such methods are of order 2^{n/2}, where n is a block size defined in Section 3. [Sweet32] is an example of attack that is based on such methods.

2. Methods based on side-channel analysis issues

   In most cases these methods do not depend on the used encryption modes and weakly depend on the used block cipher features. Limitations resulting from these considerations are usually the most restrictive ones. [TEMPEST] is an example of attack that is based on such methods.

3. Methods based on the properties of the used block cipher

   The most common methods of this type are linear and differential cryptanalysis [LDC]. In most cases these methods do not depend on the used modes of operation. In case of secure block ciphers, bounds resulting from such methods are roughly the same as the natural bounds of 2^n, and are dominated by the other bounds above. Therefore, they can be excluded from the considerations here.

As a result, it is important to replace a key when the total size of the processed plaintext under that key approaches the lifetime limitation. A specific value of the key lifetime should be determined in accordance with some safety margin for protocol security and the methods outlined above.

Suppose L is a key lifetime limitation in some protocol P. For simplicity, assume that all messages have the same length m. Hence,

the number of messages q that can be processed with a single key K should be such that m * q <= L. This can be depicted graphically as a rectangle with sides m and q which is enclosed by area L (see Figure 1).

```
+-------------------------+
|                       L |
| +--------m----------+   |
| |===================|   |
| |===================|   |
| q===================|   |       m * q <= L
| |===================|   |
| |===================|   |
| +-------------------+   |
+-------------------------+
```

Figure 1: Graphic display of the key lifetime limitation

In practice, such amount of data that corresponds to limitation L may not be enough. The simplest and obvious way in this situation is a regular renegotiation of an initial key after processing this threshold amount of data L. However, this reduces the total performance, since it usually entails termination of application data transmission, additional service messages, the use of random number generator and many other additional calculations, including resource-intensive public key cryptography.

For the protocols based on block ciphers or stream ciphers a more efficient way to increasing the key lifetime is to use various re-keying mechanisms. This specification considers only the case of re-keying mechanisms for block ciphers, while re-keying mechanisms typical for stream ciphers (e.g., [Pietrzak2009], [FPS2012]) case go beyond the scope of this document.

Re-keying mechanisms can be applied on the different protocol levels: on the block cipher level (this approach is known as fresh re-keying and is described, for instance, in [FRESHREKEYING]), on the block cipher mode of operation level (see Section 6), on the protocol level above the block cipher mode of operation (see Section 5). The usage of the first approach is highly inefficient due to the key changing after processing each message block. Moreover, fresh re-keying mechanisms can change the block cipher internal structure, and, consequently, can require the additional security analysis for each particular block cipher. As a result, this approach depends on particular primitive properties and can not be applied to any

arbitrary block cipher without additional security analysis, therefore, fresh re-keying mechanisms go beyond the scope of this document.

Thus, this document contains the list of recommended re-keying mechanisms that can be used in the symmetric encryption schemes based on the block ciphers. These mechanisms are independent from the particular block cipher specification and their security properties rely only on the standard block cipher security assumption.

This specification presents two basic approaches to extend the lifetime of a key while avoiding renegotiation that were introduced in [AAOS2017]:

1. External re-keying

   External re-keying is performed by a protocol, and it is independent of the underlying block cipher and the mode of operation. External re-keying can use parallel and serial constructions. In the parallel case, data processing keys K^1, K^2, ... are generated directly from the initial key K independently of each other. In the serial case, every data processing key depends on the state that is updated after the generation of each new data processing key.

   As a generalization of external parallel re-keying an external tree-based mechanism can be considered. It is specified in the Section 5.2.3 and can be viewed as the [GGM] tree generalization. Similar constructions are used in the one-way tree mechanism ([OWT]) and [AESDUKPT] standard.

2. Internal re-keying

   Internal re-keying is built into the mode, and it depends heavily on the properties of the mode of operation and the block size.

The re-keying approaches extend the key lifetime for a single initial key by providing the possibility to limit the leakages (via side channels) and by improving combinatorial properties of the used block cipher mode of operation.

In practical applications, re-keying can be useful for protocols that need to operate in hostile environments or under restricted resource conditions (e.g., that require lightweight cryptography, where ciphers have a small block size, that imposes strict combinatorial limitations). Moreover, mechanisms that use external or internal re-keying may provide some protection against possible future attacks (by limiting the number of plaintext-ciphertext pairs that an

adversary can collect) and some properties of forward or backward security (meaning that past or future data processing keys remain secure even if the current key is compromised, see for more details [AbBell]). External or internal re-keying can be used in network protocols as well as in the systems for data-at-rest encryption.

Depending on the concrete protocol characteristics there might be situations in which both external and internal re-keying mechanisms (see Section 7) can be applied. For example, the similar approach was used in the Taha's tree construction (see [TAHA]).

Note that there are key updating (key regression) algorithms (e.g., [FKK2005] and [KMNT2003]) which are called "re-keying" as well, but they pursue the goal different from increasing key lifetime. Therefore, key regression algorithms are excluded from the considerations here.

This document represents the consensus of the Crypto Forum Research Group (CFRG).

2. Conventions Used in This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Basic Terms and Definitions

This document uses the following terms and definitions for the sets and operations on the elements of these sets:

V* the set of all bit strings of a finite length (hereinafter referred to as strings), including the empty string;

V_s the set of all bit strings of length s, where s is a non-negative integer;

|X| the bit length of the bit string X;

A | B concatenation of strings A and B both belonging to V*, i.e., a string in V_{|A|+|B|}, where the left substring in V_|A| is equal to A, and the right substring in V_|B| is equal to B;

(xor) exclusive-or of two bit strings of the same length;

Z_{2^n} ring of residues modulo 2^n;

Int_s: V_s -> Z_{2^s} the transformation that maps a string a = (a_s, ... , a_1) in V_s into the integer Int_s(a) = 2^{s-1} * a_s + ... + 2 * a_2 + a_1 (the interpretation of the binary string as an integer);

Vec_s: Z_{2^s} -> V_s the transformation inverse to the mapping Int_s (the interpretation of an integer as a binary string);

MSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ... , a_1) in V_s into the string MSB_i(a) = (a_s, ... , a_{s-i+1}) in V_i (most significant bits);

LSB_i: V_s -> V_i the transformation that maps the string a = (a_s, ... , a_1) in V_s into the string LSB_i(a) = (a_i, ... , a_1) in V_i (least significant bits);

Inc_c: V_s -> V_s the transformation that maps the string a = (a_s, ... , a_1) in V_s into the string Inc_c(a) = MSB_{|a|-c}(a) | Vec_c(Int_c(LSB_c(a)) + 1(mod 2^c)) in V_s (incrementing the least significant c bits of the bit string, regarded as the binary representation of an integer);

a^s the string in V_s that consists of s 'a' bits;

E_{K}: V_n -> V_n the block cipher permutation under the key K in V_k;

ceil(x) the smallest integer that is greater than or equal to x;

floor(x) the biggest integer that is less than or equal to x;

k the bit-length of the K; k is assumed to be divisible by 8;

n the block size of the block cipher (in bits); n is assumed to be divisible by 8;

b the number of data blocks in the plaintext P (b = ceil(|P|/n));

N the section size (the number of bits that are processed with one section key before this key is transformed).

A plaintext message P and the corresponding ciphertext C are divided into b = ceil(|P|/n) blocks, denoted P = P_1 | P_2 | ... | P_b and C = C_1 | C_2 | ... | C_b, respectively. The first b-1 blocks P_i and C_i are in V_n, for i = 1, 2, ... , b-1. The b-th blocks P_b, C_b may be an incomplete blocks, i.e., in V_r, where r <= n if not otherwise specified.

4. Choosing Constructions and Security Parameters

External re-keying is an approach assuming that a key is transformed after encrypting a limited number of entire messages. External re-keying method is chosen at the protocol level, regardless of the underlying block cipher or the encryption mode. External re-keying is recommended for protocols that process relatively short messages or for protocols that have a way to divide a long message into manageable pieces. Through external re-keying the number of messages that can be securely processed with a single initial key K is substantially increased without loss in message length.

External re-keying has the following advantages:

1. it increases the lifetime of an initial key by increasing the number of messages processed with this key;

2. it has minimal impact on performance, when the number of messages processed under one initial key is sufficiently large;

3. it provides forward and backward security of data processing keys.

However, the use of external re-keying has the following disadvantage: in case of restrictive key lifetime limitations the message sizes can become inconvenient due to impossibility of processing sufficiently large messages, so it could be necessary to perform additional fragmentation at the protocol level. E.g. if the key lifetime L is 1 GB and the message length m = 3 GB, then this message cannot be processed as a whole and it should be divided into three fragments that will be processed separately.

Internal re-keying is an approach assuming that a key is transformed during each separate message processing. Such procedures are integrated into the base modes of operations, so every internal re-keying mechanism is defined for the particular operation mode and the block size of the used cipher. Internal re-keying is recommended for protocols that process long messages: the size of each single message can be substantially increased without loss in number of messages that can be securely processed with a single initial key.

Internal re-keying has the following advantages:

1. it increases the lifetime of an initial key by increasing the size of the messages processed with one initial key;

2. it has minimal impact on performance;

3. internal re-keying mechanisms without a master key does not affect short messages transformation at all;

4. it is transparent (works like any mode of operation): does not require changes of IV's and restarting MACing.

However, the use of internal re-keying has the following disadvantages:

1. a specific method must not be chosen independently of a mode of operation;

2. internal re-keying mechanisms without a master key do not provide backward security of data processing keys.

Any block cipher modes of operations with internal re-keying can be jointly used with any external re-keying mechanisms. Such joint usage increases both the number of messages processed with one initial key and their maximum possible size.

If the adversary has access to the data processing interface the use of the same cryptographic primitives both for data processing and re-keying transformation decreases the code size but can lead to some possible vulnerabilities (the possibility of mounting a chosen-plaintext attack may lead to the compromise of the following keys). This vulnerability can be eliminated by using different primitives for data processing and re-keying, e.g., block cipher for data processing and hash for re-keying (see Section 5.2.2 and Section 5.3.2). However, in this case the security of the whole scheme cannot be reduced to standard notions like PRF or PRP, so security estimations become more difficult and unclear.

Summing up the above-mentioned issues briefly:

1. If a protocol assumes processing long records (e.g., [CMS]), internal re-keying should be used. If a protocol assumes processing a significant amount of ordered records, which can be considered as a single data stream (e.g., [TLS], [SSH]), internal re-keying may also be used.

2. For protocols which allow out-of-order delivery and lost records (e.g., [DTLS], [ESP]) external re-keying should be used as in this case records cannot be considered as a single data stream. If at the same time records are long enough, internal re-keying should be additionally used during each separate message processing.

For external re-keying:

1. If it is desirable to separate transformations used for data processing and for key update, hash function based re-keying should be used.

2. If parallel data processing is required, then parallel external re-keying should be used.

3. In case of restrictive key lifetime limitations external tree-based re-keying should be used.

For internal re-keying:

1. If the property of forward and backward security is desirable for data processing keys and if additional key material can be easily obtained for the data processing stage, internal re-keying with a master key should be used.

5. External Re-keying Mechanisms

This section presents an approach to increase the initial key lifetime by using a transformation of a data processing key (frame key) after processing a limited number of entire messages (frame). It provides external parallel and serial re-keying mechanisms (see [AbBell]). These mechanisms use initial key K only for frame keys generation and never use it directly for data processing. Such mechanisms operate outside of the base modes of operations and do not change them at all, therefore they are called "external re-keying" mechanisms in this document.

External re-keying mechanisms are recommended for usage in protocols that process quite small messages, since the maximum gain in increasing the initial key lifetime is achieved by increasing the number of messages.

External re-keying increases the initial key lifetime through the following approach. Suppose there is a protocol P with some mode of operation (base encryption or authentication mode). Let L1 be a key lifetime limitation induced by side-channel analysis methods (side-channel limitation), let L2 be a key lifetime limitation induced by methods based on the combinatorial properties of a used mode of operation (combinatorial limitation) and let q1, q2 be the total numbers of messages of length m, that can be safely processed with an initial key K according to these limitations.

Let L = min(L1, L2), q = min (q1, q2), q * m <= L. As L1 limitation is usually much stronger than L2 limitation (L1 < L2), the final key lifetime restriction is equal to the most restrictive limitation L1.

Thus, as displayed in Figure 2, without re-keying only q1 (q1 * m <= L1) messages can be safely processed.

```
                <--------m------->
                +----------------+ ^ ^
                |================| | |
                |================| | |
          K-->  |================| q1|
                |================| | |
                |==============L1| | |
                +----------------+ v |
                |                  | |
                |                  | |
                |                  |  q2
                |                  |  |
                |                  |  |
                |                  |  |
                |                  |  |
                |                  |  |
                |                  |  |
                |                  |  |
                |                  |  |
                |              L2|    |
                +----------------+    v
```

Figure 2: Basic principles of message processing without external re-keying

Suppose that the safety margin for the protocol P is fixed and the external re-keying approach is applied to the initial key K to generate the sequence of frame keys. The frame keys are generated in such a way that the leakage of a previous frame key does not have any impact on the following one, so the side channel limitation L1 goes off. Thus, the resulting key lifetime limitation of the initial key K can be calculated on the basis of a new combinatorial limitation L2'. It is proven (see [AbBell]) that the security of the mode of operation that uses external re-keying leads to an increase when compared to base mode without re-keying (thus, L2 < L2'). Hence, as displayed in Figure 3, the resulting key lifetime limitation in case of using external re-keying can be increased up to L2'.

```
                <--------m------->
         K      +----------------+
         |      |================|
         v      |================|
        K^1-->  |================|
         |      |================|
         |      |==============L1|
         |      +----------------+
         |      |================|
         v      |================|
        K^2-->  |================|
         |      |================|
         |      |==============L1|
         |      +----------------+
         |      |================|
         v      |================|
        ...     |       . . .    |
                |                |
                |                |
                |              L2|
                +----------------+
                |                |
               ...              ...
                |             L2'|
                +----------------+
```

Figure 3: Basic principles of message processing
with external re-keying

Note: the key transformation process is depicted in a simplified form. A specific approach (parallel and serial) is described below.

Consider an example. Let the message size in a protocol P be equal to 1 KB. Suppose L1 = 128 MB and L2 = 1 TB. Thus, if an external re-keying mechanism is not used, the initial key K must be renegotiated after processing 128 MB / 1 KB = 131072 messages.

If an external re-keying mechanism is used, the key lifetime limitation L1 goes off. Hence the resulting key lifetime limitation L2' can be set to more then 1 TB. Thus if an external re-keying mechanism is used, more then 1 TB / 1 KB = 2^30 messages can be processed before the initial key K is renegotiated. This is 8192 times greater than the number of messages that can be processed, when external re-keying mechanism is not used.

5.1. Methods of Key Lifetime Control

Suppose L is an amount of data that can be safely processed with one frame key. For i in {1, 2, ... , t} the frame key K^i (see Figure 4 and Figure 6) should be transformed after processing q_i messages, where q_i can be calculated in accordance with one of the following approaches:

Explicit approach:

q_i is such that |M^{i,1}| + ... + |M^{i,q_i}| <= L, |M^{i,1}| + ... + |M^{i,q_i+1}| > L.
This approach allows to use the frame key K^i in almost optimal way but it can be applied only in case when messages cannot be lost or reordered (e.g., TLS records).

Implicit approach:

q_i = L / m_max, i = 1, ... , t.
The amount of data processed with one frame key K^i is calculated under the assumption that every message has the maximum length m_max. Hence this amount can be considerably less than the key lifetime limitation L. On the other hand, this approach can be applied in case when messages may be lost or reordered (e.g., DTLS records).

Dynamic key changes:

We can organize the key change using the Protected Point to Point ([P3]) solution by building a protected tunnel between the endpoints in which the information about frame key updating can be safely passed across. This can be useful, for example, when we wish the adversary not to detect the key change during the protocol evaluation.

5.2. Parallel Constructions

External parallel re-keying mechanisms generate frame keys K^1, K^2, ... directly from the initial key K independently of each other.

The main idea behind external re-keying with a parallel construction is presented in Figure 4:

```
Maximum message size = m_max.
_____________________________________________________________________

                                   m_max
                             <----------------->
                   M^{1,1}   |===               |
                   M^{1,2}   |================  |
         +->K^1-->   ...             ...
         |         M^{1,q_1} |========          |
         |
         |
         |         M^{2,1}   |=================|
         |         M^{2,2}   |=====             |
K-----|->K^2-->   ...             ...
         |         M^{2,q_2} |==========        |
         |
        ...
         |         M^{t,1}   |============      |
         |         M^{t,2}   |==============    |
         +->K^t-->   ...             ...
                   M^{t,q_t} |==========        |

_____________________________________________________________________
```

Figure 4: External parallel re-keying mechanisms

The frame key K^i, i = 1, ... , t-1, is updated after processing a certain amount of messages (see Section 5.1).

5.2.1. Parallel Construction Based on a KDF on a Block Cipher

ExtParallelC re-keying mechanism is based on the key derivation function on a block cipher and is used to generate t frame keys as follows:

```
K^1 | K^2 | ... | K^t = ExtParallelC(K, t * k) = MSB_{t *
k}(E_{K}(Vec_n(0)) |
E_{K}(Vec_n(1)) | ... | E_{K}(Vec_n(R - 1))),
```

where R = ceil(t * k/n).

5.2.2. Parallel Construction Based on a KDF on a Hash Function

ExtParallelH re-keying mechanism is based on the key derivation function HKDF-Expand, described in [RFC5869], and is used to generate t frame keys as follows:

```
K^1 | K^2 | ... | K^t = ExtParallelH(K, t * k) = HKDF-Expand(K,
label, t * k),
```

where label is a string (may be a zero-length string) that is defined by a specific protocol.

5.2.3. Tree-based Construction

The application of external tree-based mechanism leads to the construction of the key tree with the initial key K (root key) at the 0-level and the frame keys K^1, K^2, ... at the last level as described in Figure 5.

```
                          K_root = K
                     ____________|____________
                    |           ...           |
                    V                         V
                  K{1,1}                   K{1,W1}
               _______|_______           _______|_______
              |      ...      |         |      ...      |
              V               V         V               V
           K{2,1}          K{2,W2}  K{2,(W1-1)*W2+1} K{2,W1*W2}
            __|__           __|__     __|__           __|__
           | ... |         | ... |   | ... |         | ... |
           V     V         V     V   V     V         V     V
        K{3,1}  ...      ...   ... ...   ...      ...   K{3,W1*W2*W3}

         ...                                          ...
        __|__                    ...                 __|__
       | ... |                                      | ... |
       V     V                                      V     V
   K{h,1}   K{h,Wh}         K{h,(W1*...*W{h-1}-1)*Wh+1}  K{h,W1*...*Wh}
     //       \\                                //       \\
   K^1       K^{Wh}        K^{(W1*...*W{h-1}-1)*Wh+1}     K^{W1*...*Wh}
```

Figure 5: External Tree-based Mechanism

The tree height h and the number of keys Wj, j in {1, ... , h}, which can be partitioned from "parent" key, are defined in accordance with a specific protocol and key lifetime limitations for the used derivation functions.

Each j-level key K{j,w}, where j in {1, ... , h}, w in {1, ... , W1 * ... * Wj}, is derived from the (j-1)-level "parent" key K{j-1,ceil(w/Wi)} (and other appropriate input data) using the j-th level

derivation function that can be based on the block cipher function or on the hash function and that is defined in accordance with a specific protocol.

The i-th frame K^i, i in {1, 2, ... , W1*...*Wh}, can be calculated as follows:

```
K^i = ExtKeyTree(K, i) = KDF_h(KDF_{h-1}(... KDF_1(K, ceil(i / (W2
* ... * Wh)) ... , ceil(i / Wh)), i),
```

where KDF_j is the j-th level derivation function that takes two arguments (the parent key value and the integer in range from 1 to W1 * ... * Wj) and outputs the j-th level key value.

The frame key K^i is updated after processing a certain amount of messages (see Section 5.1).

In order to create an efficient implementation, during frame key K^i generation the derivation functions KDF_j, j in {1, ... , h-1}, should be used only in case when ceil(i / (W{j+1} * ... * Wh)) != ceil((i - 1) / (W{j+1} * ... * Wh)); otherwise it is necessary to use previously generated value. This approach also makes it possible to take countermeasures against side channels attacks.

Consider an example. Suppose h = 3, W1 = W2 = W3 = W and KDF_1, KDF_2, KDF_3 are key derivation functions based on the KDF_GOSTR3411_2012_256 (hereafter simply KDF) function described in [RFC7836]. The resulting ExtKeyTree function can be defined as follows:

```
ExtKeyTree(K, i) = KDF(KDF(KDF(K, "level1", ceil(i / W^2)),
"level2", ceil(i / W)), "level3", i).
```

where i in {1, 2, ... , W^3}.

The structure similar to external tree-based mechanism can be found in Section 6 of [NISTSP800-108].

5.3. Serial Constructions

External serial re-keying mechanisms generate frame keys, each of which depends on the secret state (K*_1, K*_2, ..., see Figure 6) that is updated after the generation of each new frame key. Similar approaches are used in the [SIGNAL] protocol, in the [TLS] updating traffic keys mechanism and were proposed for use in the [U2F] protocol.

External serial re-keying mechanisms have the obvious disadvantage of the impossibility to be implemented in parallel, but they can be preferred if additional forward secrecy is desirable: in case all keys are securely deleted after usage, compromise of a current secret state at some time does not lead to a compromise of all previous secret states and frame keys. In terms of [TLS], compromise of application_traffic_secret_N does not compromise all previous application_traffic_secret_i, i < N.

The main idea behind external re-keying with a serial construction is presented in Figure 6:

```
Maximum message size = m_max.
_____________________________________________________________
                                        m_max
                                  <---------------->
                      M^{1,1}    |===              |
                      M^{1,2}    |=============== |
K*_1 = K --->K^1-->    ...             ...
  |                   M^{1,q_1} |========        |
  |
  |
  |                   M^{2,1}    |================|
  v                   M^{2,2}    |=====           |
K*_2 ------->K^2-->    ...             ...
  |                   M^{2,q_2} |==========      |
  |
 ...
  |                   M^{t,1}    |============    |
  v                   M^{t,2}    |=============   |
K*_t ------->K^t-->    ...             ...
                      M^{t,q_t} |==========      |


_____________________________________________________________
```

Figure 6: External serial re-keying mechanisms

The frame key K^i, i = 1, ... , t - 1, is updated after processing a certain amount of messages (see Section 5.1).

5.3.1. Serial Construction Based on a KDF on a Block Cipher

The frame key K^i is calculated using ExtSerialC transformation as follows:

```
K^i = ExtSerialC(K, i) =
MSB_k(E_{K*_i}(Vec_n(0)) |E_{K*_i}(Vec_n(1)) | ... |
E_{K*_i}(Vec_n(J - 1))),
```

where J = ceil(k / n), i = 1, ... , t, K*_i is calculated as follows:

```
K*_1 = K,

K*_{j+1} = MSB_k(E_{K*_j}(Vec_n(J)) | E_{K*_j}(Vec_n(J + 1)) |
... |
E_{K*_j}(Vec_n(2 * J - 1))),
```

where j = 1, ... , t - 1.

5.3.2. Serial Construction Based on a KDF on a Hash Function

The frame key K^i is calculated using ExtSerialH transformation as follows:

```
K^i = ExtSerialH(K, i) = HKDF-Expand(K*_i, label1, k),
```

where i = 1, ... , t, HKDF-Expand is the HMAC-based key derivation function, described in [RFC5869], K*_i is calculated as follows:

```
K*_1 = K,

K*_{j+1} = HKDF-Expand(K*_j, label2, k), where j = 1, ... , t - 1,
```

where label1 and label2 are different strings from V* that are defined by a specific protocol (see, for example, TLS 1.3 updating traffic keys algorithm [TLS]).

5.4. Using Additional Entropy during Re-keying

In many cases using additional entropy during re-keying won’t increase security, but may give a false sense of that, therefore one can rely on additional entropy only after conducting a deep security analysis. For example, good PRF constructions do not require additional entropy for the quality of keys, so in most cases there is no need for using additional entropy with external re-keying mechanisms based on secure KDFs. However, in some situations mixed-in entropy can still increase security in the case of a time-limited but complete breach of the system, when an adversary can access the frame keys generation interface, but cannot reveal master keys (e.g., when master keys are stored in an HSM).

For example, an external parallel construction based on a KDF on a Hash function with a mixed-in entropy can be described as follows:

K^i = HKDF-Expand(K, label_i, k),

where label_i is additional entropy that must be sent to the recipient (e.g., be sent jointly with encrypted message). The entropy label_i and the corresponding key K^i must be generated directly before message processing.

6. Internal Re-keying Mechanisms

This section presents an approach to increase the key lifetime by using a transformation of a data processing key (section key) during each separate message processing. Each message is processed starting with the same key (the first section key) and each section key is updated after processing N bits of message (section).

This section provides internal re-keying mechanisms called ACPKM (Advanced Cryptographic Prolongation of Key Material) and ACPKM-Master that do not use a master key and use a master key respectively. Such mechanisms are integrated into the base modes of operation and actually form new modes of operation, therefore they are called "internal re-keying" mechanisms in this document.

Internal re-keying mechanisms are recommended to be used in protocols that process large single messages (e.g., CMS messages), since the maximum gain in increasing the key lifetime is achieved by increasing the length of a message, while it provides almost no increase in the number of messages that can be processed with one initial key.

Internal re-keying increases the key lifetime through the following approach. Suppose protocol P uses some base mode of operation. Let L1 and L2 be a side channel and combinatorial limitations respectively and for some fixed amount of messages q let m1, m2 be the lengths of messages, that can be safely processed with a single initial key K according to these limitations.

Thus, by analogy with the Section 5 without re-keying the final key lifetime restriction, as displayed in Figure 7, is equal to L1 and only q messages of the length m1 can be safely processed.

```
                K
                |
                v
      ^ +----------------+-----------------------------------+
      | |==============L1|                                 L2|
      | |================|                                   |
      q |================|                                   |
      | |================|                                   |
      | |================|                                   |
      v +----------------+-----------------------------------+
        <-------m1------->
        <----------------------------m2----------------------->
```

Figure 7: Basic principles of message processing
without internal re-keying

Suppose that the safety margin for the protocol P is fixed and internal re-keying approach is applied to the base mode of operation. Suppose further that every message is processed with a section key, which is transformed after processing N bits of data, where N is a parameter. If q * N does not exceed L1 then the side channel limitation L1 goes off and the resulting key lifetime limitation of the initial key K can be calculated on the basis of a new combinatorial limitation L2'. The security of the mode of operation that uses internal re-keying increases when compared to base mode of operation without re-keying (thus, L2 < L2'). Hence, as displayed in Figure 8, the resulting key lifetime limitation in case of using internal re-keying can be increased up to L2'.

```
  K-----> K^1-------------> K^2 -----------> . . .
          |                 |
          v                 v
^ +---------------+---------------+------------------+--...--+
| |=============L1|=============L1|======          L2|    L2'|
| |===============|===============|======            |       |
q |===============|===============|====== . . .      |       |
| |===============|===============|======            |       |
| |===============|===============|======            |       |
v +---------------+---------------+------------------+--...--+
  <-------N------->
```

Figure 8: Basic principles of message processing
with internal re-keying

Note: the key transformation process is depicted in a simplified form. A specific approach (ACPKM and ACPKM-Master re-keying mechanisms) is described below.

Since the performance of encryption can slightly decrease for rather small values of N, the parameter N should be selected for a particular protocol as maximum possible to provide necessary key lifetime for the considered security models.

Consider an example. Suppose L1 = 128 MB and L2 = 10 TB. Let the message size in the protocol be large/unlimited (may exhaust the whole key lifetime L2). The most restrictive resulting key lifetime limitation is equal to 128 MB.

Thus, there is a need to put a limit on the maximum message size m_max. For example, if m_max = 32 MB, it may happen that the renegotiation of initial key K would be required after processing only four messages.

If an internal re-keying mechanism with section size N = 1 MB is used, more than L1 / N = 128 MB / 1 MB = 128 messages can be processed before the renegotiation of initial key K (instead of 4 messages in case when an internal re-keying mechanism is not used). Note that only one section of each message is processed with the section key K^i, and, consequently, the key lifetime limitation L1 goes off. Hence the resulting key lifetime limitation L2' can be set to more then 10 TB (in the case when a single large message is processed using the initial key K).

6.1. Methods of Key Lifetime Control

Suppose L is an amount of data that can be safely processed with one section key, N is a section size (fixed parameter). Suppose M^{i}_1 is the first section of message M^{i}, i = 1, ... , q (see Figure 9 and Figure 10), then the parameter q can be calculated in accordance with one of the following two approaches:

- Explicit approach:
  q_i is such that |M^{1}_1| + ... + |M^{q}_1| <= L, |M^{1}_1| + ... + |M^{q+1}_1| > L
  This approach allows to use the section key K^i in an almost optimal way but it can be applied only in case when messages cannot be lost or reordered (e.g., TLS records).

- Implicit approach:
  q = L / N.
  The amount of data processed with one section key K^i is calculated under the assumption that the length of every message

is equal or greater than section size N and so it can be considerably less than the key lifetime limitation L. On the other hand, this approach can be applied in case when messages may be lost or reordered (e.g., DTLS records).

6.2. Constructions that Do Not Require Master Key

This section describes the block cipher modes that use the ACPKM re-keying mechanism, which does not use a master key: an initial key is used directly for the data encryption.

6.2.1. ACPKM Re-keying Mechanisms

This section defines periodical key transformation without a master key, which is called ACPKM re-keying mechanism. This mechanism can be applied to one of the base encryption modes (CTR and GCM block cipher modes) for getting an extension of this encryption mode that uses periodical key transformation without a master key. This extension can be considered as a new encryption mode.

An additional parameter that defines functioning of base encryption modes with the ACPKM re-keying mechanism is the section size N. The value of N is measured in bits and is fixed within a specific protocol based on the requirements of the system capacity and the key lifetime. The section size N MUST be divisible by the block size n.

The main idea behind internal re-keying without a master key is presented in Figure 9:

```
   Section size = const = N,
   maximum message size = m_max.
   ____________________________________________________________________

                 ACPKM       ACPKM               ACPKM
          K^1 = K ---> K^2 ---...-> K^{l_max-1} ----> K^{l_max}
              |           |                  |            |
              |           |                  |            |
              v           v                  v            v
   M^{1} |==========|==========| ... |==========|=======:  |
   M^{2} |==========|==========| ... |===       |       :  |
     .         .          .        .       .          .  :
     :         :          :        :       :          :  :
   M^{q} |==========|==========| ... |==========|=====  :  |
                      section                            :
                    <---------->                       m_max
                       N bit
   ____________________________________________________________________
   l_max = ceil(m_max/N).
```

Figure 9: Internal re-keying without a master key

During the processing of the input message M with the length m in some encryption mode that uses ACPKM key transformation of the initial key K the message is divided into l = ceil(m / N) sections (denoted as M = M_1 | M_2 | ... | M_l, where M_i is in V_N for i in {1, 2, ... , l - 1} and M_l is in V_r, r <= N). The first section of each message is processed with the section key K^1 = K. To process the (i + 1)-th section of each message the section key K^{i+1} is calculated using ACPKM transformation as follows:

K^{i+1} = ACPKM(K^i) = MSB_k(E_{K^i}(D_1) | ... | E_{K^i}(D_J)),

where J = ceil(k/n) and D_1, D_2, ... , D_J are in V_n and are calculated as follows:

D_1 | D_2 | ... | D_J = MSB_{J * n}(D),

where D is the following constant in V_{1024}:

```
D = ( 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87
    | 88 | 89 | 8a | 8b | 8c | 8d | 8e | 8f
    | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97
    | 98 | 99 | 9a | 9b | 9c | 9d | 9e | 9f
    | a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7
    | a8 | a9 | aa | ab | ac | ad | ae | af
    | b0 | b1 | b2 | b3 | b4 | b5 | b6 | b7
    | b8 | b9 | ba | bb | bc | bd | be | bf
    | c0 | c1 | c2 | c3 | c4 | c5 | c6 | c7
    | c8 | c9 | ca | cb | cc | cd | ce | cf
    | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7
    | d8 | d9 | da | db | dc | dd | de | df
    | e0 | e1 | e2 | e3 | e4 | e5 | e6 | e7
    | e8 | e9 | ea | eb | ec | ed | ee | ef
    | f0 | f1 | f2 | f3 | f4 | f5 | f6 | f7
    | f8 | f9 | fa | fb | fc | fd | fe | ff)
```

N o t e : The constant D is such that D_1, ... , D_J are pairwise different for any allowed n and k values.

N o t e : The highest bit of each octet of the constant D is equal to 1. This condition is important, as in conjunction with a certain mode message length limitation it allows to prevent collisions of block cipher permutation inputs in cases of key transformation and message processing (for more details see Section 4.4 of [AAOS2017]).

6.2.2. CTR-ACPKM Encryption Mode

This section defines a CTR-ACPKM encryption mode that uses the ACPKM internal re-keying mechanism for the periodical key transformation.

The CTR-ACPKM mode can be considered as the base encryption mode CTR (see [MODES]) extended by the ACPKM re-keying mechanism.

The CTR-ACPKM encryption mode can be used with the following parameters:

- o 64 <= n <= 512;
- o 128 <= k <= 512;
- o the number c of bits in a specific part of the block to be incremented is such that 32 <= c <= 3 / 4 n, c is a multiple of 8;
- o the maximum message size m_max = n * 2^{c-1}.

The CTR-ACPKM mode encryption and decryption procedures are defined as follows:

```
+-----------------------------------------------------------------+
|  CTR-ACPKM-Encrypt(N, K, ICN, P)                                |
|-----------------------------------------------------------------|
|  Input:                                                         |
|  - section size N,                                              |
|  - initial key K,                                               |
|  - initial counter nonce ICN in V_{n-c},                        |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                 |
|  Output:                                                        |
|  - ciphertext C.                                                |
|-----------------------------------------------------------------|
|  1. CTR_1 = ICN | 0^c                                           |
|  2. For j = 2, 3, ... , b do                                    |
|         CTR_{j} = Inc_c(CTR_{j-1})                              |
|  3. K^1 = K                                                     |
|  4. For i = 2, 3, ... , ceil(|P| / N)                           |
|         K^i = ACPKM(K^{i-1})                                    |
|  5. For j = 1, 2, ... , b do                                    |
|         i = ceil(j * n / N),                                    |
|         G_j = E_{K^i}(CTR_j)                                    |
|  6. C = P (xor) MSB_{|P|}(G_1 | ... | G_b)                      |
|  7. Return C                                                    |
+-----------------------------------------------------------------+
```

```
+-----------------------------------------------------------------+
|  CTR-ACPKM-Decrypt(N, K, ICN, C)                                |
|-----------------------------------------------------------------|
|  Input:                                                         |
|  - section size N,                                              |
|  - initial key K,                                               |
|  - initial counter nonce ICN in V_{n-c},                        |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.               |
|  Output:                                                        |
|  - plaintext P.                                                 |
|-----------------------------------------------------------------|
|  1. P = CTR-ACPKM-Encrypt(N, K, ICN, C)                         |
|  2. Return P                                                    |
+-----------------------------------------------------------------+
```

The initial counter nonce ICN value for each message that is encrypted under the given initial key K must be chosen in a unique manner.

6.2.3. GCM-ACPKM Authenticated Encryption Mode

This section defines GCM-ACPKM authenticated encryption mode that uses the ACPKM internal re-keying mechanism for the periodical key transformation.

The GCM-ACPKM mode can be considered as the base authenticated encryption mode GCM (see [GCM]) extended by the ACPKM re-keying mechanism.

The GCM-ACPKM authenticated encryption mode can be used with the following parameters:

- o n in {128, 256};
- o 128 <= k <= 512;
- o the number c of bits in a specific part of the block to be incremented is such that 1 / 4 n <= c <= 1 / 2 n, c is a multiple of 8;
- o authentication tag length t;
- o the maximum message size m_max = min{n * (2^{c-1} - 2), 2^{n/2} - 1}.

The GCM-ACPKM mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  GHASH(X, H)                                                      |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - bit string X = X_1 | ... | X_m, X_1, ... , X_m in V_n.         |
|  Output:                                                          |
|  - block GHASH(X, H) in V_n.                                      |
|-------------------------------------------------------------------|
|  1. Y_0 = 0^n                                                     |
|  2. For i = 1, ... , m do                                         |
|         Y_i = (Y_{i-1} (xor) X_i) * H                             |
|  3. Return Y_m                                                    |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCTR(N, K, ICB, X)                                               |
|-------------------------------------------------------------------|
|  Input:                                                           |
```

```
|  - section size N,                                                  |
|  - initial key K,                                                   |
|  - initial counter block ICB,                                       |
|  - X = X_1 | ... | X_b.                                             |
|  Output:                                                            |
|  - Y in V_{|X|}.                                                    |
|---------------------------------------------------------------------|
|  1. If X in V_0 then return Y, where Y in V_0                       |
|  2. GCTR_1 = ICB                                                    |
|  3. For i = 2, ... , b do                                           |
|         GCTR_i = Inc_c(GCTR_{i-1})                                  |
|  4. K^1 = K                                                         |
|  5. For j = 2, ... , ceil(|X| / N)                                  |
|         K^j = ACPKM(K^{j-1})                                        |
|  6. For i = 1, ... , b do                                           |
|         j = ceil(i * n / N),                                        |
|         G_i = E_{K_j}(GCTR_i)                                       |
|  7. Y = X (xor) MSB_{|X|}(G_1 | ... | G_b)                          |
|  8. Return Y                                                        |
+---------------------------------------------------------------------+
```

```
+---------------------------------------------------------------------+
|  GCM-ACPKM-Encrypt(N, K, ICN, P, A)                                 |
|---------------------------------------------------------------------|
|  Input:                                                             |
|  - section size N,                                                  |
|  - initial key K,                                                   |
|  - initial counter nonce ICN in V_{n-c},                            |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max,                     |
|  - additional authenticated data A.                                 |
|  Output:                                                            |
|  - ciphertext C,                                                    |
|  - authentication tag T.                                            |
|---------------------------------------------------------------------|
|  1. H = E_{K}(0^n)                                                  |
|  2. ICB_0 = ICN | 0^{c-1} | 1                                     |
|  3. C = GCTR(N, K, Inc_c(ICB_0), P)                                 |
|  4. u = n * ceil(|C| / n) - |C|                                     |
|     v = n * ceil(|A| / n) - |A|                                     |
|  5. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |                 |
|                | Vec_{n/2}(|C|), H)                                 |
|  6. T = MSB_t(E_{K}(ICB_0) (xor) S)                                 |
|  7. Return C | T                                                    |
+---------------------------------------------------------------------+
```

```
+---------------------------------------------------------------------+
|  GCM-ACPKM-Decrypt(N, K, ICN, A, C, T)                              |
|---------------------------------------------------------------------|
```

```
   |  Input:                                                          |
   |  - section size N,                                               |
   |  - initial key K,                                                |
   |  - initial counter block ICN,                                    |
   |  - additional authenticated data A,                              |
   |  - ciphertext C = C_1 | ... | C_b, |C| <= m_max,                |
   |  - authentication tag T.                                         |
   |  Output:                                                         |
   |  - plaintext P or FAIL.                                          |
   |------------------------------------------------------------------|
   |  1. H = E_{K}(0^n)                                               |
   |  2. ICB_0 = ICN | 0^{c-1} | 1                                  |
   |  3. P = GCTR(N, K, Inc_c(ICB_0), C)                              |
   |  4. u = n * ceil(|C| / n) - |C|                                 |
   |     v = n * ceil(|A| / n) - |A|                                 |
   |  5. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |              |
   |               | Vec_{n/2}(|C|), H)                               |
   |  6. T' = MSB_t(E_{K}(ICB_0) (xor) S)                             |
   |  7. If T = T' then return P; else return FAIL                    |
   +------------------------------------------------------------------+
```

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements defined by the polynomial f as follows (by analogy with [GCM]):

n = 128: f = a^128 + a^7 + a^2 + a^1 + 1,

n = 256: f = a^256 + a^10 + a^5 + a^2 + 1.

The initial counter nonce ICN value for each message that is encrypted under the given initial key K must be chosen in a unique manner.

The key for computing values E_{K}(ICB_0) and H is not updated and is equal to the initial key K.

6.3. Constructions that Require Master Key

This section describes the block cipher modes that use the ACPKM-Master re-keying mechanism, which use the initial key K as a master key, so K is never used directly for data processing but is used for key derivation.

6.3.1. ACPKM-Master Key Derivation from the Master Key

This section defines periodical key transformation with a master key, which is called ACPKM-Master re-keying mechanism. This mechanism can be applied to one of the base modes of operation (CTR, GCM, CBC, CFB, OMAC modes) for getting an extension that uses periodical key transformation with a master key. This extension can be considered as a new mode of operation.

Additional parameters that define the functioning of modes of operation that use the ACPKM-Master re-keying mechanism are the section size N, the change frequency T* of the master keys K*_1, K*_2, ... (see Figure 10) and the size d of the section key material. The values of N and T* are measured in bits and are fixed within a specific protocol, based on the requirements of the system capacity and the key lifetime. The section size N MUST be divisible by the block size n. The master key frequency T* MUST be divisible by d and by n.

The main idea behind internal re-keying with a master key is presented in Figure 10:

```
Master key frequency T*,
section size N,
maximum message size = m_max.
_____________________________________________________________________

                        ACPKM                 ACPKM
             K*_1 = K----------> K*_2 ---------...-----> K*_l_max
            ___|___             ___|___                 ___|___
           |       |           |       |               |       |
           v  ...  v           v  ...  v               v  ...  v
         K[1]     K[t]     K[t+1]  K[2*t]  K[(l_max-1)t+1] K[l_max*t]
           |       |           |       |                |       |
           |       |           |       |                |       |
           v       v           v       v                v       v
M^{1}||======|...|======||======|...|======||...||======|...|==  : ||
M^{2}||======|...|======||======|...|======||...||======|...|====: ||
 ...  ||      |   |      ||      |   |      ||   ||      |   |    : ||
M^{q}||======|...|======||====  |...|      ||...||      |...|    : ||
       section                                                    :
      <------>                                                    :
        N bit                                                   m_max
_____________________________________________________________________
|K[i]| = d,
t = T* / d,
l_max = ceil(m_max / (N * t)).
```

Figure 10: Internal re-keying with a master key

During the processing of the input message M with the length m in some mode of operation that uses ACPKM-Master key transformation with the initial key K and the master key frequency T* the message M is divided into l = ceil(m / N) sections (denoted as M = M_1 | M_2 | ... | M_l, where M_i is in V_N for i in {1, 2, ... , l - 1} and M_l is in V_r, r <= N). The j-th section of each message is processed with the key material K[j], j in {1, ... , l}, |K[j]| = d, that is calculated with the ACPKM-Master algorithm as follows:

K[1] | ... | K[l] = ACPKM-Master(T*, K, d, l) = CTR-ACPKM-Encrypt (T*, K, 1^{n/2}, 0^{d*l}).

Note: the parameters d and l MUST be such that d * l <= n * 2^{n/2-1}.

6.3.2. CTR-ACPKM-Master Encryption Mode

This section defines a CTR-ACPKM-Master encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The CTR-ACPKM-Master encryption mode can be considered as the base encryption mode CTR (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CTR-ACPKM-Master encryption mode can be used with the following parameters:

- $64 <= n <= 512$;
- $128 <= k <= 512$;
- the number c of bits in a specific part of the block to be incremented is such that $32 <= c <= 3 / 4\ n$, c is a multiple of 8;
- the maximum message size $m\_max = \min\{N * (n * 2^{n/2-1} / k), n * 2^c\}$.

The key material K[j] that is used for one section processing is equal to K^j, $|K^j| = k$ bits.

The CTR-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-----------------------------------------------------------------+
|  CTR-ACPKM-Master-Encrypt(N, K, T*, ICN, P)                     |
|-----------------------------------------------------------------|
|  Input:                                                         |
|  - section size N,                                              |
|  - initial key K,                                               |
|  - master key frequency T*,                                     |
|  - initial counter nonce ICN in V_{n-c},                        |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.                 |
|  Output:                                                        |
|  - ciphertext C.                                                |
|-----------------------------------------------------------------|
|  1. CTR_1 = ICN | 0^c                                           |
|  2. For j = 2, 3, ... , b do                                    |
|         CTR_{j} = Inc_c(CTR_{j-1})                              |
|  3. l = ceil(|P| / N)                                           |
|  4. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                 |
|  5. For j = 1, 2, ... , b do                                    |
|         i = ceil(j * n / N),                                    |
|         G_j = E_{K^i}(CTR_j)                                    |
|  6. C = P (xor) MSB_{|P|}(G_1 | ... |G_b)                       |
|  7. Return C                                                    |
|-----------------------------------------------------------------+

+-----------------------------------------------------------------+
|  CTR-ACPKM-Master-Decrypt(N, K, T*, ICN, C)                     |
|-----------------------------------------------------------------|
|  Input:                                                         |
|  - section size N,                                              |
|  - initial key K,                                               |
|  - master key frequency T*,                                     |
|  - initial counter nonce ICN in V_{n-c},                        |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.               |
|  Output:                                                        |
|  - plaintext P.                                                 |
|-----------------------------------------------------------------|
|  1. P = CTR-ACPKM-Master-Encrypt(N, K, T*, ICN, C)              |
|  1. Return P                                                    |
+-----------------------------------------------------------------+
```

The initial counter nonce ICN value for each message that is encrypted under the given initial key must be chosen in a unique manner.

6.3.3. GCM-ACPKM-Master Authenticated Encryption Mode

This section defines a GCM-ACPKM-Master authenticated encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The GCM-ACPKM-Master authenticated encryption mode can be considered as the base authenticated encryption mode GCM (see [GCM]) extended by the ACPKM-Master re-keying mechanism.

The GCM-ACPKM-Master authenticated encryption mode can be used with the following parameters:

- o n in {128, 256};
- o 128 <= k <= 512;
- o the number c of bits in a specific part of the block to be incremented is such that 1 / 4 n <= c <= 1 / 2 n, c is a multiple of 8;
- o authentication tag length t;
- o the maximum message size m_max = min{N * (n * 2^{n/2-1} / k), n * (2^c - 2), 2^{n/2} - 1}.

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

The GCM-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------------+
|  GHASH(X, H)                                                      |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - bit string X = X_1 | ... | X_m, X_i in V_n for i in {1, ... ,m}|
|  Output:                                                          |
|  - block GHASH(X, H) in V_n                                       |
|-------------------------------------------------------------------|
|  1. Y_0 = 0^n                                                     |
|  2. For i = 1, ... , m do                                         |
|         Y_i = (Y_{i-1} (xor) X_i) * H                             |
|  3. Return Y_m                                                    |
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
```

```
| GCTR(N, K, T*, ICB, X)                                            |
|-------------------------------------------------------------------|
| Input:                                                            |
| - section size N,                                                 |
| - initial key K,                                                  |
| - master key frequency T*,                                        |
| - initial counter block ICB,                                      |
| - X = X_1 | ... | X_b.                                            |
| Output:                                                           |
| - Y in V_{|X|}.                                                   |
|-------------------------------------------------------------------|
| 1. If X in V_0 then return Y, where Y in V_0                      |
| 2. GCTR_1 = ICB                                                   |
| 3. For i = 2, ... , b do                                          |
|        GCTR_i = Inc_c(GCTR_{i-1})                                 |
| 4. l = ceil(|X| / N)                                              |
| 5. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                          |
| 6. For j = 1, ... , b do                                          |
|        i = ceil(j * n / N),                                       |
|        G_j = E_{K^i}(GCTR_j)                                      |
| 7. Y = X (xor) MSB_{|X|}(G_1 | ... | G_b)                        |
| 8. Return Y                                                       |
+-------------------------------------------------------------------+
```

```
+-------------------------------------------------------------------+
| GCM-ACPKM-Master-Encrypt(N, K, T*, ICN, P, A)                     |
|-------------------------------------------------------------------|
| Input:                                                            |
| - section size N,                                                 |
| - initial key K,                                                  |
| - master key frequency T*,                                        |
| - initial counter nonce ICN in V_{n-c},                           |
| - plaintext P = P_1 | ... | P_b, |P| <= m_max.                   |
| - additional authenticated data A.                                |
| Output:                                                           |
| - ciphertext C,                                                   |
| - authentication tag T.                                           |
|-------------------------------------------------------------------|
| 1. K^1 = ACPKM-Master(T*, K, k, 1)                                |
| 2. H = E_{K^1}(0^n)                                               |
| 3. ICB_0 = ICN | 0^{c-1} | 1                                    |
| 4. C = GCTR(N, K, T*, Inc_c(ICB_0), P)                            |
| 5. u = n * ceil(|C| / n) - |C|                                    |
|    v = n * ceil(|A| / n) - |A|                                    |
| 6. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |              |
|              | Vec_{n/2}(|C|), H)                                 |
| 7. T = MSB_t(E_{K^1}(ICB_0) (xor) S)                              |
| 8. Return C | T                                                   |
```

```
+-------------------------------------------------------------------+

+-------------------------------------------------------------------+
|  GCM-ACPKM-Master-Decrypt(N, K, T*, ICN, A, C, T)                 |
|-------------------------------------------------------------------|
|  Input:                                                           |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - initial counter nonce ICN in V_{n-c},                          |
|  - additional authenticated data A.                               |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max,                  |
|  - authentication tag T.                                          |
|  Output:                                                          |
|  - plaintext P or FAIL.                                           |
|-------------------------------------------------------------------|
|  1. K^1 = ACPKM-Master(T*, K, k, 1)                               |
|  2. H = E_{K^1}(0^n)                                              |
|  3. ICB_0 = ICN | 0^{c-1} | 1                                   |
|  4. P = GCTR(N, K, T*, Inc_c(ICB_0), C)                           |
|  5. u = n * ceil(|C| / n) - |C|                                   |
|     v = n * ceil(|A| / n) - |A|                                   |
|  6. S = GHASH(A | 0^v | C | 0^u | Vec_{n/2}(|A|) |               |
|               | Vec_{n/2}(|C|), H)                                |
|  7. T' = MSB_t(E_{K^1}(ICB_0) (xor) S)                            |
|  8. IF T = T' then return P; else return FAIL.                    |
+-------------------------------------------------------------------+
```

The * operation on (pairs of) the 2^n possible blocks corresponds to the multiplication operation for the binary Galois (finite) field of 2^n elements defined by the polynomial f as follows (by analogy with [GCM]):

n = 128: f = a^128 + a^7 + a^2 + a^1 + 1,

n = 256: f = a^256 + a^10 + a^5 + a^2 + 1.

The initial counter nonce ICN value for each message that is encrypted under the given initial key must be chosen in a unique manner.

6.3.4. CBC-ACPKM-Master Encryption Mode

This section defines a CBC-ACPKM-Master encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The CBC-ACPKM-Master encryption mode can be considered as the base encryption mode CBC (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CBC-ACPKM-Master encryption mode can be used with the following parameters:

- o 64 <= n <= 512;

- o 128 <= k <= 512;

- o the maximum message size m_max = N * (n * 2^{n/2-1} / k).

In the specification of the CBC-ACPKM-Master mode the plaintext and ciphertext must be a sequence of one or more complete data blocks. If the data string to be encrypted does not initially satisfy this property, then it MUST be padded to form complete data blocks. The padding methods are out of the scope of this document. An example of a padding method can be found in Appendix A of [MODES].

The key material K[j] that is used for the j-th section processing is equal to K^j, |K^j| = k bits.

We will denote by D_{K} the decryption function which is a permutation inverse to E_{K}.

The CBC-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-----------------------------------------------------------------+
|  CBC-ACPKM-Master-Encrypt(N, K, T*, IV, P)                      |
|-----------------------------------------------------------------|
|  Input:                                                         |
|  - section size N,                                              |
|  - initial key K,                                               |
|  - master key frequency T*,                                     |
|  - initialization vector IV in V_n,                             |
|  - plaintext P = P_1 | ... | P_b, |P_b| = n, |P| <= m_max.      |
|  Output:                                                        |
|  - ciphertext C.                                                |
|-----------------------------------------------------------------|
|  1. l = ceil(|P| / N)                                           |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                 |
|  3. C_0 = IV                                                    |
|  4. For j = 1, 2, ... , b do                                    |
|         i = ceil(j * n / N),                                    |
|         C_j = E_{K^i}(P_j (xor) C_{j-1})                        |
|  5. Return C = C_1 | ... | C_b                                  |
|-----------------------------------------------------------------+
```

```
+-----------------------------------------------------------------+
|  CBC-ACPKM-Master-Decrypt(N, K, T*, IV, C)                      |
|-----------------------------------------------------------------|
|  Input:                                                         |
|  - section size N,                                              |
|  - initial key K,                                               |
|  - master key frequency T*,                                     |
|  - initialization vector IV in V_n,                             |
|  - ciphertext C = C_1 | ... | C_b, |C_b| = n, |C| <= m_max.     |
|  Output:                                                        |
|  - plaintext P.                                                 |
|-----------------------------------------------------------------|
|  1. l = ceil(|C| / N)                                           |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)                 |
|  3. C_0 = IV                                                    |
|  4. For j = 1, 2, ... , b do                                    |
|         i = ceil(j * n / N)                                     |
|         P_j = D_{K^i}(C_j) (xor) C_{j-1}                        |
|  5. Return P = P_1 | ... | P_b                                  |
+-----------------------------------------------------------------+
```

The initialization vector IV for any particular execution of the encryption process must be unpredictable.

6.3.5. CFB-ACPKM-Master Encryption Mode

This section defines a CFB-ACPKM-Master encryption mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The CFB-ACPKM-Master encryption mode can be considered as the base encryption mode CFB (see [MODES]) extended by the ACPKM-Master re-keying mechanism.

The CFB-ACPKM-Master encryption mode can be used with the following parameters:

- o $64 <= n <= 512$;

- o $128 <= k <= 512$;

- o the maximum message size $m\_max = N * (n * 2^{n/2-1} / k)$.

The key material K[j] that is used for the j-th section processing is equal to K^j, $|K^j| = k$ bits.

The CFB-ACPKM-Master mode encryption and decryption procedures are defined as follows:

```
+-------------------------------------------------------------+
|  CFB-ACPKM-Master-Encrypt(N, K, T*, IV, P)                  |
|-------------------------------------------------------------|
|  Input:                                                     |
|  - section size N,                                          |
|  - initial key K,                                           |
|  - master key frequency T*,                                 |
|  - initialization vector IV in V_n,                         |
|  - plaintext P = P_1 | ... | P_b, |P| <= m_max.             |
|  Output:                                                    |
|  - ciphertext C.                                            |
|-------------------------------------------------------------|
|  1. l = ceil(|P| / N)                                       |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)            |
|  3. C_0 = IV                                                |
|  4. For j = 1, 2, ... , b - 1 do                            |
|         i = ceil(j * n / N),                                |
|         C_j = E_{K^i}(C_{j-1}) (xor) P_j                    |
|  5. C_b = MSB_{|P_b|}(E_{K^l}(C_{b-1})) (xor) P_b           |
|  6. Return C = C_1 | ... | C_b                              |
|-------------------------------------------------------------+
```

```
+-------------------------------------------------------------+
|  CFB-ACPKM-Master-Decrypt(N, K, T*, IV, C)                  |
|-------------------------------------------------------------|
|  Input:                                                     |
|  - section size N,                                          |
|  - initial key K,                                           |
|  - master key frequency T*,                                 |
|  - initialization vector IV in V_n,                         |
|  - ciphertext C = C_1 | ... | C_b, |C| <= m_max.           |
|  Output:                                                    |
|  - plaintext P.                                             |
|-------------------------------------------------------------|
|  1. l = ceil(|C| / N)                                       |
|  2. K^1 | ... | K^l = ACPKM-Master(T*, K, k, l)            |
|  3. C_0 = IV                                                |
|  4. For j = 1, 2, ... , b - 1 do                            |
|         i = ceil(j * n / N),                                |
|         P_j = E_{K^i}(C_{j-1}) (xor) C_j                    |
|  5. P_b = MSB_{|C_b|}(E_{K^l}(C_{b-1})) (xor) C_b           |
|  6. Return P = P_1 | ... | P_b                              |
+-------------------------------------------------------------+
```

The initialization vector IV for any particular execution of the encryption process must be unpredictable.

6.3.6. OMAC-ACPKM-Master Authentication Mode

This section defines an OMAC-ACPKM-Master message authentication code calculation mode that uses the ACPKM-Master internal re-keying mechanism for the periodical key transformation.

The OMAC-ACPKM-Master mode can be considered as the base message authentication code calculation mode OMAC, which is also known as CMAC (see [RFC4493]), extended by the ACPKM-Master re-keying mechanism.

The OMAC-ACPKM-Master message authentication code calculation mode can be used with the following parameters:

- o n in {64, 128, 256};
- o 128 <= k <= 512;
- o the maximum message size m_max = N * (n * 2^{n/2-1} / (k + n)).

The key material K[j] that is used for one section processing is equal to K^j | K^j_1, where |K^j| = k and |K^j_1| = n.

The following is a specification of the subkey generation process of OMAC:

```
+-------------------------------------------------------------------+
| Generate_Subkey(K1, r)                                            |
|-------------------------------------------------------------------|
| Input:                                                            |
|  - key K1.                                                        |
|  Output:                                                          |
|  - key SK.                                                        |
|-------------------------------------------------------------------|
|   1. If r = n then return K1                                      |
|   2. If r < n then                                                |
|          if MSB_1(K1) = 0                                         |
|              return K1 << 1                                       |
|          else                                                     |
|              return (K1 << 1) (xor) R_n                           |
|                                                                   |
+-------------------------------------------------------------------+
```

Here R_n takes the following values:

- o n = 64: R_{64} = 0^{59} | 11011;

- o n = 128: R_{128} = 0^{120} | 10000111;

- o n = 256: R_{256} = 0^{145} | 10000100101.

The OMAC-ACPKM-Master message authentication code calculation mode is defined as follows:

```
+-------------------------------------------------------------------+
| OMAC-ACPKM-Master(K, N, T*, M)                                    |
|-------------------------------------------------------------------|
| Input:                                                            |
|  - section size N,                                                |
|  - initial key K,                                                 |
|  - master key frequency T*,                                       |
|  - plaintext M = M_1 | ... | M_b, |M| <= m_max.                   |
|  Output:                                                          |
|  - message authentication code T.                                 |
|-------------------------------------------------------------------|
| 1. C_0 = 0^n                                                      |
| 2. l = ceil(|M| / N)                                              |
| 3. K^1 | K^1_1 | ... | K^l | K^l_1 =                            |
|                  = ACPKM-Master(T*, K, (k + n), l)                |
| 4. For j = 1, 2, ... , b - 1 do                                   |
|        i = ceil(j * n / N),                                       |
|        C_j = E_{K^i}(M_j (xor) C_{j-1})                           |
| 5. SK = Generate_Subkey(K^l_1, |M_b|)                             |
| 6. If |M_b| = n then M*_b = M_b                                   |
|                  else M*_b = M_b | 1 | 0^{n - 1 -|M_b|}          |
| 7. T = E_{K^l}(M*_b (xor) C_{b-1} (xor) SK)                       |
| 8. Return T                                                       |
+-------------------------------------------------------------------+
```

7. Joint Usage of External and Internal Re-keying

Both external re-keying and internal re-keying have their own advantages and disadvantages discussed in Section 1. For instance, using external re-keying can essentially limit the message length, while in the case of internal re-keying the section size, which can be chosen as the maximal possible for operational properties, limits the amount of separate messages. Therefore, the choice of re-keying mechanism (either external or internal) depends on particular protocol features. However, some protocols may have features that require to take advantages provided by both external and internal re-keying mechanisms: for example, the protocol mainly transmits messages of small length, but it must additionally support very long messages processing. In such situations it is necessary to use

external and internal re-keying jointly, since these techniques negate each other's disadvantages.

For composition of external and internal re-keying techniques any mechanism described in Section 5 can be used with any mechanism described in Section 6.

For example, consider the GCM-ACPKM mode with external serial re-keying based on a KDF on a Hash function. Denote by a frame size the number of messages in each frame (in the case of implicit approach to the key lifetime control) for external re-keying.

Let L be a key lifetime limitation. The section size N for internal re-keying and the frame size q for external re-keying must be chosen in such a way that q * N must not exceed L.

Suppose that t messages (ICN_i, P_i, A_i), with initial counter nonce ICN_i, plaintext P_i and additional authenticated data A_i, will be processed before renegotiation.

For authenticated encryption of each message (ICN_i, P_i, A_i), i = 1, ..., t, the following algorithm can be applied:

```
1. j = ceil(i / q),
2. K^j = ExtSerialH(K, j),
3. C_i | T_i = GCM-ACPKM-Encrypt(N, K^j, ICN_i, P_i, A_i).
```

Note that nonces ICN_i, that are used under the same frame key, must be unique for each message.

8. Security Considerations

Re-keying should be used to increase "a priori" security properties of ciphers in hostile environments (e.g., with side-channel adversaries). If some efficient attacks are known for a cipher, it must not be used. So re-keying cannot be used as a patch for vulnerable ciphers. Base cipher properties must be well analyzed, because the security of re-keying mechanisms is based on the security of a block cipher as a pseudorandom function.

Re-keying is not intended to solve any post-quantum security issues for symmetric cryptography, since the reduction of security caused by Grover's algorithm is not connected with a size of plaintext transformed by a cipher - only a negligible (sufficient for key uniqueness) material is needed; and the aim of re-keying is to limit a size of plaintext transformed under one initial key.

Re-keying can provide backward security only if previous key material is securely deleted after usage by all parties.

9. IANA Considerations

This document does not require any IANA actions.

10. References

10.1. Normative References

[CMS] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://www.rfc-editor.org/info/rfc5652>.

[DTLS] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <http://www.rfc-editor.org/info/rfc6347>.

[ESP] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <http://www.rfc-editor.org/info/rfc4303>.

[GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf, November 2007.

[MODES] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, December 2001.

[NISTSP800-108] National Institute of Standards and Technology, "Recommendation for Key Derivation Using Pseudorandom Functions", NIST Special Publication 800-108, November 2008, <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>.

[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>.

[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/info/rfc4493>.

[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>.

[RFC7836] Smyshlyaev, S., Ed., Alekseev, E., Oshkin, I., Popov, V., Leontiev, S., Podobaev, V., and D. Belyavsky, "Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012", RFC 7836, DOI 10.17487/RFC7836, March 2016, <https://www.rfc-editor.org/info/rfc7836>.

[SSH] Ylonen, T. and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", RFC 4253, DOI 10.17487/RFC4253, January 2006, <http://www.rfc-editor.org/info/rfc4253>.

[TLS] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <http://www.rfc-editor.org/info/rfc8446>.

10.2. Informative References

[AAOS2017] Ahmetzyanova, L., Alekseev, E., Oshkin, I., and S. Smyshlyaev, "Increasing the Lifetime of Symmetric Keys for the GCM Mode by Internal Re-keying", Cryptology ePrint Archive Report 2017/697, 2017, <https://eprint.iacr.org/2017/697.pdf>.

[AbBell] Michel Abdalla and Mihir Bellare, "Increasing the Lifetime of a Key: A Comparative Analysis of the Security of Re-keying Techniques", ASIACRYPT2000, LNCS 1976, pp. 546-559, 2000.

[AESDUKPT] ANSI, "Retail Financial Services Symmetric Key Management - Part 3: Derived Unique Key Per Transaction", ANSI X9.24-3-2017, 2017.

[FKK2005] Fu, K., Kamara, S., and T. Kohno, "Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage", November 2005, <https://homes.cs.washington.edu/~yoshi/papers/KR/NDSS06.pdf>.

[FPS2012] Faust, S., Pietrzak, K., and j. Schipper, "Practical Leakage-Resilient Symmetric Cryptography", CHES2012 LNCS, vol. 7428, pp. 213-232,, 2012, <https://link.springer.com/content/pdf/10.1007%2F978-3-642-33027-8_13.pdf>.

[FRESHREKEYING] Dziembowski, S., Faust, S., Herold, G., Journault, A., Masny, D., and F. Standaert, "Towards Sound Fresh Re-Keying with Hard (Physical) Learning Problems", Cryptology ePrint Archive Report 2016/573, June 2016, <https://eprint.iacr.org/2016/573>.

[GGM] Goldreich, O., Goldwasser, S., and S. Micali, "How to Construct Random Functions", Journal of the Association for Computing Machinery Vol.33, No.4, pp. 792-807, October 1986, <http://www.wisdom.weizmann.ac.il/~/oded/X/ggm.pdf>.

[KMNT2003] Kim, Y., Maino, F., Narasimha, M., and G. Tsudik, "Secure Group Services for Storage Area Networks", IEEE Communication Magazine 41, pp. 92-99, 2003, <http://www.ics.uci.edu/~gts/paps/kmnt02.pdf>.

[LDC] Howard M. Heys, "A Tutorial on Linear and Differential Cryptanalysis", 2017, <http://www.cs.bc.edu/~straubin/crypto2017/heys.pdf>.

[OWT] Joye, M. and S. Yen, "One-Way Cross-Trees and Their Applications", DOI 10.1007/3-540-45664-3_25, February 2002, <https://link.springer.com/content/pdf/10.1007%2F3-540-45664-3_25.pdf>.

[P3] Peter Alexander, "Dynamic Key Changes on Encrypted Sessions", CFRG mail archive , December 2017, <https://www.ietf.org/mail-archive/web/cfrg/current/msg09401.html>.

[Pietrzak2009] Pietrzak, K., "A Leakage-Resilient Mode of Operation", EUROCRYPT2009 LNCS, vol. 5479, pp. 462-482,, 2009, <https://iacr.org/archive/eurocrypt2009/54790461/54790461.pdf>.

[SIGNAL] Perrin, T., Ed. and M. Marlinspike, "The Double Ratchet Algorithm", November 2016, <https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf>.

[Sweet32] Karthikeyan Bhargavan, Gaetan Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", Cryptology ePrint Archive Report 2016/798, 2016, <https://sweet32.info/SWEET32_CCS16.pdf>.

[TAHA] Taha, M. and P. Schaumont, "Key Updating for Leakage Resiliency With Application to AES Modes of Operation", DOI 10.1109/TIFS.2014.2383359, December 2014, <http://ieeexplore.ieee.org/document/6987331/>.

[TEMPEST] By Craig Ramsay, Jasper Lohuis, "TEMPEST attacks against AES. Covertly stealing keys for 200 euro", 2017, <https://www.fox-it.com/en/wp-content/uploads/sites/11/Tempest_attacks_against_AES.pdf>.

[U2F] Chang, D., Mishra, S., Sanadhya, S., and A. Singhl, "On Making U2F Protocol Leakage-Resilient via Re-keying.", Cryptology ePrint Archive Report 2017/721, August 2017, <https://eprint.iacr.org/2017/721.pdf>.

Appendix A. Test Examples

A.1. Test Examples for External Re-keying

A.1.1. External Re-keying with a Parallel Construction

```
External re-keying with a parallel construction based on AES-256
****************************************************************
k = 256
t = 128

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1

K^2:
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15
B8 02 92 32 D8 D3 8D 73 FE DC DD C6 C8 36 78 BD

K^3:
B6 40 24 85 A4 24 BD 35 B4 26 43 13 76 26 70 B6
5B F3 30 3D 3B 20 EB 14 D1 3B B7 91 74 E3 DB EC

...

K^126:
2F 3F 15 1B 53 88 23 CD 7D 03 FC 3D FD B3 57 5E
23 E4 1C 4E 46 FF 6B 33 34 12 27 84 EF 5D 82 23

K^127:
8E 51 31 FB 0B 64 BB D0 BC D4 C5 7B 1C 66 EF FD
97 43 75 10 6C AF 5D 5E 41 E0 17 F4 05 63 05 ED

K^128:
77 4F BF B3 22 60 C5 3B A3 8E FE B1 96 46 76 41
94 49 AF 84 2D 84 65 A7 F4 F7 2C DC A4 9D 84 F9
```

```
External re-keying with a parallel construction based on SHA-256
****************************************************************
k = 256
t = 128

label:
SHA2label

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
C1 A1 4C A0 30 29 BE 43 9F 35 3C 79 1A 51 48 57
26 7A CD 5A E8 7D E7 D1 B2 E2 C7 AF A4 29 BD 35

K^2:
03 68 BB 74 41 2A 98 ED C4 7B 94 CC DF 9C F4 9E
A9 B8 A9 5F 0E DC 3C 1E 3B D2 59 4D D1 75 82 D4

K^3:
2F D3 68 D3 A7 8F 91 E6 3B 68 DC 2B 41 1D AC 80
0A C3 14 1D 80 26 3E 61 C9 0D 24 45 2A BD B1 AE

...

K^126:
55 AC 2B 25 00 78 3E D4 34 2B 65 0E 75 E5 8B 76
C8 04 E9 D3 B6 08 7D C0 70 2A 99 A4 B5 85 F1 A1

K^127:
77 4D 15 88 B0 40 90 E5 8C 6A D7 5D 0F CF 0A 4A
6C 23 F1 B3 91 B1 EF DF E5 77 64 CD 09 F5 BC AF

K^128:
E5 81 FF FB 0C 90 88 CD E5 F4 A5 57 B6 AB D2 2E
94 C3 42 06 41 AB C1 72 66 CC 2F 59 74 9C 86 B3
```

A.1.2. External Re-keying with a Serial Construction

```
External re-keying with a serial construction based on AES-256
**************************************************************
AES 256 examples:
k = 256
t = 128
```

```
Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K*_1:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_2:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^2:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_3:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^3:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

...

K*_126:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^126:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_127:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^127:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86

K*_128:
64 7D 5C D5 1C 3D 62 98 BC 09 B1 D8 64 EC D9 B1
```

```
6F ED F5 D3 77 57 48 75 35 2B 5F 4D B6 5B E0 15

K^128:
66 B8 BD E5 90 6C EC DF FA 8A B2 FD 92 84 EB F0
51 16 8A B6 C8 A8 38 65 54 85 31 A5 D2 BA C3 86



External re-keying with a serial construction based on SHA-256
**************************************************************
k = 256
t = 128

Initial key:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

label1:
SHA2label1

label2:
SHA2label2

K*_1:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00

K^1:
2D A8 D1 37 6C FD 52 7F F7 36 A4 E2 81 C6 0A 9B
F3 8E 66 97 ED 70 4F B5 FB 10 33 CC EC EE D5 EC

K*_2:
14 65 5A D1 7C 19 86 24 9B D3 56 DF CC BE 73 6F
52 62 4A 9D E3 CC 40 6D A9 48 DA 5C D0 68 8A 04

K^2:
2F EA 8D 57 2B EF B8 89 42 54 1B 8C 1B 3F 8D B1
84 F9 56 C7 FE 01 11 99 1D FB 98 15 FE 65 85 CF

K*_3:
18 F0 B5 2A D2 45 E1 93 69 53 40 55 43 70 95 8D
70 F0 20 8C DF B0 5D 67 CD 1B BF 96 37 D3 E3 EB

K^3:
53 C7 4E 79 AE BC D1 C8 24 04 BF F6 D7 B1 AC BF
F9 C0 0E FB A8 B9 48 29 87 37 E1 BA E7 8F F7 92
```

```
...

K*_126:
A3 6D BF 02 AA 0B 42 4A F2 C0 46 52 68 8B C7 E6
5E F1 62 C3 B3 2F DD EF E4 92 79 5D BB 45 0B CA

K^126:
6C 4B D6 22 DC 40 48 0F 29 C3 90 B8 E5 D7 A7 34
23 4D 34 65 2C CE 4A 76 2C FE 2A 42 C8 5B FE 9A

K*_127:
84 5F 49 3D B8 13 1D 39 36 2B BE D3 74 8F 80 A1
05 A7 07 37 BA 15 72 E0 73 49 C2 67 5D 0A 28 A1

K^127:
57 F0 BD 5A B8 2A F3 6B 87 33 CF F7 22 62 B4 D0
F0 EE EF E1 50 74 E5 BA 13 C1 23 68 87 36 29 A2

K*_128:
52 F2 0F 56 5C 9C 56 84 AF 69 AD 45 EE B8 DA 4E
7A A6 04 86 35 16 BA 98 E4 CB 46 D2 E8 9A C1 09

K^128:
9B DD 24 7D F3 25 4A 75 E0 22 68 25 68 DA 9D D5
C1 6D 2D 2B 4F 3F 1F 2B 5E 99 82 7F 15 A1 4F A4
```

A.2. Test Examples for Internal Re-keying

A.2.1. Internal Re-keying Mechanisms that Do Not Require Master Key

```
CTR-ACPKM mode with AES-256
***************************
k = 256
n = 128
c = 64
N = 256

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Plain text P:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
```

```
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

ICN:
12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12
23 34 45 56 67 78 89 90 12 13 14 15 16 17 18 19

D_1:
00000:   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F

D_2:
00000:   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F

Section_1

Section key K^1:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Input block CTR_1:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 00

Output block G_1:
00000:   FD 7E F8 9A D9 7E A4 B8 8D B8 B5 1C 1C 9D 6D D0

Input block CTR_2:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 01

Output block G_2:
00000:   19 98 C5 71 76 37 FB 17 11 E4 48 F0 0C 0D 60 B2

Section_2

Section key K^2:
00000:   F6 80 D1 21 2F A4 3D F4 EC 3A 91 DE 2A B1 6F 1B
00010:   36 B0 48 8A 4F C1 2E 09 98 D2 E4 A8 88 E8 4F 3D

Input block CTR_3:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 02

Output block G_3:
00000:   E4 88 89 4F B6 02 87 DB 77 5A 07 D9 2C 89 46 EA

Input block CTR_4:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 03
```

```
Output block G_4:
00000:   BC 4F 87 23 DB F0 91 50 DD B4 06 C3 1D A9 7C A4

Section_3

Section key K^3:
00000:   8E B9 7E 43 27 1A 42 F1 CA 8E E2 5F 5C C7 C8 3B
00010:   1A CE 9E 5E D0 6A A5 3B 57 B9 6A CF 36 5D 24 B8

Input block CTR_5:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 04

Output block G_5:
00000:   68 6F 22 7D 8F B2 9C BD 05 C8 C3 7D 22 FE 3B B7

Input block CTR_6:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 05

Output block G_6:
00000:   C0 1B F9 7F 75 6E 12 2F 80 59 55 BD DE 2D 45 87

Section_4

Section key K^4:
00000:   C5 71 6C C9 67 98 BC 2D 4A 17 87 B7 8A DF 94 AC
00010:   E8 16 F8 0B DB BC AD 7D 60 78 12 9C 0C B4 02 F5

Block number 7:

Input block CTR_7:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 06

Output block G_7:
00000:   03 DE 34 74 AB 9B 65 8A 3B 54 1E F8 BD 2B F4 7D


The result G = G_1 | G_2 | G_3 | G_4 | G_5 | G_6 | G_7:
00000:   FD 7E F8 9A D9 7E A4 B8 8D B8 B5 1C 1C 9D 6D D0
00010:   19 98 C5 71 76 37 FB 17 11 E4 48 F0 0C 0D 60 B2
00020:   E4 88 89 4F B6 02 87 DB 77 5A 07 D9 2C 89 46 EA
00030:   BC 4F 87 23 DB F0 91 50 DD B4 06 C3 1D A9 7C A4
00040:   68 6F 22 7D 8F B2 9C BD 05 C8 C3 7D 22 FE 3B B7
00050:   C0 1B F9 7F 75 6E 12 2F 80 59 55 BD DE 2D 45 87
00060:   03 DE 34 74 AB 9B 65 8A 3B 54 1E F8 BD 2B F4 7D

The result ciphertext C = P (xor) MSB_{|P|}(G):
00000:   EC 5C CB DE 8C 18 D3 B8 72 56 68 D0 A7 37 F4 58
00010:   19 89 E7 42 32 62 9D 60 99 7D E2 4B C0 E3 9F B8
```

```
00020:   F5 AA BA 0B E3 64 F0 53 EE F0 BC 15 C2 76 4C EA
00030:   9E 7C C3 76 BD 87 19 C9 77 0F CA 2D E2 A3 7C B5
00040:   5B 2B 77 1B F8 3A 05 17 BE 04 2D 82 28 FE 2A 95
00050:   84 4E 9F 08 FD F7 B8 94 4C B7 AA B7 DE 3C 67 B4
00060:   56 B8 43 FC 32 31 DE 46 D5 AB 14 F8 AC 09 C7 39
```

```
GCM-ACPKM mode with AES-128
***************************
k = 128
n = 128
c = 32
N = 256

Initilal Key K:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Additional data A:
00000:   11 22 33

Plaintext:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

ICN:
00000:   00 00 00 00 00 00 00 00 00 00 00 00

Number of sections: 2

Section key K^1:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Section key K^2:
00000:   15 1A 9F B0 B6 AC C5 97 6A FB 50 31 D1 DE C8 41

Encrypted GCTR_1 | GCTR_2 | GCTR_3:
00000:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00010:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00020:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD

Ciphertext C:
00000:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00010:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00020:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD

GHASH input:
```

```
00000:   11 22 33 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00020:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00030:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD
00040:   00 00 00 00 00 00 00 18 00 00 00 00 00 00 01 80

GHASH output S:
00000:   E8 ED E9 94 9A DD 55 30 B0 F4 4E F5 00 FC 3E 3C

Authentication tag  T:
00000:   B0 0F 15 5A 60 A3 65 51 86 8B 53 A2 A4 1B 7B 66

The result C | T:
00000:   03 88 DA CE 60 B6 A3 92 F3 28 C2 B9 71 B2 FE 78
00010:   F7 95 AA AB 49 4B 59 23 F7 FD 89 FF 94 8B C1 E0
00020:   D6 B3 12 46 E9 CE 9F F1 3A B3 42 7E E8 91 96 AD
00030:   B0 0F 15 5A 60 A3 65 51 86 8B 53 A2 A4 1B 7B 66
```

A.2.2. Internal Re-keying Mechanisms with a Master Key

```
CTR-ACPKM-Master mode with AES-256
**********************************
k = 256
n = 128
c for CTR-ACPKM mode = 64
c for CTR-ACPKM-Master mode = 64
N = 256
T* = 512

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Initial vector ICN:
00000:   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12

Plaintext P:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44
```

```
K^1 | K^2 | K^3 | K^4:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00050:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

Input block CTR_1:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 00

Output block G_1:
00000:   8C A2 B6 82 A7 50 65 3F 8E BF 08 E7 9F 99 4D 5C

Input block CTR_2:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 01

Output block G_2:
00000:   F6 A6 A5 BA 58 14 1E ED 23 DC 31 68 D2 35 89 A1


Section_2

K^2:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00010:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3

Input block CTR_3:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 02

Output block G_3:
00000:   4A 07 5F 86 05 87 72 94 1D 8E 7D F8 32 F4 23 71

Input block CTR_4:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 03

Output block G_4:
00000:   23 35 66 AF 61 DD FE A7 B1 68 3F BA B0 52 4A D7
```

```
Section_3

K^3:
00000:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00010:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94

Input block CTR_5:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 04

Output block G_5:
00000:   A8 09 6D BC E8 BB 52 FC DE 6E 03 70 C1 66 95 E8

Input block CTR_6:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 05

Output block G_6:
00000:   C6 E3 6E 8E 5B 82 AA C4 A6 6C 14 8D B1 F6 9B EF


Section_4

K^4:
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Input block CTR_7:
00000:   12 34 56 78 90 AB CE F0 00 00 00 00 00 00 00 06

Output block G_7:
00000:   82 2B E9 07 96 37 44 95 75 36 3F A7 07 F8 40 22


The result G = G_1 | G_2 | G_3 | G_4 | G_5 | G_6 | G_7:
00000:   8C A2 B6 82 A7 50 65 3F 8E BF 08 E7 9F 99 4D 5C
00010:   F6 A6 A5 BA 58 14 1E ED 23 DC 31 68 D2 35 89 A1
00020:   4A 07 5F 86 05 87 72 94 1D 8E 7D F8 32 F4 23 71
00030:   23 35 66 AF 61 DD FE A7 B1 68 3F BA B0 52 4A D7
00040:   A8 09 6D BC E8 BB 52 FC DE 6E 03 70 C1 66 95 E8
00050:   C6 E3 6E 8E 5B 82 AA C4 A6 6C 14 8D B1 F6 9B EF
00060:   82 2B E9 07 96 37 44 95 75 36 3F A7 07 F8 40 22

The result ciphertext C = P (xor) MSB_{|P|}(G):
00000:   9D 80 85 C6 F2 36 12 3F 71 51 D5 2B 24 33 D4 D4
00010:   F6 B7 87 89 1C 41 78 9A AB 45 9B D3 1E DB 76 AB
00020:   5B 25 6C C2 50 E1 05 1C 84 24 C6 34 DC 0B 29 71
00030:   01 06 22 FA 07 AA 76 3E 1B D3 F3 54 4F 58 4A C6
00040:   9B 4D 38 DA 9F 33 CB 56 65 A2 ED 8F CB 66 84 CA
00050:   82 B6 08 F9 D3 1B 00 7F 6A 82 EB 87 B1 E7 B9 DC
```

```
00060:   D7 4D 9E 8F 0F 9D FF 59 9B C9 35 A7 16 DA 73 66



GCM-ACPKM-Master mode with AES-256
**********************************
k = 192
n = 128
c for the CTR-ACPKM mode = 64
c for the GCM-ACPKM-Master mode = 32
T* = 384
N = 256

Initila Key K:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   00 00 00 00 00 00 00 00

Additional data A:
00000:   11 22 33

Plaintext:
00000:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00030:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

ICN:
00000:   00 00 00 00 00 00 00 00 00 00 00 00

Number of sections: 3

K^1 | K^2 | K^3:
00000:   93 BA AF FB 35 FB E7 39 C1 7C 6A C2 2E EC F1 8F
00010:   7B 89 F0 BF 8B 18 07 05 96 48 68 9F 36 A7 65 CC
00020:   CD 5D AC E2 0D 47 D9 18 D7 86 D0 41 A8 3B AB 99
00030:   F5 F8 B1 06 D2 71 78 B1 B0 08 C9 99 0B 72 E2 87
00040:   5A 2D 3C BE F1 6E 67 3C

Encrypted GCTR_1 | ... | GCTR_5
00000:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00010:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00020:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00030:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00040:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08

Ciphertext C:
```

```
00000:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00010:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00020:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00030:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00040:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08

GHASH input:
00000:   11 22 33 00 00 00 00 00 00 00 00 00 00 00 00 00
00010:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00020:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00030:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00040:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00050:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08
00060:   00 00 00 00 00 00 00 18 00 00 00 00 00 00 02 80

GHASH output S:
00000:   6E A3 4B D5 6A C5 40 B7 3E 55 D5 86 D1 CC 09 7D

Authentication tag  T:
00050:   CC 3A BA 11 8C E7 85 FD 77 78 94 D4 B5 20 69 F8

The result C | T:
00000:   43 FA 71 81 64 B1 E3 D7 1E 7B 65 39 A7 02 1D 52
00010:   69 9B 9E 1B 43 24 B7 52 95 74 E7 90 F2 BE 60 E8
00020:   11 62 C9 90 2A 2B 77 7F D9 6A D6 1A 99 E0 C6 DE
00030:   4B 91 D4 29 E3 1A 8C 11 AF F0 BC 47 F6 80 AF 14
00040:   40 1C C1 18 14 63 8E 76 24 83 37 75 16 34 70 08
00050:   CC 3A BA 11 8C E7 85 FD 77 78 94 D4 B5 20 69 F8
```

```
CBC-ACPKM-Master mode with AES-256
**********************************
k = 256
n = 128
c for the CTR-ACPKM mode = 64
N = 256
T* = 512

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Initial vector IV:
00000:   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12

Plaintext P:
```

```
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

K^1 | K^2 | K^3 | K^4:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00050:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

Plaintext block P_1:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Input block P_1 (xor) C_0:
00000:   03 16 65 3C C5 CD B9 F0 5E 5C 1E 18 5E 5A 98 9A

Output block C_1:
00000:   59 CB 5B CA C2 69 2C 60 0D 46 03 A0 C7 40 C9 7C

Plaintext block P_2:
00000:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

Input block P_2 (xor) C_1:
00000:   59 DA 79 F9 86 3C 4A 17 85 DF A9 1B 0B AE 36 76

Output block C_2:
00000:   80 B6 02 74 54 8B F7 C9 78 1F A1 05 8B F6 8B 42

Section_2

K^2:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00010:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
```

```
Plaintext block P_3:
00000:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

Input block P_3 (xor) C_2:
00000:   91 94 31 30 01 ED 80 41 E1 B5 1A C9 65 09 81 42

Output block C_3:
00000:   8C 24 FB CF 68 15 B1 AF 65 FE 47 75 95 B4 97 59

Plaintext block P_4:
00000:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11

Input block P_4 (xor) C_3:
00000:   AE 17 BF 9A 0E 62 39 36 CF 45 8B 9B 6A BE 97 48

Output block C_4:
00000:   19 65 A5 00 58 0D 50 23 72 1B E9 90 E1 83 30 E9

Section_3

K^3:
00000:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00010:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94

Plaintext block P_5:
00000:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

Input block P_5 (xor) C_4:
00000:   2A 21 F0 66 2F 85 C9 89 C9 D7 07 6F EB 83 21 CB

Output block C_5:
00000:   56 D8 34 F4 6F 0F 4D E6 20 53 A9 5C B5 F6 3C 14

Plaintext block P_6:
00000:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33

Input block P_6 (xor) C_5:
00000:   12 8D 52 83 E7 96 E7 5D EC BD 56 56 B5 E7 1E 27

Output block C_6:
00000:   66 68 2B 8B DD 6E B2 7E DE C7 51 D6 2F 45 A5 45

Section_4

K^4:
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12
```

```
Plaintext block P_7:
00000:   55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33 44

Input block P_7 (xor) C_6:
00000:   33 0E 5C 03 44 C4 09 B2 30 38 5B D6 3E 67 96 01

Output block C_7:
00000:   7F 4D 87 F9 CA E9 56 09 79 C4 FA FE 34 0B 45 34

Cipher text C:
00000:   59 CB 5B CA C2 69 2C 60 0D 46 03 A0 C7 40 C9 7C
00010:   80 B6 02 74 54 8B F7 C9 78 1F A1 05 8B F6 8B 42
00020:   8C 24 FB CF 68 15 B1 AF 65 FE 47 75 95 B4 97 59
00030:   19 65 A5 00 58 0D 50 23 72 1B E9 90 E1 83 30 E9
00040:   56 D8 34 F4 6F 0F 4D E6 20 53 A9 5C B5 F6 3C 14
00050:   66 68 2B 8B DD 6E B2 7E DE C7 51 D6 2F 45 A5 45
00060:   7F 4D 87 F9 CA E9 56 09 79 C4 FA FE 34 0B 45 34
```

```
CFB-ACPKM-Master mode with AES-256
**********************************
k = 256
n = 128
c for the CTR-ACPKM mode = 64
N = 256
T* = 512

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Initial vector IV:
00000:   12 34 56 78 90 AB CE F0 A1 B2 C3 D4 E5 F0 01 12

Plaintext P:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22
00050:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33
00060:   55 66 77 88 99 AA BB CC

K^1 | K^2 | K^3 | K^4
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
```

```
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00050:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

Plaintext block P_1:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Encrypted block E_{K^1}(C_0):
00000:   1C 39 9D 59 F8 5D 91 91 A9 D2 12 9F 63 15 90 03

Output block C_1 = E_{K^1}(C_0) (xor) P_1:
00000:   0D 1B AE 1D AD 3B E6 91 56 3C CF 53 D8 BF 09 8B

Plaintext block P_2:
00000:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

Encrypted block E_{K^1}(C_1):
00000:   6B A2 C5 42 52 69 C6 0B 15 14 06 87 90 46 F6 2E

Output block C_2 = E_{K^1}(C_1) (xor) P_2:
00000:   6B B3 E7 71 16 3C A0 7C 9D 8D AC 3C 5C A8 09 24

Section_2

K^2:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00010:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3

Plaintext block P_3:
00000:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

Encrypted block E_{K^2}(C_2):
00000:   95 45 5F DB C3 9E 0A 13 9F CB 10 F5 BD 79 A3 88

Output block C_3 = E_{K^2}(C_2) (xor) P_3:
00000:   84 67 6C 9F 96 F8 7D 9B 06 61 AB 39 53 86 A9 88

Plaintext block P_4:
00000:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
```

```
Encrypted block E_{K^2}(C_3):
00000:   E0 AA 32 5D 80 A4 47 95 BA 42 BF 63 F8 4A C8 B2

Output block C_4 = E_{K^2}(C_3) (xor) P_4:
00000:   C2 99 76 08 E6 D3 CF 0C 10 F9 73 8D 07 40 C8 A3

Section_3

K^3:
00000:   E8 76 2B 30 8B 08 EB CE 3E 93 9A C2 C0 3E 76 D4
00010:   60 9A AB D9 15 33 13 D3 CF D3 94 E7 75 DF 3A 94

Plaintext block P_5:
00000:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

Encrypted block E_{K^3}(C_4):
00000:   FE 42 8C 70 C2 51 CE 13 36 C1 BF 44 F8 49 66 89

Output block C_5 = E_{K^3}(C_4) (xor) P_5:
00000:   CD 06 D9 16 B5 D9 57 B9 8D 0D 51 BB F2 49 77 AB

Plaintext block P_6:
00000:   44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22 33

Encrypted block E_{K^3}(C_5):
00000:   01 24 80 87 86 18 A5 43 11 0A CC B5 0A E5 02 A3

Output block C_6 = E_{K^3}(C_5) (xor) P_6:
00000:   45 71 E6 F0 0E 81 0F F8 DD E4 33 BF 0A F4 20 90

Section_4

K^4:
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

Plaintext block P_7:
00000:   55 66 77 88 99 AA BB CC

Encrypted block MSB_{|P_7|}(E_{K^4}(C_6)):
00000:   97 5C 96 37 55 1E 8C 7F

Output block C_7 = MSB_{|P_7|}(E_{K^4}(C_6)) (xor) P_7
00000:   C2 3A E1 BF CC B4 37 B3

Cipher text C:
00000:   0D 1B AE 1D AD 3B E6 91 56 3C CF 53 D8 BF 09 8B
00010:   6B B3 E7 71 16 3C A0 7C 9D 8D AC 3C 5C A8 09 24
```

```
00020:   84 67 6C 9F 96 F8 7D 9B 06 61 AB 39 53 86 A9 88
00030:   C2 99 76 08 E6 D3 CF 0C 10 F9 73 8D 07 40 C8 A3
00040:   CD 06 D9 16 B5 D9 57 B9 8D 0D 51 BB F2 49 77 AB
00050:   45 71 E6 F0 0E 81 0F F8 DD E4 33 BF 0A F4 20 90
00060:   C2 3A E1 BF CC B4 37 B3
```

```
OMAC-ACPKM-Master mode with AES-256
***********************************
k = 256
n = 128
c for the CTR-ACPKM mode = 64
N = 256
T* = 768

Initial key K:
00000:   88 99 AA BB CC DD EE FF 00 11 22 33 44 55 66 77
00010:   FE DC BA 98 76 54 32 10 01 23 45 67 89 AB CD EF

Plaintext M:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88
00010:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A
00020:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00
00030:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11
00040:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

K^1 | K^1_1 | K^2 | K^2_1 | K^3 | K^3_1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60
00020:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
00030:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00040:   9D CC 66 42 0D FF 45 5B 21 F3 93 F0 D4 D6 6E 67
00050:   BB 1B 06 0B 87 66 6D 08 7A 9D A7 49 55 C3 5B 48
00060:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00070:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12
00080:   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

Section_1

K^1:
00000:   9F 10 BB F1 3A 79 FB BD 4A 4C A8 64 C4 90 74 64
00010:   39 FE 50 6D 4B 86 9B 21 03 A3 B6 A4 79 28 3C 60

K^1_1:
00000:   77 91 17 50 E0 D1 77 E5 9A 13 78 2B F1 89 08 D0
```

```
Plaintext block M_1:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Input block M_1 (xor) C_0:
00000:   11 22 33 44 55 66 77 00 FF EE DD CC BB AA 99 88

Output block C_1:
00000:   0B A5 89 BF 55 C1 15 42 53 08 89 76 A0 FE 24 3E

Plaintext block M_2:
00000:   00 11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A

Input block M_2 (xor) C_1:
00000:   0B B4 AB 8C 11 94 73 35 DB 91 23 CD 6C 10 DB 34

Output block C_2:
00000:   1C 53 DD A3 6D DC E1 17 ED 1F 14 09 D8 6A F3 2C

Section_2

K^2:
00000:   AB 6B 59 EE 92 49 05 B3 AB C7 A4 E3 69 65 76 C3
00010:   9D CC 66 42 0D FF 45 5B 21 F3 93 F0 D4 D6 6E 67

K^2_1:
00000:   BB 1B 06 0B 87 66 6D 08 7A 9D A7 49 55 C3 5B 48

Plaintext block M_3:
00000:   11 22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00

Input block M_3 (xor) C_2:
00000:   0D 71 EE E7 38 BA 96 9F 74 B5 AF C5 36 95 F9 2C

Output block C_3:
00000:   4E D4 BC A6 CE 6D 6D 16 F8 63 85 13 E0 48 59 75

Plaintext block M_4:
00000:   22 33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11

Input block M_4 (xor) C_3:
00000:   6C E7 F8 F3 A8 1A E5 8F 52 D8 49 FD 1F 42 59 64

Output block C_4:
00000:   B6 83 E3 96 FD 30 CD 46 79 C1 8B 24 03 82 1D 81

Section_3

K^3:
```

```
00000:   F2 EE 91 45 6B DC 3D E4 91 2C 87 C3 29 CF 31 A9
00010:   2F 20 2E 5A C4 9A 2A 65 31 33 D6 74 8C 4F F9 12

K^3_1:
00000:   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

MSB1(K1) == 0 -> K2 = K1 << 1

K1:
00000:   78 21 C7 C7 6C BD 79 63 56 AC F8 8E 69 6A 00 07

K2:
00000:   F0 43 8F 8E D9 7A F2 C6 AD 59 F1 1C D2 D4 00 0E

Plaintext M_5:
00000:   33 44 55 66 77 88 99 AA BB CC EE FF 0A 00 11 22

Using K1, padding is not required

Input block M_5 (xor) C_4:
00000:   FD E6 71 37 E6 05 2D 8F 94 A1 9D 55 60 E8 0C A4

Output block C_5:
00000:   B3 AD B8 92 18 32 05 4C 09 21 E7 B8 08 CF A0 B8

Message authentication code T:
00000:   B3 AD B8 92 18 32 05 4C 09 21 E7 B8 08 CF A0 B8
```

Appendix B. Contributors

- Russ Housley
  Vigil Security, LLC
  housley@vigilsec.com

- Evgeny Alekseev
  CryptoPro
  alekseev@cryptopro.ru

- Ekaterina Smyshlyaeva
  CryptoPro
  ess@cryptopro.ru

- Shay Gueron
  University of Haifa, Israel
  Intel Corporation, Israel Development Center, Israel
  shay.gueron@gmail.com

- Daniel Fox Franke
  Akamai Technologies
  dfoxfranke@gmail.com

- Lilia Ahmetzyanova
  CryptoPro
  lah@cryptopro.ru

Appendix C. Acknowledgments

We thank Mihir Bellare, Scott Fluhrer, Dorothy Cooley, Yoav Nir, Jim Schaad, Paul Hoffman, Dmitry Belyavsky, Yaron Sheffer, Alexey Melnikov and Spencer Dawkins for their useful comments.

Author's Address

Stanislav Smyshlyaev (editor)
CryptoPro
18, Suschevskiy val
Moscow 127018
Russian Federation

Phone: +7 (495) 995-48-20
Email: svs@cryptopro.ru

Crypto Forum Research Group
Internet-Draft
Intended status: Informational
Expires: July 11, 2019

D. McGrew
M. Curcio
S. Fluhrer
Cisco Systems
January 7, 2019

Hash-Based Signatures
draft-mcgrew-hash-sigs-15

Abstract

This note describes a digital signature system based on cryptographic hash functions, following the seminal work in this area of Lamport, Diffie, Winternitz, and Merkle, as adapted by Leighton and Micali in 1995. It specifies a one-time signature scheme and a general signature scheme. These systems provide asymmetric authentication without using large integer mathematics and can achieve a high security level. They are suitable for compact implementations, are relatively simple to implement, and naturally resist side-channel attacks. Unlike most other signature systems, hash-based signatures would still be secure even if it proves feasible for an attacker to build a quantum computer.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

Copyright Notice

Table of Contents

1. Introduction

One-time signature systems, and general purpose signature systems built out of one-time signature systems, have been known since 1979 [Merkle79], were well studied in the 1990s [USPTO5432852], and have benefited from renewed attention in the last decade. The characteristics of these signature systems are small private and public keys and fast signature generation and verification, but large signatures and moderately slow key generation (in comparison with RSA and ECDSA). Private keys can be made very small by appropriate key generation, for example, as described in Appendix A. In recent years there has been interest in these systems because of their post-quantum security and their suitability for compact verifier implementations.

This note describes the Leighton and Micali adaptation [USPTO5432852] of the original Lamport-Diffie-Winternitz-Merkle one-time signature system [Merkle79] [C:Merkle87][C:Merkle89a][C:Merkle89b] and general signature system [Merkle79] with enough specificity to ensure interoperability between implementations.

A signature system provides asymmetric message authentication. The key generation algorithm produces a public/private key pair. A message is signed by a private key, producing a signature, and a message/signature pair can be verified by a public key. A One-Time

Signature (OTS) system can be used to sign one message securely, but will become insecure if more than one is signed with the same public/private key pair. An N-time signature system can be used to sign N or fewer messages securely. A Merkle tree signature scheme is an N-time signature system that uses an OTS system as a component.

In the Merkle scheme, a binary tree of height h is used to hold 2^h OTS key pairs. Each interior node of the tree holds a value which is the hash of the values of its two children nodes. The public key of the tree is the value of the root node (a recursive hash of the OTS public keys), while the private key of the tree is the collection of all the OTS private keys, together with the index of the next OTS private key to sign the next message with.

In this note we describe the Leighton-Micali Signature (LMS) system, which is a variant of the Merkle scheme, and a Hierarchical Signature System (HSS) built on top of it that can efficiently scale to larger numbers of signatures. In order to support signing a large number of messages on resource constrained systems, the Merkle tree can be subdivided into a number of smaller trees. Only the bottom-most tree is used to sign messages, while trees above that are used to sign the public keys of their children. For example, in the simplest case with 2 levels with both levels consisting of height h trees, the root tree is used to sign 2^h trees with 2^h OTS key pairs, and each second level tree has 2^h OTS key pairs, for a total of 2^(2h) bottom level key pairs, and so can sign 2^(2h) messages. The advantage of this scheme is that only the active trees need to be instantiated, which saves both time (for key generation) and space (for key storage). On the other hand, using a multilevel signature scheme inceases the size of the signature, as well as the signature verification time.

This note is structured as follows. Notes on postquantum cryptography are discussed in Section 1.1. Intellectual Property issues are discussed in Section 1.2. The notation used within this note is defined in Section 3, and the public formats are described in Section 3.3. The LM-OTS signature system is described in Section 4, and the LMS and HSS N-time signature systems are described in Section 5 and Section 6, respectively. Sufficient detail is provided to ensure interoperability. The rationale for the design decisions is given in Section 7. The IANA registry for these signature systems is described in Section 8. Security considerations are presented in Section 9. Comparison with another hash based signature algorithm (XMSS) is in Section 10.

This document represents the rough consensus of the CFRG.

1.1. CFRG Note on Post-Quantum Cryptography

All post-quantum algorithms documented by the Crypto Forum Research Group (CFRG) are today considered ready for experimentation and further engineering development (e.g., to establish the impact of performance and sizes on IETF protocols). However, at the time of writing, we do not have significant deployment experience with such algorithms.

Many of these algorithms come with specific restrictions, e.g., change of classical interface or less cryptanalysis of proposed parameters than established schemes. CFRG has consensus that all documents describing post-quantum technologies include the above paragraph and a clear additional warning about any specific restrictions, especially as those might affect use or deployment of the specific scheme. That guidance may be changed over time via document updates.

Additionally, for LMS:

CFRG consensus is that we are confident in the cryptographic security of the signature schemes described in this document against quantum computers, given the current state of the research community's knowledge about quantum algorithms. Indeed, we are confident that the security of a significant part of the Internet could be made dependent on the signature schemes defined in this document, if developers take care of the following.

In contrast to traditional signature schemes, the signature schemes described in this document are stateful, meaning the secret key changes over time. If a secret key state is used twice, no cryptographic security guarantees remain. In consequence, it becomes feasible to forge a signature on a new message. This is a new property that most developers will not be familiar with and requires careful handling of secret keys. Developers should not use the schemes described here except in systems that prevent the reuse of secret key states.

Note that the fact that the schemes described in this document are stateful also implies that classical APIs for digital signatures cannot be used without modification. The API MUST be able to handle a secret key state; in particular, this means that the API MUST allow to return an updated secret key state.

1.2. Intellectual Property

This draft is based on U.S. patent 5,432,852, which was issued over twenty years ago and is thus expired.

1.2.1. Disclaimer

This document is not intended as legal advice. Readers are advised to consult with their own legal advisers if they would like a legal interpretation of their rights.

The IETF policies and processes regarding intellectual property and patents are outlined in [RFC3979] and [RFC4879] and at https://datatracker.ietf.org/ipr/about.

1.3. Conventions Used In This Document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Interface

The LMS signing algorithm is stateful; it modifies and updates the private key as a side effect of generating a signature. Once a particular value of the private key is used to sign one message, it MUST NOT be used to sign another.

The key generation algorithm takes as input an indication of the parameters for the signature system. If it is successful, it returns both a private key and a public key. Otherwise, it returns an indication of failure.

The signing algorithm takes as input the message to be signed and the current value of the private key. If successful, it returns a signature and the next value of the private key, if there is such a value. After the private key of an N-time signature system has signed N messages, the signing algorithm returns the signature and an indication that there is no next value of the private key that can be used for signing. If unsuccessful, it returns an indication of failure.

The verification algorithm takes as input the public key, a message, and a signature, and returns an indication of whether or not the signature and message pair is valid.

A message/signature pair is valid if the signature was returned by the signing algorithm upon input of the message and the private key

corresponding to the public key; otherwise, the signature and message pair is not valid with probability very close to one.

3. Notation

3.1. Data Types

Bytes and byte strings are the fundamental data types. A single byte is denoted as a pair of hexadecimal digits with a leading "0x". A byte string is an ordered sequence of zero or more bytes and is denoted as an ordered sequence of hexadecimal characters with a leading "0x". For example, 0xe534f0 is a byte string with a length of three. An array of byte strings is an ordered set, indexed starting at zero, in which all strings have the same length.

Unsigned integers are converted into byte strings by representing them in network byte order. To make the number of bytes in the representation explicit, we define the functions u8str(X), u16str(X), and u32str(X), which take a non-negative integer X as input and return one, two, and four byte strings, respectively. We also make use of the function strTou32(S), which takes a four-byte string S as input and returns a non-negative integer; the identity u32str(strTou32(S)) = S holds for any four-byte string S.

3.1.1. Operators

When a and b are real numbers, mathematical operators are defined as follows:

^ : a ^ b denotes the result of a raised to the power of b

* : a * b denotes the product of a multiplied by b

/ : a / b denotes the quotient of a divided by b

% : a % b denotes the remainder of the integer division of a by b (with a and b being restricted to integers in this case)

+ : a + b denotes the sum of a and b

- : a - b denotes the difference of a and b

AND : a AND b denotes the bitwise AND of the two nonnegative integers a and b (represented in binary notation)

The standard order of operations is used when evaluating arithmetic expressions.

When B is a byte and i is an integer, then B >> i denotes the logical right-shift operation by i bit positions. Similarly, B << i denotes the logical left-shift operation.

If S and T are byte strings, then S || T denotes the concatenation of S and T. If S and T are equal length byte strings, then S AND T denotes the bitwise logical and operation.

The i-th element in an array A is denoted as A[i].

3.1.2. Functions

If r is a non-negative real number, then we define the following functions:

ceil(r) : returns the smallest integer greater than or equal to r

floor(r) : returns the largest integer less than or equal to r

lg(r) : returns the base-2 logarithm of r

3.1.3. Strings of w-bit elements

If S is a byte string, then byte(S, i) denotes its i-th byte, where the index starts at 0 at the left. Hence, byte(S, 0) is the leftmost byte of S, byte(S, 1) is the second byte at the left and (assuming S is n bytes long) byte(S, n-1) is the rightmost byte of S. In addition, bytes(S, i, j) denotes the range of bytes from the i-th to the j-th byte, inclusive. For example, if S = 0x02040608, then byte(S, 0) is 0x02 and bytes(S, 1, 2) is 0x0406.

A byte string can be considered to be a string of w-bit unsigned integers; the correspondence is defined by the function coef(S, i, w) as follows:

If S is a string, i is a positive integer, and w is a member of the set { 1, 2, 4, 8 }, then coef(S, i, w) is the i-th, w-bit value, if S is interpreted as a sequence of w-bit values. That is,

```
coef(S, i, w) = (2^w - 1) AND
                ( byte(S, floor(i * w / 8)) >>
                  (8 - (w * (i % (8 / w)) + w)) )
```

For example, if S is the string 0x1234, then coef(S, 7, 1) is 0 and coef(S, 0, 4) is 1.

```
                 S (represented as bits)
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      | 0| 0| 0| 1| 0| 0| 1| 0| 0| 0| 1| 1| 0| 1| 0| 0|
      +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                            ^
                            |
                      coef(S, 7, 1)


              S (represented as four-bit values)
      +-----------+-----------+-----------+-----------+
      |     1     |     2     |     3     |     4     |
      +-----------+-----------+-----------+-----------+
            ^
            |
      coef(S, 0, 4)
```

The return value of coef is an unsigned integer. If i is larger than the number of w-bit values in S, then coef(S, i, w) is undefined, and an attempt to compute that value MUST raise an error.

3.2. Typecodes

A typecode is an unsigned integer that is associated with a particular data format. The format of the LM-OTS, LMS, and HSS signatures and public keys all begin with a typecode that indicates the precise details used in that format. These typecodes are represented as four-byte unsigned integers in network byte order; equivalently, they are XDR enumerations (see Section 3.3).

3.3. Notation and Formats

The signature and public key formats are formally defined using the External Data Representation (XDR) [RFC4506] in order to provide an unambiguous, machine readable definition. For clarity, we also include a private key format as well, though consistency is not needed for interoperability and an implementation MAY use any private key format. Though XDR is used, these formats are simple and easy to parse without any special tools. An illustration of the layout of data in these objects is provided below. The definitions are as follows:

```
/* one-time signatures */
```

```
enum lmots_algorithm_type {
  lmots_reserved       = 0,
  lmots_sha256_n32_w1  = 1,
  lmots_sha256_n32_w2  = 2,
  lmots_sha256_n32_w4  = 3,
  lmots_sha256_n32_w8  = 4
};

typedef opaque bytestring32[32];

struct lmots_signature_n32_p265 {
  bytestring32 C;
  bytestring32 y[265];
};

struct lmots_signature_n32_p133 {
  bytestring32 C;
  bytestring32 y[133];
};

struct lmots_signature_n32_p67 {
  bytestring32 C;
  bytestring32 y[67];
};

struct lmots_signature_n32_p34 {
  bytestring32 C;
  bytestring32 y[34];
};

union lmots_signature switch (lmots_algorithm_type type) {
 case lmots_sha256_n32_w1:
   lmots_signature_n32_p265 sig_n32_p265;
 case lmots_sha256_n32_w2:
   lmots_signature_n32_p133 sig_n32_p133;
 case lmots_sha256_n32_w4:
   lmots_signature_n32_p67  sig_n32_p67;
 case lmots_sha256_n32_w8:
   lmots_signature_n32_p34  sig_n32_p34;
 default:
   void;   /* error condition */
};


/* hash based signatures (hbs) */

enum lms_algorithm_type {
  lms_reserved       = 0,
```

```
    lms_sha256_n32_h5  = 5,
    lms_sha256_n32_h10 = 6,
    lms_sha256_n32_h15 = 7,
    lms_sha256_n32_h20 = 8,
    lms_sha256_n32_h25 = 9,
  };

  /* leighton-micali signatures (lms) */

  union lms_path switch (lms_algorithm_type type) {
   case lms_sha256_n32_h5:
     bytestring32 path_n32_h5[5];
   case lms_sha256_n32_h10:
     bytestring32 path_n32_h10[10];
   case lms_sha256_n32_h15:
     bytestring32 path_n32_h15[15];
   case lms_sha256_n32_h20:
     bytestring32 path_n32_h20[20];
   case lms_sha256_n32_h25:
     bytestring32 path_n32_h25[25];
   default:
     void;     /* error condition */
  };

  struct lms_signature {
    unsigned int q;
    lmots_signature lmots_sig;
    lms_path nodes;
  };

  struct lms_key_n32 {
    lmots_algorithm_type ots_alg_type;
    opaque I[16];
    opaque K[32];
  };

  union lms_public_key switch (lms_algorithm_type type) {
   case lms_sha256_n32_h5:
   case lms_sha256_n32_h10:
   case lms_sha256_n32_h15:
   case lms_sha256_n32_h20:
   case lms_sha256_n32_h25:
        lms_key_n32 z_n32;
   default:
     void;     /* error condition */
  };

  /* hierarchical signature system (hss)  */
```

```
struct hss_public_key {
  unsigned int L;
  lms_public_key pub;
};

struct signed_public_key {
  lms_signature sig;
  lms_public_key pub;
}

struct hss_signature {
  signed_public_key signed_keys<7>;
  lms_signature sig_of_message;
};
```

4. LM-OTS One-Time Signatures

This section defines LM-OTS signatures. The signature is used to validate the authenticity of a message by associating a secret private key with a shared public key. These are one-time signatures; each private key MUST be used at most one time to sign any given message.

As part of the signing process, a digest of the original message is computed using the cryptographic hash function H (see Section 4.1), and the resulting digest is signed.

In order to facilitate its use in an N-time signature system, the LM-OTS key generation, signing, and verification algorithms all take as input parameters I and q. The parameter I is a 16 byte string, which indicates which Merkle tree this LM-OTS is used with. The parameter q is a 32 bit integer which indicates the leaf of the Merkle tree where the OTS public key appears. These parameters are used as part of the security string, as listed in Section 7.1. When the LM-OTS signature system is used outside of an N-time signature system, the value I MAY be used to differentiate this one time signatures from others; however the value q MUST be set to the all-zero value.

4.1. Parameters

The signature system uses the parameters n and w, which are both positive integers. The algorithm description also makes use of the internal parameters p and ls, which are dependent on n and w. These parameters are summarized as follows:

n : the number of bytes of the output of the hash function

w : the width (in bits) of the Winternitz coefficients; that is, the number of bits from the hash or checksum that is used with a single Winternitz chain. It is a member of the set { 1, 2, 4, 8 }

p : the number of n-byte string elements that make up the LM-OTS signature. This is a function of n and w; the values for the defined parameter sets are lited in Table 1; it can also be computed by the algorithm given in Appendix B.

ls : the number of left-shift bits used in the checksum function Cksm (defined in Section 4.4)

H : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an n-byte string

For more background on the cryptographic security requirements on H, see the Section 9.

The value of n is determined by the hash function selected for use as part of the LM-OTS algorithm; the choice of this value has a strong effect on the security of the system. The parameter w determines the length of the Winternitz chains computed as a part of the OTS signature (which involve 2^w-1 invocations of the hash function); it has little effect on security. Increasing w will shorten the signature, but at a cost of a larger computation to generate and verify a signature. The values of p and ls are dependent on the choices of the parameters n and w, as described in Appendix B. A table illustrating various combinations of n, w, p and ls, along with the resulting signature length, is provided in Table 1.

The value of w describes a space/time trade-off; increasing the value of w will cause the signature to shrink (by decreasing the value of p) while increasing the amount of time needed to perform operations with it (generate the public key, generate and verify the signature); in general, the LM-OTS signature is 4+n*(p+1) bytes long, and public key generation will take p*(2^w-1)+1 hash computations (and signature generation and verification will take approximately half that on average).

Parameter Set Name	H	n	w	p	ls	sig_len
LMOTS_SHA256_N32_W1	SHA256	32	1	265	7	8516
LMOTS_SHA256_N32_W2	SHA256	32	2	133	6	4292
LMOTS_SHA256_N32_W4	SHA256	32	4	67	4	2180
LMOTS_SHA256_N32_W8	SHA256	32	8	34	0	1124

Table 1

Here SHA256 denotes the SHA-256 hash fucntion defined in NIST standard [FIPS180].

4.2. Private Key

The format of the LM-OTS private key is an internal matter to the implementation, and this document does not attempt to define it. One possibility is that the private key may consist of a typecode indicating the particular LM-OTS algorithm, an array x[] containing p n-byte strings, and the 16-byte string I and the 4 byte string q. This private key MUST be used to sign (at most) one message. The following algorithm shows pseudocode for generating a private key.

Algorithm 0: Generating a Private Key

1. retrieve the values of q and I (the 16-byte identifier of the LMS public/private keypair) from the LMS tree that this LM-OTS private key will be used with

2. set type to the typecode of the algorithm

3. set n and p according to the typecode and Table 1

4. compute the array x as follows:
```
for ( i = 0; i < p; i = i + 1 ) {
  set x[i] to a uniformly random n-byte string
}
```

5. return u32str(type) || I || u32str(q) || x[0] || x[1] || ... || x[p-1]

An implementation MAY use a pseudorandom method to compute x[i], as suggested in [Merkle79], page 46. The details of the pseudorandom method do not affect interoperability, but the cryptographic strength

MUST match that of the LM-OTS algorithm. Appendix A provides an example of a pseudorandom method for computing the LM-OTS private key.

4.3. Public Key

The LM-OTS public key is generated from the private key by iteratively applying the function H to each individual element of x, for 2^w - 1 iterations, then hashing all of the resulting values.

The public key is generated from the private key using the following algorithm, or any equivalent process.

Algorithm 1: Generating a One Time Signature Public Key From a Private Key

1. set type to the typecode of the algorithm

2. set the integers n, p, and w according to the typecode and Table 1

3. determine x, I and q from the private key

4. compute the string K as follows:

```
for ( i = 0; i < p; i = i + 1 ) {
  tmp = x[i]
  for ( j = 0; j < 2^w - 1; j = j + 1 ) {
     tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
  }
  y[i] = tmp
}
K = H(I || u32str(q) || u16str(D_PBLC) || y[0] || ... || y[p-1])
```

5. return u32str(type) || I || u32str(q) || K

where D_PBLC is the fixed two byte value 0x8080, which is used to distinguish the last hash from every other hash in this system.

The public key is the value returned by Algorithm 1.

4.4. Checksum

A checksum is used to ensure that any forgery attempt that manipulates the elements of an existing signature will be detected. This checksum is needed because an attacker can freely advance any of the Winternitz chains. That is, if this checksum were not present, then an attacker who could find a hash that has every digit larger than the valid hash could replace it (and adjust the Winternitz chains). The security property that it provides is detailed in

Section 9. The checksum function Cksm is defined as follows, where S denotes the n-byte string that is input to that function, and the value sum is a 16-bit unsigned integer:

Algorithm 2: Checksum Calculation

```
sum = 0
for ( i = 0; i < (n*8/w); i = i + 1 ) {
  sum = sum + (2^w - 1) - coef(S, i, w)
}
return (sum << ls)
```

ls is the parameter that shifts the significant bits of the checksum into the positions that will actually be used by the coef function when encoding the digits of the checksum. The actual ls parameter is a function of the n and w parameters; the values for the currently defined parameter sets is shown in table 1. It is calculated by the algorithm given in Appendix B.

Because of the left-shift operation, the rightmost bits of the result of Cksm will often be zeros. Due to the value of p, these bits will not be used during signature generation or verification.

4.5. Signature Generation

The LM-OTS signature of a message is generated by first prepending the LMS key identifier I, the LMS leaf identifier q, the value D_MESG (0x8181) and the randomizer C to the message, then computing the hash, and then concatenating the checksum of the hash to the hash itself, then considering the resulting value as a sequence of w-bit values, and using each of the w-bit values to determine the number of times to apply the function H to the corresponding element of the private key. The outputs of the function H are concatenated together and returned as the signature. The pseudocode for this procedure is shown below.

Algorithm 3: Generating a One Time Signature From a Private Key and a Message

1. set type to the typecode of the algorithm

2. set n, p, and w according to the typecode and Table 1

3. determine x, I and q from the private key

4. set C to a uniformly random n-byte string

5. compute the array y as follows:

```
Q = H(I || u32str(q) || u16str(D_MESG) || C || message)
for ( i = 0; i < p; i = i + 1 ) {
  a = coef(Q || Cksm(Q), i, w)
  tmp = x[i]
  for ( j = 0; j < a; j = j + 1 ) {
     tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
  }
  y[i] = tmp
}
```

6. return u32str(type) || C || y[0] || ... || y[p-1]

Note that this algorithm results in a signature whose elements are intermediate values of the elements computed by the public key algorithm in Section 4.3.

The signature is the string returned by Algorithm 3. Section 3.3 specifies the typecode and more formally defines the encoding and decoding of the string.

4.6. Signature Verification

In order to verify a message with its signature (an array of n-byte strings, denoted as y), the receiver must "complete" the chain of iterations of H using the w-bit coefficients of the string resulting from the concatenation of the message hash and its checksum. This computation should result in a value that matches the provided public key.

Algorithm 4a: Verifying a Signature and Message Using a Public Key

1. if the public key is not at least four bytes long,
   return INVALID

2. parse pubtype, I, q, and K from the public key as follows:
   a. pubtype = strTou32(first 4 bytes of public key)

   b. set n according to the pubkey and Table 1; if the public key is not exactly 24 + n bytes long, return INVALID

   c. I = next 16 bytes of public key

   d. q = strTou32(next 4 bytes of public key)

   e. K = next n bytes of public key

3. compute the public key candidate Kc from the signature, message, pubtype and the identifiers I and q obtained from the public key, using Algorithm 4b. If Algorithm 4b returns INVALID, then return INVALID.

4. if Kc is equal to K, return VALID; otherwise, return INVALID

Algorithm 4b: Computing a Public Key Candidate Kc from a Signature, Message, Signature Typecode pubtype, and identifiers I, q

```
 1. if the signature is not at least four bytes long, return INVALID

 2. parse sigtype, C, and y from the signature as follows:
    a. sigtype = strTou32(first 4 bytes of signature)

    b. if sigtype is not equal to pubtype, return INVALID

    c. set n and p according to the pubtype and Table 1;  if the
    signature is not exactly 4 + n * (p+1) bytes long, return INVALID

    d. C = next n bytes of signature

    e.  y[0] = next n bytes of signature
        y[1] = next n bytes of signature
        ...
      y[p-1] = next n bytes of signature

 3. compute the string Kc as follows
    Q = H(I || u32str(q) || u16str(D_MESG) || C || message)
    for ( i = 0; i < p; i = i + 1 ) {
      a = coef(Q || Cksm(Q), i, w)
      tmp = y[i]
      for ( j = a; j < 2^w - 1; j = j + 1 ) {
         tmp = H(I || u32str(q) || u16str(i) || u8str(j) || tmp)
      }
      z[i] = tmp
    }
    Kc = H(I || u32str(q) || u16str(D_PBLC) ||
                                z[0] || z[1] || ... || z[p-1])

 4. return Kc
```

5. Leighton-Micali Signatures

The Leighton-Micali Signature (LMS) method can sign a potentially large but fixed number of messages. An LMS system uses two cryptographic components: a one-time signature method and a hash function. Each LMS public/private key pair is associated with a perfect binary tree, each node of which contains an m-byte value, where m is the output length of the hash function. Each leaf of the tree contains the value of the public key of an LM-OTS public/private key pair. The value contained by the root of the tree is the LMS public key. Each interior node is computed by applying the hash function to the concatenation of the values of its children nodes.

Each node of the tree is associated with a node number, an unsigned integer that is denoted as node_num in the algorithms below, which is computed as follows. The root node has node number 1; for each node with node number N < 2^h (where h is the height of the tree), its left child has node number 2*N, while its right child has node number 2*N+1. The result of this is that each node within the tree will have a unique node number, and the leaves will have node numbers 2^h, (2^h)+1, (2^h)+2, ..., (2^h)+(2^h)-1. In general, the j-th node at level i has node number 2^i + j. The node number can conveniently be computed when it is needed in the LMS algorithms, as described in those algorithms.

5.1. Parameters

An LMS system has the following parameters:

h : the height of the tree, and

m : the number of bytes associated with each node.

H : a second-preimage-resistant cryptographic hash function that accepts byte strings of any length, and returns an m-byte string.

There are 2^h leaves in the tree.

The overall strength of the LMS signatures is governed by the weaker of the hash function used within the LM-OTS and the hash function used within the LMS system. In order to minimize the risk, these two hash functions SHOULD be the same (so that an attacker could not take advantage of the weaker hash function choice).

Name	H	m	h
LMS_SHA256_M32_H5	SHA256	32	5
LMS_SHA256_M32_H10	SHA256	32	10
LMS_SHA256_M32_H15	SHA256	32	15
LMS_SHA256_M32_H20	SHA256	32	20
LMS_SHA256_M32_H25	SHA256	32	25

Table 2

5.2. LMS Private Key

The format of the LMS private key is an internal matter to the implementation, and this document does not attempt to define it. One possibility is that it may consist of an array OTS_PRIV[] of 2^h LM-OTS private keys, and the leaf number q of the next LM-OTS private key that has not yet been used. The q-th element of OTS_PRIV[] is generated using Algorithm 0 with the identifiers I, q. The leaf number q is initialized to zero when the LMS private key is created. The process is as follows:

Algorithm 5: Computing an LMS Private Key.

1. determine h and m from the typecode and Table 2.

2. set I to a uniformly random 16-byte string

3. compute the array OTS_PRIV[] as follows:
```
for ( q = 0; q < 2^h; q = q + 1) {
   OTS_PRIV[q] = LM-OTS private key with identifiers I, q
}
```

4. q = 0

An LMS private key MAY be generated pseudorandomly from a secret value, in which case the secret value MUST be at least m bytes long, be uniformly random, and MUST NOT be used for any other purpose than the generation of the LMS private key. The details of how this process is done do not affect interoperability; that is, the public key verification operation is independent of these details. Appendix A provides an example of a pseudorandom method for computing an LMS private key.

The signature generation logic uses q as the next leaf to use, hence step 4 starts it off at the left-most one. Because the signature proces increments q after the signature operation, the first signature will have q=0.

5.3. LMS Public Key

An LMS public key is defined as follows, where we denote the public key final hash value (namely, the K value computed in Algorithm 1) associated with the i-th LM-OTS private key as OTS_PUB_HASH[i], with i ranging from 0 to (2^h)-1. Each instance of an LMS public/private key pair is associated with a balanced binary tree, and the nodes of that tree are indexed from 1 to 2^(h+1)-1. Each node is associated with an m-byte string, and the string for the r-th node is denoted as T[r] and is defined as

```
if r >= 2^h:
     H(I||u32str(r)||u16str(D_LEAF)||OTS_PUB_HASH[r-2^h])
else
     H(I||u32str(r)||u16str(D_INTR)||T[2*r]||T[2*r+1])
```

where D_LEAF is the fixed two byte value 0x8282, and D_INTR is the fixed two byte value 0x8383, both of which are used to distinguish this hash from every other hash in this system.

When we have r >= 2^h, then we are processing a leaf node (and thus hashing only a single LM-OTS public key). When we have r < 2^h, then we are processing an internal node, that is, a node with two child nodes that we need to combine.

The LMS public key is the string

```
u32str(type) || u32str(otstype) || I || T[1]
```

Section 3.3 specifies the format of the type variable. The value otstype is the parameter set for the LM-OTS public/private keypairs used. The value I is the private key identifier, and is the value used for all computations for the same LMS tree. The value T[1] can be computed via recursive application of the above equation, or by any equivalent method. An iterative procedure is outlined in Appendix C.

5.4. LMS Signature

An LMS signature consists of

the number q of the leaf associated with the LM-OTS signature, as a four-byte unsigned integer in network byte order,

an LM-OTS signature,

a typecode indicating the particular LMS algorithm,

an array of h m-byte values that is associated with the path through the tree from the leaf associated with the LM-OTS signature to the root.

Symbolically, the signature can be represented as

```
u32str(q) || lmots_signature || u32str(type) ||
          path[0] || path[1] || path[2] || ... || path[h-1]
```

Section 3.3 specifies the typecode and more formally defines the format. The array for a tree with height h will have h values and

contains the values of the siblings of (that is, is adjacent to) the nodes on the path from the leaf to the root, where the sibling to node A is the other node which shares node A's parent. In the signature, 0 is counted from the bottom level of the tree, and so path[0] is the value of the node adjacent to leaf node q; path[1] is the second level node that is adjacent to leaf node q's parent, and so up the tree until we get to path[h-1], which is the value of the next-to-the-top level node that leaf node q does not reside in.

Below is a simple example of the authentication path for h=3 and q=2. The leaf marked OTS is the one time signature which is used to sign the actual message. The nodes on the path from the OTS public key to the root are marked with a *, while the nodes that are used within the path array are marked with a **. The values in the path array are those nodes which are siblings of the nodes on the path; path[0] is the leaf** node that is adjacent to the OTS public key (which is the start of the path); path[1] is the T[4]** node which is the sibling of the second node T[5]* on the path, and path[2] is the T[3]** node which is the sibling of the third node T[2]* on the path.

```
                              Root
                               |
                 ---------------------------------
                 |                               |
               T[2]*                           T[3]**
                 |                               |
           ------------------            -----------------
           |                |            |               |
       T[4]**           T[5]*         T[6]            T[7]
           |                |            |               |
      ---------       ----------     --------       ---------
      |       |       |        |     |      |       |       |
    leaf    leaf    OTS   leaf**   leaf   leaf    leaf    leaf
```

The idea behind this authentication path is that it allows us to validate the OTS hash with using h path array values and hash computations. What the verifier does is recompute the hashes up the path; first, he hashes the given OTS and path[0] value, giving a tentative T[5]' value. Then, he hashes his path[1] and tentative T[5]' value to get a tentative T[2]' value. Then, he hashes that and the path[2] value to get a tentative Root' value. If that value is the known public key of the Merkle tree, then we can assume that the value T[2]' he got was the correct T[2] value in the original tree, and so the T[5]' value he got was the correct T[5] value in the original tree, and so the OTS public key is the same as in the original, and hence is correct.

5.4.1. LMS Signature Generation

To compute the LMS signature of a message with an LMS private key, the signer first computes the LM-OTS signature of the message using the leaf number of the next unused LM-OTS private key. The leaf number q in the signature is set to the leaf number of the LMS private key that was used in the signature. Before releasing the signature, the leaf number q in the LMS private key MUST be incremented, to prevent the LM-OTS private key from being used again. If the LMS private key is maintained in nonvolatile memory, then the implementation MUST ensure that the incremented value has been stored before releasing the signature. The issue this tries to prevent is a scenario where a) we generate a signature, using one LM-OTS private key, and release it to the application, b) before we update the nonvolatile memory, we crash, and c) we reboot, and generate a second signature using the same LM-OTS private key; with two different signatures using the same LM-OTS private key, someone could potentially generate a forged signature of a third message.

The array of node values in the signature MAY be computed in any way. There are many potential time/storage tradeoffs that can be applied. The fastest alternative is to store all of the nodes of the tree and set the array in the signature by copying them; pseudocode to do so appears in Appendix D. The least storage intensive alternative is to recompute all of the nodes for each signature. Note that the details of this procedure are not important for interoperability; it is not necessary to know any of these details in order to perform the signature verification operation. The internal nodes of the tree need not be kept secret, and thus a node-caching scheme that stores only internal nodes can sidestep the need for strong protections.

Several useful time/storage tradeoffs are described in the 'Small-Memory LM Schemes' section of [USPTO5432852].

5.4.2. LMS Signature Verification

An LMS signature is verified by first using the LM-OTS signature verification algorithm (Algorithm 4b) to compute the LM-OTS public key from the LM-OTS signature and the message. The value of that public key is then assigned to the associated leaf of the LMS tree, then the root of the tree is computed from the leaf value and the array path[] as described in Algorithm 6 below. If the root value matches the public key, then the signature is valid; otherwise, the signature fails.

Algorithm 6: LMS Signature Verification

1. if the public key is not at least eight bytes long, return INVALID

2. parse pubtype, I, and T[1] from the public key as follows:

   a. pubtype = strTou32(first 4 bytes of public key)

   b. ots_typecode = strTou32(next 4 bytes of public key)

   c. set m according to pubtype, based on Table 2

   d. if the public key is not exactly 24 + m bytes long, return INVALID

   e. I = next 16 bytes of the public key

   f. T[1] = next m bytes of the public key

3. compute the LMS Public Key Candidate Tc from the signature, message, identifier, pubtype and ots_typecode using Algorithm 6a.

4. if Tc is equal to T[1], return VALID; otherwise, return INVALID

Algorithm 6a: Computing an LMS Public Key Candidate from a Signature, Message, Identifier, and algorithm typecode

1. if the signature is not at least eight bytes long, return INVALID

2. parse sigtype, q, lmots_signature, and path from the signature as follows:

   a. q = strTou32(first 4 bytes of signature)

   b. otssigtype = strTou32(next 4 bytes of signature)

   c. if otssigtype is not the OTS typecode from the public key, return INVALID

   d. set n, p according to otssigtype and Table 1; if the signature is not at least 12 + n * (p + 1) bytes long, return INVALID

   e. lmots_signature = bytes 4 through 7 + n * (p + 1) of signature

   f. sigtype = strTou32(bytes 8 + n * (p + 1)) through

```
         11 + n * (p + 1) of signature)

      f. if sigtype is not the LM typecode from the public key,
             return INVALID

      g. set m, h according to sigtype and Table 2

      h. if q >= 2^h or the signature is not exactly
         12 + n * (p + 1) + m * h bytes long,
             return INVALID

      i. set path as follows:
            path[0] = next m bytes of signature
            path[1] = next m bytes of signature
            ...
            path[h-1] = next m bytes of signature

    3. Kc = candidate public key computed by applying Algorithm 4b
       to the signature lmots_signature, the message, and the
       identifiers I, q

    4. compute the candidate LMS root value Tc as follows:
       node_num = 2^h + q
       tmp = H(I || u32str(node_num) || u16str(D_LEAF) || Kc)
       i = 0
       while (node_num > 1) {
         if (node_num is odd):
           tmp = H(I||u32str(node_num/2)||u16str(D_INTR)||path[i]||tmp)
         else:
           tmp = H(I||u32str(node_num/2)||u16str(D_INTR)||tmp||path[i])
         node_num = node_num/2
         i = i + 1
       }
       Tc = tmp

    5. return Tc
```

6. Hierarchical signatures

In scenarios where it is necessary to minimize the time taken by the public key generation process, a Hierarchical N-time Signature System (HSS) can be used. This hierarchical scheme, which we describe in this section, uses the LMS scheme as a component. In HSS, we have a sequence of L LMS trees, where the public key for the first LMS tree is included in the public key of the HSS system, and where each LMS private key signs the next LMS public key, and where the last LMS private key signs the actual message. For example, if we have a three level hierarchy (L=3), then to sign a message, we would have:

The first LMS private key (level 0) signs a level 1 LMS public key.

The second LMS private key (level 1) signs a level 2 LMS public key.

The third LMS private key (level 2) signs the message.

The root of the level 0 LMS tree is contained in the HSS public key.

To verify the LMS signature, we would verify all the signatures:

We would verify that the level 1 LMS public key is correctly signed by the level 0 signature.

We would verify that the level 2 LMS public key is correctly signed by the level 1 signature.

We would verify that the message is correctly signed by the level 2 signature.

We would accept the HSS signature only if all the signatures validated.

During the signature generation process, we sign messages with the lowest (level L-1) LMS tree. Once we have used all the leafs in that tree to sign messages, we would discard it, generate a fresh LMS tree, and sign it with the next (level L-2) LMS tree (and when that is used up, recursively generate and sign a fresh level L-2 LMS tree).

HSS, in essence, utilizes a tree of LMS trees. There is a single LMS tree at level 0 (the root). Each LMS tree (actually, the private key corresponding to the LMS tree) at level i is used to sign 2^h objects (where h is the height of trees at level i). If i < L-1, then each object will be another LMS tree (actually, the public key) at level i+1; if i = L-1, we've reached the bottom of the HSS tree, and so each object is a message from the application. The HSS public key contains the public key of the LMS tree at the root, and an HSS signature is associated with a path from the root of the HSS tree to the leaf.

Compared to LMS, HSS has a much reduced public key generation time, as only the root tree needs to be generated prior to the distribution of the HSS public key. For example, a L=3 tree (with h=10 at each level) would have 1 level 0 LMS tree, 2^10 level 1 LMS trees (with each such level 1 public key signed by one of the 1024 level 0 OTS public keys), and 2^20 level 2 LMS trees. Only 1024 OTS public keys

need to be computed to generate the HSS public key (as you need to compute only the level 0 LMS tree to compute that value; you can, of course, decide to compute the initial level 1 and level 2 LMS trees). And, the 2^20 level 2 LMS trees can jointly sign a total of over a billion messages. In contrast, a single LMS tree that could sign a billion messages would require a billion OTS public keys to be computed first (even if h=30 were allowed in a supported parameter set).

Each LMS tree within the hierarchy is associated with a distinct LMS public key, private key, signature, and identifier. The number of levels is denoted L, and is between one and eight, inclusive. The following notation is used, where i is an integer between 0 and L-1 inclusive, and the root of the hierarchy is level 0:

prv[i] is the current LMS private key of the i-th level.

pub[i] is the current LMS public key of the i-th level, as described in Section 5.3.

sig[i] is the LMS signature of public key pub[i+1] generated using the private key prv[i].

It is expected that the above arrays are maintained for the course of the HSS key. The contents of the prv[] array MUST be kept private; the pub[] and sig[] array may be revealed, should the implementation find that convienent.

In this section, we say that an N-time private key is exhausted when it has generated N signatures, and thus it can no longer be used for signing.

For i > 0, the values prv[i], pub[i] and (for all values of i) sig[i] will be updated over time, as private keys are exhausted, and replaced by newer keys.

When these keys pairs are updated (or initially generated before the first message is signed), then the LMS key generation processes outlined in sections Section 5.2 and Section 5.3 are performed. If the generated key pairs are for level i of the HSS hierarchy, then we store the public key in pub[i] and the private key in prv[i]. In addition, if i > 0, then we sign the generated public key with the LMS private key at level i-1, placing the signature into sig[i-1]. When the LMS key pair are generated, the key pair and the corresponding identifier MUST be generated indepently of all other keypairs.

HSS allows L=1, in which case the HSS public key and signature formats are essentially the LMS public key and signature formats, prepended by a fixed field. Since HSS with L=1 has very little overhead compared to LMS, all implementations MUST support HSS in order to maximize interoperability.

We specifically allow different LMS levels to use different parameter sets. For example, the 0-th LMS public key (the root) may use the LMS_SHA256_M32_H15 parameter set, while the 1-th public key may use LMS_SHA256_M32_H10. There are practical reasons to allow this; for one, the signer may decide to store parts of the 0-th LMS tree (that it needs to construct while computing the public key) to accelerate later operations. As the 0-th tree is never updated, these internal nodes will never need to be recomputed. In addition, during the signature generation operation, almost all the operations involved with updating the authentication path occurs with the bottom (L-1th) LMS public key; hence it may be useful to make the tree that implements that to be shorter.

A close reading of the HSS verification pseudocode would show that it would allow the parameters of the non-top LMS public keys to change over time; for example, the signer might initially have the 1-th LMS public key to use LMS_SHA256_M32_H10, but when that tree is exhausted, the signer might replace it with LMS_SHA256_M32_H15 LMS public key. While this would work with the example verification pseudocode, the signer MUST NOT change the parameter sets for a specific level. This prohibition is to support verifiers that may keep state over the course of several signature verifications.

6.1. Key Generation

The public key of the HSS scheme consists of the number of levels L, followed by pub[0], the public key of the top level.

The HSS private key consists of prv[0], ... , prv[L-1], along with the associated pub[0], ... pub[L-1] and sig[0], ..., sig[L-2] values. As stated earlier, the values of the pub[] and sig[] arrays need not be kept secret, but may be revealed. The value of pub[0] does not change (and, except for the index q, the value of prv[0] need not change), though the values of pub[i] and prv[i] are dynamic for i > 0, and are changed by the signature generation algorithm.

During the key generation, the public and private keys are initialized. Here is some pseudocode that explains the key generation logic

Algorithm 7: Generating an HSS keypair

1. generate an LMS key pair, as specified in sections 5.2 and 5.3, placing the private key into priv[0], and the public key into pub[0]

2. for i = 1 to L-1 do {
   generate an LMS key pair, placing the private key into priv[i] and the public key into pub[i]

   sig[i-1] = lms_signature(pub[i], priv[i-1])
   }

3. return u32str(L) || pub[0] as the public key, and the priv[], pub[] and sig[] arrays as the private key

In the above algorithm, each LMS public/private keypair generated MUST be generated indepedently.

Note that the value of the public key does not depend on the execution of step 2. As a result, an implementation may decide to delay step 2 until later, for example, during the initial signature generation operation.

6.2. Signature Generation

To sign a message using an HSS keypair, the following steps are performed:

If prv[L-1] is exhausted, then determine the smallest integer d such that all of the private keys prv[d], prv[d+1], ... , prv[L-1] are exhausted. If d is equal to zero, then the HSS key pair is exhausted, and it MUST NOT generate any more signatures. Otherwise, the key pairs for levels d through L-1 must be regenerated during the signature generation process, as follows. For i from d to L-1, a new LMS public and private key pair with a new identifier is generated, pub[i] and prv[i] are set to those values, then the public key pub[i] is signed with prv[i-1], and sig[i-1] is set to the resulting value.

The message is signed with prv[L-1], and the value sig[L-1] is set to that result.

The value of the HSS signature is set as follows. We let signed_pub_key denote an array of octet strings, where signed_pub_key[i] = sig[i] || pub[i+1], for i between 0 and Nspk-1, inclusive, where Nspk = L-1 denotes the number of signed public

keys. Then the HSS signature is u32str(Nspk) ||
signed_pub_key[0] || ... || signed_pub_key[Nspk-1] || sig[Nspk].

Note that the number of signed_pub_key elements in the signature is indicated by the value Nspk that appears in the initial four bytes of the signature.

Here is some pseudocode of the above logic

Algorithm 8: Generating an HSS signature

1. If the message-signing key prv[L-1] is exhausted, regenerate that key pair, together with any parent key pairs that might be necessary.
   If the root key pair is exhausted, then the HSS key pair is exhausted and it MUST NOT generate any more signatures.

```
d = L
while (prv[d-1].q == 2^(prv[d-1].h)) {
    d = d - 1
    if (d == 0)
        return FAILURE
}
while (d < L) {
    create lms keypair pub[d], prv[d]
    sig[d-1] = lms_signature( pub[d], prv[d-1] )
    d = d + 1
}
```

2. sign the message
   sig[L-1] = lms_signature(msg, prv[L-1])

3. Create the list of signed public keys

```
i = 0;
while (i < L-1) {
    signed_pub_key[i] = sig[i] || pub[i+1]
    i = i + 1
}
```

4. return u32str(L-1) || signed_pub_key[0] || ...
   || signed_pub_key[L-2] || sig[L-1]

In the specific case of L=1, the format of an HSS signature is

u32str(0) || sig[0]

In the general case, the format of an HSS signature is

```
u32str(Nspk) || signed_pub_key[0] || ...
                         || signed_pub_key[Nspk-1] || sig[Nspk]
```

which is equivalent to

```
u32str(Nspk) || sig[0] || pub[1] || ...
                         || sig[Nspk-1] || pub[Nspk] || sig[Nspk]
```

6.3. Signature Verification

To verify a signature S and message using the public key pub, the following steps are performed:

```
The signature S is parsed into its components as follows:

Nspk = strTou32(first four bytes of S)
if Nspk+1 is not equal to the number of levels L in pub:
   return INVALID
for (i = 0; i < Nspk; i = i + 1) {
   siglist[i] = next LMS signature parsed from S
   publist[i] = next LMS public key parsed from S
}
siglist[Nspk] = next LMS signature parsed from S

key = pub
for (i = 0; i < Nspk; i = i + 1) {
   sig = siglist[i]
   msg = publist[i]
   if (lms_verify(msg, key, sig) != VALID):
       return INVALID
   key = msg
}
return lms_verify(message, key, siglist[Nspk])
```

Since the length of an LMS signature cannot be known without parsing it, the HSS signature verification algorithm makes use of an LMS signature parsing routine that takes as input a string consisting of an LMS signature with an arbitrary string appended to it, and returns both the LMS signature and the appended string. The latter is passed on for further processing.

6.4. Parameter Set Recommendations

As for guidance as to the number of LMS level, and the size of each, any discussion of performance is implementation specific. In general, the sole drawback for a single LMS tree is the time it takes

to generate the public key; as every LM-OTS public key needs to be generated, the time this takes can be substantial. For a two level tree, only the top level LMS tree and the initial bottom level LMS tree needs to be generated initially (before the first signature is generated); this will in general be significantly quicker.

To give a general idea on the trade-offs available, we include some measurements taken with the github.com/cisco/hash-sigs LMS implementation, taken on a 3.3 GHz Xeon processor, with threading enabled. We tried various parameter sets, all with W=8 (which minimizes signature size, while increasing time). These are here to give a guideline as to what's possible; for the computational time, your mileage may vary, depending on the computing resources you have. The machine these tests were performed on does not have the SHA-256 extensions; you could possibly do significantly better.

ParmSet	KeyGenTime	SigSize	KeyLifetime
15	6 sec	1616	30 seconds
20	3 min	1776	16 minutes
25	1.5 hour	1936	9 hours
15/10	6 sec	3172	9 hours
15/15	6 sec	3332	12 days
20/10	3 min	3332	12 days
20/15	3 min	3492	1 year
25/10	1.5 hour	3492	1 year
25/15	1.5 hour	3652	34 years

Table 3

ParmSet: this is the height of the Merkle tree(s); parameter sets listed as a single integer have L=1, and consist a single Merkle tree of that height; parameter sets with L=2 are listed as x/y, with x being the height of the top level Merkle tree, and y being the bottom level.

KeyGenTime: the measured key generation time; that is, the time needed to generate the public private key pair.

SigSize: the size of a signature (in bytes)

KeyLifetime: the lifetime of a key, assuming we generated 1000 signatures per second. In practice, we're not likely to get anywhere close to 1000 signatures per second sustained; if you have a more appropriate figure for your scenario, this column is pretty easy to recompute.

As for signature generation or verification times, those are moderately insensitive to the above parameter settings (except for the Winternitz setting, and the number of Merkle trees for verification). Tests on the same machine (without multithreading) gave approximately 4msec to sign a short message, 2.6msec to verify; these tests used a two level ParmSet; a single level would approximately halve the verification time. All times can be significantly improved (by perhaps a factor of 8) by using a parameter set with W=4; however that also about doubles the signature size.

7. Rationale

The goal of this note is to describe the LM-OTS, LMS and HSS algorithms following the original references and present the modern security analysis of those algorithms. Other signature methods are out of scope and may be interesting follow-on work.

We adopt the techniques described by Leighton and Micali to mitigate attacks that amortize their work over multiple invocations of the hash function.

The values taken by the identifier I across different LMS public/private key pairs are chosen randomly in order to improve security. The analysis of this method in [Fluhrer17] shows that we do not need uniqueness to ensure security; we do need to ensure that we don't have a large number of private keys that use the same I value. By randomly selecting 16 byte I values, the chance that, out of 2^64 private keys, 4 or more of them will use the same I value is negligible (that is, has probability less than 2^-128).

The reason 16 bytes I values were selected was to optimize the Winternitz hash chain operation. With the current settings, the value being hashed is exactly 55 bytes long (for a 32 byte hash function), which SHA-256 can hash in a single hash compression operation. Other hash functions may be used in future specifications; all the ones that we will be likely to support (SHA-512/256 and the various SHA-3 hashes) would work well with a 16-byte I value.

The signature and public key formats are designed so that they are relatively easy to parse. Each format starts with a 32-bit enumeration value that indicates the details of the signature algorithm and provides all of the information that is needed in order to parse the format.

The Checksum Section 4.4 is calculated using a non-negative integer "sum", whose width was chosen to be an integer number of w-bit fields such that it is capable of holding the difference of the total possible number of applications of the function H as defined in the signing algorithm of Section 4.5 and the total actual number. In the case that the number of times H is applied is 0, the sum is (2^w - 1) * (8*n/w). Thus for the purposes of this document, which describes signature methods based on H = SHA256 (n = 32 bytes) and w = { 1, 2, 4, 8 }, the sum variable is a 16-bit non-negative integer for all combinations of n and w. The calculation uses the parameter ls defined in Section 4.1 and calculated in Appendix B, which indicates the number of bits used in the left-shift operation.

7.1. Security String

To improve security against attacks that amortize their effort against multiple invocations of the hash function, Leighton and Micali introduce a "security string" that is distinct for each invocation of that function. Whenever this process computes a hash, the string being hashed will start with a string formed from the below fields. These fields will appear in fixed locations in the value we compute the hash of, and so we list where in the hash these fields would be present. These fields that make up this string are:

I - a 16-byte identifier for the LMS public/private key pair. It MUST be chosen uniformly at random, or via a pseudorandom process, at the time that a key pair is generated, in order to minimize the probability that any specific value of I be used for a large number of different LMS private keys. This is always bytes 0-15 of the value being hashed.

r - in the LMS N-time signature scheme, the node number r associated with a particular node of a hash tree is used as an input to the hash used to compute that node. This value is represented as a 32-bit (four byte) unsigned integer in network byte order. Either r or q (depending on the domain separation parameter) will be bytes 16-19 of the value being hashed.

q - in the LMS N-time signature scheme, each LM-OTS signature is associated with the leaf of a hash tree, and q is set to the leaf number. This ensures that a distinct value of q is used for each distinct LM-OTS public/private key pair. This value is

represented as a 32-bit (four byte) unsigned integer in network byte order. Either r or q (depending on the domain separation parameter) will be bytes 16-19 of the value being hashed.

D - a domain separation parameter, which is a two byte identifier that takes on different values in the different contexts in which the hash function is invoked. D occurs in bytes 20, 21 of the value being hashed and takes on the following values:

D_PBLC = 0x8080 when computing the hash of all of the iterates in the LM-OTS algorithm

D_MESG = 0x8181 when computing the hash of the message in the LM-OTS algorithms

D_LEAF = 0x8282 when computing the hash of the leaf of an LMS tree

D_INTR = 0x8383 when computing the hash of an interior node of an LMS tree

i - a value between 0 and 264; this is used in the LM-OTS scheme, when either computing the iterations of the Winternitz chain, or when using the suggested LM-OTS private key generation process. It is represented as a 16-bit (two-byte) unsigned integer in network byte order. If present, it occurs at bytes 20, 21 of the value being hashed.

j - in the LM-OTS scheme, j is the iteration number used when the private key element is being iteratively hashed. It is represented as an 8-bit (one byte) unsigned integer and is present if i is a value between 0 and 264. If present, it occurs at bytes 22 to 21+n of the value being hashed.

C - an n-byte randomizer that is included with the message whenever it is being hashed to improve security. C MUST be chosen uniformly at random, or via a pseudorandom process. It is present if D=D_MESG, and it occurs at bytes 22 to 21+n of the value being hashed.

8. IANA Considerations

The Internet Assigned Numbers Authority (IANA) is requested to create two registries: one for OTS signatures, which includes all of the LM-OTS signatures as defined in Section 4, and one for Leighton-Micali Signatures, as defined in Section 5.

Additions to these registries require that a specification be documented in an RFC or another permanent and readily available reference in sufficient detail that interoperability between independent implementations is possible. IANA MUST verify that all applications for additions to these registries hve first been reviewed by the IRTF Crypto Forum Research Group (CFRG).

Each entry in the registry contains the following elements:

a short name, such as "LMS_SHA256_M32_H10",

a positive number, and

a reference to a specification that completely defines the signature method test cases that can be used to verify the correctness of an implementation.

The numbers between 0xDDDDDDDD (decimal 3,722,304,989) and 0xFFFFFFFF (decimal 4,294,967,295) inclusive, will not be assigned by IANA, and are reserved for private use; no attempt will be made to prevent multiple sites from using the same value in different (and incompatible) ways [RFC2434].

The LM-OTS registry is as follows.

Name	Reference	Numeric Identifier
Reserved		0x00000000
LMOTS_SHA256_N32_W1	Section 4	0x00000001
LMOTS_SHA256_N32_W2	Section 4	0x00000002
LMOTS_SHA256_N32_W4	Section 4	0x00000003
LMOTS_SHA256_N32_W8	Section 4	0x00000004

Table 4

The LMS registry is as follows.

Name	Reference	Numeric Identifier
Reserved		0x0 - 0x4
LMS_SHA256_M32_H5	Section 5	0x00000005
LMS_SHA256_M32_H10	Section 5	0x00000006
LMS_SHA256_M32_H15	Section 5	0x00000007
LMS_SHA256_M32_H20	Section 5	0x00000008
LMS_SHA256_M32_H25	Section 5	0x00000009

Table 5

An IANA registration of a signature system does not constitute an endorsement of that system or its security.

The LM-OTS and the LMS registries currently occupy a disjoint set of values. This coincidence is a historical accident; the correctness of the system does not depend on this. IANA is not required to maintain this situation.

9. Security Considerations

The hash function H MUST have second preimage resistance: it must be computationally infeasible for an attacker that is given one message M to be able to find a second message M' such that H(M) = H(M').

The security goal of a signature system is to prevent forgeries. A successful forgery occurs when an attacker who does not know the private key associated with a public key can find a message (distinct from all previously signed ones) and signature that is valid with that public key (that is, the Signature Verification algorithm applied to that signature and message and public key will return VALID). Such an attacker, in the strongest case, may have the ability to forge valid signatures for an arbitrary number of other messages.

LMS is provably secure in the random oracle model, where the hash compression function is considered the random oracle, as shown by [Fluhrer17]. Corollary 1 of that paper states:

> If we have no more than 2^64 randomly chosen LMS private keys, allow the attacker access to a signing oracle and a SHA-256 hash

compression oracle, and allow a maximum of 2^120 hash compression computations, then the probability of an attacker being able to generate a single forgery against any of those LMS keys is less than 2^-129.

Many of the objects within the public key and the signature start with a typecode. A verifier MUST check each of these typecodes, and a verification operation on a signature with an unknown type, or a type that does not correspond to the type within the public key MUST return INVALID. The expected length of a variable-length object can be determined from its typecode, and if an object has a different length, then any signature computed from the object is INVALID.

9.1. Hash Formats

The format of the inputs to the hash function H have the property that each invocation of that function has an input that is repeated by a small bounded number of other inputs (due to potential repeats of the I value), and in particular, will vary somewhere in the first 23 bytes of the value being hashed. This property is important for a proof of security in the random oracle model.

The formats used during key generation and signing (including the recommended pseudorandom key generation procedure in Appendix A):

```
I || u32str(q) || u16str(i) || u8str(j) || tmp
I || u32str(q) || u16str(D_PBLC) || y[0] || ... || y[p-1]
I || u32str(q) || u16str(D_MESG) || C || message
I || u32str(r) || u16str(D_LEAF) || OTS_PUB_HASH[r-2^h]
I || u32str(r) || u16str(D_INTR) || T[2*r] || T[2*r+1]
I || u32str(q) || u16str(i) || u8str(0xff) || SEED
```

Each hash type listed is distinct; at locations 20, 21 of the value being hashed, there exists either a fixed value D_PBLC, D_MESG, D_LEAF, D_INTR, or a 16 bit value i. These fixed values are distinct from each other, and are large (over 32768), while the 16 bit values of i are small (currently no more than 265; possibly being slightly larger if larger hash functions are supported); hence the range of possible values of i will not collide any of the D_PBLC, D_MESG, D_LEAF, D_INTR identifiers. The only other collision possibility is the Winternitz chain hash colliding with the recommended pseudorandom key generation process; here, at location 22 of the value being hashed, the Winternitz chain function has the value u8str(j), where j is a value between 0 and 254, while location 22 of the recommended pseudorandom key generation process has value 255.

For the Winternitz chaining function, D_PBLC, and D_MESG, the value of I || u32str(q) is distinct for each LMS leaf (or equivalently, for

each q value). For the Winternitz chaining function, the value of u16str(i) || u8str(j) is distinct for each invocation of H for a given leaf. For D_PBLC and D_MESG, the input format is used only once for each value of q, and thus distinctness is assured. The formats for D_INTR and D_LEAF are used exactly once for each value of r, which ensures their distinctness. For the recommended pseudorandom key generation process, for a given value of I, q and j are distinct for each invocation of H.

The value of I is chosen uniformly at random from the set of all 128 bit strings. If 2^64 public keys are generated (and hence 2^64 random I values), there is a nontrivial probability of a duplicate (which would imply duplicate prefixes). However, there will be an extremely high probability there will not be a four-way collision (that is, any I value used for four distinct LMS keys; probability < 2^-132), and hence the number of repeats for any specific prefix will be limited to at most 3. This is shown (in [Fluhrer17]) to have only a limited effect on the security of the system.

9.2. Stateful signature algorithm

The LMS signature system, like all N-time signature systems, requires that the signer maintain state across different invocations of the signing algorithm, to ensure that none of the component one-time signature systems are used more than once. This section calls out some important practical considerations around this statefulness. These issues are discussed in greater detail in [STMGMT].

In a typical computing environment, a private key will be stored in non-volatile media such as on a hard drive. Before it is used to sign a message, it will be read into an application's Random Access Memory (RAM). After a signature is generated, the value of the private key will need to be updated by writing the new value of the private key into non-volatile storage. It is essential for security that the application ensure that this value is actually written into that storage, yet there may be one or more memory caches between it and the application. Memory caching is commonly done in the file system, and in a physical memory unit on the hard disk that is dedicated to that purpose. To ensure that the updated value is written to physical media, the application may need to take several special steps. In a POSIX environment, for instance, the O_SYNC flag (for the open() system call) will cause invocations of the write() system call to block the calling process until the data has been written to the underlying hardware. However, if that hardware has its own memory cache, it must be separately dealt with using an operating system or device specific tool such as hdparm to flush the on-drive cache, or turn off write caching for that drive. Because these details vary across different operating systems and devices,

this note does not attempt to provide complete guidance; instead, we call the implementer's attention to these issues.

When hierarchical signatures are used, an easy way to minimize the private key synchronization issues is to have the private key for the second level resident in RAM only, and never write that value into non-volatile memory. A new second level public/private key pair will be generated whenever the application (re)starts; thus, failures such as a power outage or application crash are automatically accommodated. Implementations SHOULD use this approach wherever possible.

9.3. Security of LM-OTS Checksum

To show the security of LM-OTS checksum, we consider the signature y of a message with a private key x and let h = H(message) and c = Cksm(H(message)) (see Section 4.5). To attempt a forgery, an attacker may try to change the values of h and c. Let h' and c' denote the values used in the forgery attempt. If for some integer j in the range 0 to u, where u = ceil(8*n/w) is the size of the range that the checksum value can cover, inclusive,

a' = coef(h', j, w),

a = coef(h, j, w), and

a' > a

then the attacker can compute F^a'(x[j]) from F^a(x[j]) = y[j] by iteratively applying function F to the j-th term of the signature an additional (a' - a) times. However, as a result of the increased number of hashing iterations, the checksum value c' will decrease from its original value of c. Thus a valid signature's checksum will have, for some number k in the range u to (p-1), inclusive,

b' = coef(c', k, w),

b = coef(c, k, w), and

b' < b

Due to the one-way property of F, the attacker cannot easily compute F^b'(x[k]) from F^b(x[k]) = y[k].

10. Comparison with other work

The eXtended Merkle Signature Scheme (XMSS) [XMSS], [RFC8391] is similar to HSS in several ways. Both are stateful hash based signature schemes, and both use a hierarchical approach, with a Merkle tree at each level of the hierarchy. XMSS signatures are slightly shorter than HSS signatures, for equivalent security and an equal number of signatures.

HSS has several advantages over XMSS. HSS operations are roughly four times faster than the comparable XMSS ones, when SHA256 is used as the underlying hash. This occurs because the hash operation done as a part of the Winternitz iterations dominates performance, and XMSS performs four compression function invocations (two for the PRF, two for the F function) where HSS needs only perform one. Additionally, HSS is somewhat simpler (as each hash invocation is just a prefix followed by the data being hashed).

11. Acknowledgements

Thanks are due to Chirag Shroff, Andreas Huelsing, Burt Kaliski, Eric Osterweil, Ahmed Kosba, Russ Housley, Philip Lafrance, Alexander Truskovsky, Mark Peruzel for constructive suggestions and valuable detailed review. We especially acknowledge Jerry Solinas, Laurie Law, and Kevin Igoe, who pointed out the security benefits of the approach of Leighton and Micali [USPTO5432852] and Jonathan Katz, who gave us security guidance, and Bruno Couillard and Jim Goodman for an especially thorough review.

12. References

12.1. Normative References

[FIPS180] National Institute of Standards and Technology, "Secure Hash Standard (SHS)", FIPS 180-4, March 2012.

[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>.

[RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 2434, DOI 10.17487/RFC2434, October 1998, <https://www.rfc-editor.org/info/rfc2434>.

[RFC3979] Bradner, S., Ed., "Intellectual Property Rights in IETF Technology", RFC 3979, DOI 10.17487/RFC3979, March 2005, <https://www.rfc-editor.org/info/rfc3979>.

[RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, RFC 4506, DOI 10.17487/RFC4506, May 2006, <https://www.rfc-editor.org/info/rfc4506>.

[RFC4879] Narten, T., "Clarification of the Third Party Disclosure Procedure in RFC 3979", RFC 4879, DOI 10.17487/RFC4879, April 2007, <https://www.rfc-editor.org/info/rfc4879>.

[USPTO5432852] Leighton, T. and S. Micali, "Large provably fast and secure digital signature schemes from secure hash functions", U.S. Patent 5,432,852, July 1995.

12.2. Informative References

[C:Merkle87] Merkle, R., "A Digital Signature Based on a Conventional Encryption Function", Lecture Notes in Computer Science crypto87vol, 1988.

[C:Merkle89a] Merkle, R., "A Certified Digital Signature", Lecture Notes in Computer Science crypto89vol, 1990.

[C:Merkle89b] Merkle, R., "One Way Hash Functions and DES", Lecture Notes in Computer Science crypto89vol, 1990.

[Fluhrer17] Fluhrer, S., "Further analysis of a proposed hash-based signature standard", EPrint https://eprint.iacr.org/2017/553.pdf, 2017.

[Grover96] Grover, L., "A fast quantum mechanical algorithm for database search", 28th ACM Symposium on the Theory of Computing p. 212, 1996.

[Katz16] Katz, J., "Analysis of a proposed hash-based signature standard", Security Standardization Research (SSR) Conference https://www.cs.umd.edu/~jkatz/papers/HashBasedSigs-SSR16.pdf, 2016.

[Merkle79]
Merkle, R., "Secrecy, Authentication, and Public Key Systems", Stanford University Information Systems Laboratory Technical Report 1979-1, 1979.

[RFC8391] Huelsing, A., Butin, D., Gazdag, S., Rijneveld, J., and A. Mohaisen, "XMSS: eXtended Merkle Signature Scheme", RFC 8391, DOI 10.17487/RFC8391, May 2018, <https://www.rfc-editor.org/info/rfc8391>.

[STMGMT] McGrew, D., Fluhrer, S., Kampanakis, P., Gazdag, S., Butin, D., and J. Buchmann, "State Management for Hash-based Signatures.", Security Standardization Resarch (SSR) Conference 224., 2016.

[XMSS] Buchmann, J., Dahmen, E., and Andreas Huelsing, "XMSS-a practical forward secure signature scheme based on minimal security assumptions.", International Workshop on Post-Quantum Cryptography Springer Berlin., 2011.

Appendix A. Pseudorandom Key Generation

An implementation MAY use the following pseudorandom process for generating an LMS private key.

SEED is an m-byte value that is generated uniformly at random at the start of the process,

I is LMS key pair identifier,

q denotes the LMS leaf number of an LM-OTS private key,

x_q denotes the x array of private elements in the LM-OTS private key with leaf number q,

i is the index of the private key element, and

H is the hash function used in LM-OTS.

The elements of the LM-OTS private keys are computed as:

x_q[i] = H(I || u32str(q) || u16str(i) || u8str(0xff) || SEED).

This process stretches the m-byte random value SEED into a (much larger) set of pseudorandom values, using a unique counter in each invocation of H. The format of the inputs to H are chosen so that they are distinct from all other uses of H in LMS and LM-OTS. A careful reader will note that this is similar to the hash we perform

when iterating through the Winternitz chain; however in that chain, the iteration index will vary between 0 and 254 maximum (for W=8), while the corresponding value in this formula is 255. This algorithm is included in the proof of security in [Fluhrer17] and hence this method is safe when used within the LMS system; however any other cryptographical secure method of generating private keys would also be safe.

Appendix B. LM-OTS Parameter Options

The LM-OTS one time signature method uses several internal parameters, which are a function of the selected parameter set. These internal parameters set:

p - This is the number of independent Winternitz chains used in the signature; it will be the number of w-bit digits needed to hold the n-bit hash (u in the below equations), along with the number of digits needed to hold the checksum (v in the below equations)

ls - This is the size of the shift needed to move the checksum so that it appears in the checksum digits

ls is needed because, while we express the checksum internally as a 16 bit value, we don't always express all 16 bits in the signature; for example, if w=4, we might use only the top 12 bits. Because we read the checksum in network order, this means that, without the shift, we'll use the higher order bits (which may be always 0), and omit the lower order bits (where the checksum value actually resides). This shift is here to ensure that the parts of the checksum we need to express (for security) actually contribute to the signature; when multiple such shifts are possible, we take the minimal value.

The parameters ls, and p are computed as follows:

```
u = ceil(8*n/w)
v = ceil((floor(lg((2^w - 1) * u)) + 1) / w)
ls = 16 - (v * w)
p = u + v
```

Here u and v represent the number of w-bit fields required to contain the hash of the message and the checksum byte strings, respectively. And as the value of p is the number of w-bit elements of (H(message) || Cksm(H(message))), it is also equivalently the number of byte strings that form the private key and the number of byte strings in the signature. The value 16 in the ls computation of ls corresponds to the 16 bits value used for the sum variable in Algorithm 2 in Section 4.4

A table illustrating various combinations of n and w with the associated values of u, v, ls, and p is provided in Table 6.

Hash Length in Bytes (n)	Winternitz Parameter (w)	w-bit Elements in Hash (u)	w-bit Elements in Checksum (v)	Left Shift (ls)	Total Number of w-bit Elements (p)
32	1	256	9	7	265
32	2	128	5	6	133
32	4	64	3	4	67
32	8	32	2	0	34

Table 6

Appendix C. An iterative algorithm for computing an LMS public key

The LMS public key can be computed using the following algorithm or any equivalent method. The algorithm uses a stack of hashes for data. It also makes use of a hash function with the typical init/update/final interface to hash functions; the result of the invocations hash_init(), hash_update(N[1]), hash_update(N[2]), ... , hash_update(N[n]), v = hash_final(), in that order, is identical to that of the invocation of H(N[1] || N[2] || ... || N[n]).

Generating an LMS Public Key from an LMS Private Key

```
  for ( i = 0; i < 2^h; i = i + 1 ) {
    r = i + num_lmots_keys;
    temp = H(I || u32str(r) || u16str(D_LEAF) || OTS_PUB_HASH[i])
    j = i;
    while (j % 2 == 1) {
      r = (r - 1)/2;
      j = (j-1) / 2;
      left_side = pop(data stack);
      temp = H(I || u32str(r) || u16str(D_INTR) || left_side || temp)
    }
    push temp onto the data stack
 }
 public_key = pop(data stack)
```

Note that this pseudocode expects that all 2^h leaves of the tree have equal depth; that is, num_lmots_keys to be a power of 2. The maximum depth of the stack will be h-1 elements, that is, a total of (h-1)*n bytes; for the currently defined parameter sets, this will never be more than 768 bytes of data.

Appendix D. Method for deriving authentication path for a signature

The LMS signature consists of u32str(q) || lmots_signature || u32str(type) || path[0] || path[1] || ... || path[h-1]. This appendix shows one method of constructing this signature, assuming that the implementation has stored the T[] array that was used to construct the public key. Note that this is not the only possible method; other methods exist which don't assume that you have the entire T[] array in memory. To construct a signature, you perform the following algorithm:

Generating an LMS Signature

1. set type to the typecode of the LMS algorithm

2. extract h from the typecode according to table 2

3. create the LM-OTS signature for the message:
   ots_signature = lmots_sign(message, LMS_PRIV[q])

4. compute the array path as follows:

```
i = 0
r = 2^h + q
while (i < h) {
    temp = (r / 2^i) xor 1
    path[i] = T[temp]
    i = i + 1
}
```

5. S = u32str(q) || ots_signature || u32str(type) ||
   path[0] || path[1] || ... || path[h-1]

6. q = q + 1

7. return S

where 'xor' is the bitwise exclusive-or operation, and / is integer division (that is, rounded down to an integer value)

Appendix E. Example Implementation

An example implementation can be found online at https://github.com/cisco/hash-sigs.

Appendix F. Test Cases

This section provides test cases that can be used to verify or debug an implementation. This data is formatted with the name of the elements on the left, and the value of the elements on the right, in hexadecimal. The concatenation of all of the values within a public key or signature produces that public key or signature, and values that do not fit within a single line are listed across successive lines.

Test Case 1 Public Key

```
---------------------------------------------
HSS public key
levels      00000002
---------------------------------------------
LMS type    00000005                        # LM_SHA256_M32_H5
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
I           61a5d57d37f5e46bfb7520806b07a1b8
K           50650e3b31fe4a773ea29a07f09cf2ea
            30e579f0df58ef8e298da0434cb2b878
---------------------------------------------
---------------------------------------------
```

Test Case 1 Message

```
---------------------------------------------
Message     54686520706f77657273206e6f742064  |The powers not d|
            656c65676174656420746f2074686520  |elegated to the |
            556e69746564205374617465732062 79  |United States by|
            2074686520436f6e737469747574696f  | the Constitutio|
            6e2c206e6f722070726f686962697465  |n, nor prohibite|
            642062792069742074 6f207468652053  |d by it to the S|
            74617465732c20617265207265736572  |tates, are reser|
            76656420746f207468652053746174 65  |ved to the State|
            73207265737065637469766 56c792c20  |s respectively, |
            6f7220746f207468652070656f706c65  |or to the people|
            2e0a                              |..|
---------------------------------------------
```

Test Case 1 Signature

```
---------------------------------------------
HSS signature
Nspk        00000001
sig[0]:
---------------------------------------------
LMS signature
q           00000005
---------------------------------------------
LMOTS signature
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
C           d32b56671d7eb98833c49b433c272586
            bc4a1c8a8970528ffa04b966f9426eb9
y[0]        965a25bfd37f196b9073f3d4a232feb6
            9128ec45146f86292f9dff9610a7bf95
y[1]        a64c7f60f6261a62043f86c70324b770
            7f5b4a8a6e19c114c7be866d488778a0
```

```
y[2]        e05fd5c6509a6e61d559cf1a77a970de
            927d60c70d3de31a7fa0100994e162a2
y[3]        582e8ff1b10cd99d4e8e413ef469559f
            7d7ed12c838342f9b9c96b83a4943d16
y[4]        81d84b15357ff48ca579f19f5e71f184
            66f2bbef4bf660c2518eb20de2f66e3b
y[5]        14784269d7d876f5d35d3fbfc7039a46
            2c716bb9f6891a7f41ad133e9e1f6d95
y[6]        60b960e7777c52f060492f2d7c660e14
            71e07e72655562035abc9a701b473ecb
y[7]        c3943c6b9c4f2405a3cb8bf8a691ca51
            d3f6ad2f428bab6f3a30f55dd9625563
y[8]        f0a75ee390e385e3ae0b906961ecf41a
            e073a0590c2eb6204f44831c26dd768c
y[9]        35b167b28ce8dc988a3748255230cef9
            9ebf14e730632f27414489808afab1d1
y[10]       e783ed04516de012498682212b078105
            79b250365941bcc98142da13609e9768
y[11]       aaf65de7620dabec29eb82a17fde35af
            15ad238c73f81bdb8dec2fc0e7f93270
y[12]       1099762b37f43c4a3c20010a3d72e2f6
            06be108d310e639f09ce7286800d9ef8
y[13]       a1a40281cc5a7ea98d2adc7c7400c2fe
            5a101552df4e3cccfd0cbf2ddf5dc677
y[14]       9cbbc68fee0c3efe4ec22b83a2caa3e4
            8e0809a0a750b73ccdcf3c79e6580c15
y[15]       4f8a58f7f24335eec5c5eb5e0cf01dcf
            4439424095fceb077f66ded5bec73b27
y[16]       c5b9f64a2a9af2f07c05e99e5cf80f00
            252e39db32f6c19674f190c9fbc506d8
y[17]       26857713afd2ca6bb85cd8c107347552
            f30575a5417816ab4db3f603f2df56fb
y[18]       c413e7d0acd8bdd81352b2471fc1bc4f
            1ef296fea1220403466b1afe78b94f7e
y[19]       cf7cc62fb92be14f18c2192384ebceaf
            8801afdf947f698ce9c6ceb696ed70e9
y[20]       e87b0144417e8d7baf25eb5f70f09f01
            6fc925b4db048ab8d8cb2a661ce3b57a
y[21]       da67571f5dd546fc22cb1f97e0ebd1a6
            5926b1234fd04f171cf469c76b884cf3
y[22]       115cce6f792cc84e36da58960c5f1d76
            0f32c12faef477e94c92eb75625b6a37
y[23]       1efc72d60ca5e908b3a7dd69fef02491
            50e3eebdfed39cbdc3ce9704882a2072
y[24]       c75e13527b7a581a556168783dc1e975
            45e31865ddc46b3c957835da252bb732
y[25]       8d3ee2062445dfb85ef8c35f8e1f3371
            af34023cef626e0af1e0bc017351aae2
```

```
y[26]       ab8f5c612ead0b729a1d059d02bfe18e
            fa971b7300e882360a93b025ff97e9e0
y[27]       eec0f3f3f13039a17f88b0cf808f4884
            31606cb13f9241f40f44e537d302c64a
y[28]       4f1f4ab949b9feefadcb71ab50ef27d6
            d6ca8510f150c85fb525bf25703df720
y[29]       9b6066f09c37280d59128d2f0f637c7d
            7d7fad4ed1c1ea04e628d221e3d8db77
y[30]       b7c878c9411cafc5071a34a00f4cf077
            38912753dfce48f07576f0d4f94f42c6
y[31]       d76f7ce973e9367095ba7e9a3649b7f4
            61d9f9ac1332a4d1044c96aefee67676
y[32]       401b64457c54d65fef6500c59cdfb69a
            f7b6dddfcb0f086278dd8ad0686078df
y[33]       b0f3f79cd893d314168648499898fbc0
            ced5f95b74e8ff14d735cdea968bee74
---------------------------------------------
LMS type    00000005                        # LM_SHA256_M32_H5
path[0]     d8b8112f9200a5e50c4a262165bd342c
            d800b8496810bc716277435ac376728d
path[1]     129ac6eda839a6f357b5a04387c5ce97
            382a78f2a4372917eefcbf93f63bb591
path[2]     12f5dbe400bd49e4501e859f885bf073
            6e90a509b30a26bfac8c17b5991c157e
path[3]     b5971115aa39efd8d564a6b90282c316
            8af2d30ef89d51bf14654510a12b8a14
path[4]     4cca1848cf7da59cc2b3d9d0692dd2a2
            0ba3863480e25b1b85ee860c62bf5136
---------------------------------------------
LMS public key
LMS type    00000005                        # LM_SHA256_M32_H5
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
I           d2f14ff6346af964569f7d6cb880a1b6
K           6c5004917da6eafe4d9ef6c6407b3db0
            e5485b122d9ebe15cda93cfec582d7ab
---------------------------------------------
final_signature:
---------------------------------------------
LMS signature
q           0000000a
---------------------------------------------
LMOTS signature
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
C           0703c491e7558b35011ece3592eaa5da
            4d918786771233e8353bc4f62323185c
y[0]        95cae05b899e35dffd71705470620998
            8ebfdf6e37960bb5c38d7657e8bffeef
y[1]        9bc042da4b4525650485c66d0ce19b31
```

```
                7587c6ba4bffcc428e25d08931e72dfb
   y[2]         6a120c5612344258b85efdb7db1db9e1
                865a73caf96557eb39ed3e3f426933ac
   y[3]         9eeddb03a1d2374af7bf771855774562
                37f9de2d60113c23f846df26fa942008
   y[4]         a698994c0827d90e86d43e0df7f4bfcd
                b09b86a373b98288b7094ad81a0185ac
   y[5]         100e4f2c5fc38c003c1ab6fea479eb2f
                5ebe48f584d7159b8ada03586e65ad9c
   y[6]         969f6aecbfe44cf356888a7b15a3ff07
                4f771760b26f9c04884ee1faa329fbf4
   y[7]         e61af23aee7fa5d4d9a5dfcf43c4c26c
                e8aea2ce8a2990d7ba7b57108b47dabf
   y[8]         beadb2b25b3cacc1ac0cef346cbb90fb
                044beee4fac2603a442bdf7e507243b7
   y[9]         319c9944b1586e899d431c7f91bcccc8
                690dbf59b28386b2315f3d36ef2eaa3c
   y[10]        f30b2b51f48b71b003dfb08249484201
                043f65f5a3ef6bbd61ddfee81aca9ce6
   y[11]        0081262a00000480dcbc9a3da6fbef5c
                1c0a55e48a0e729f9184fcb1407c3152
   y[12]        9db268f6fe50032a363c9801306837fa
                fabdf957fd97eafc80dbd165e435d0e2
   y[13]        dfd836a28b354023924b6fb7e48bc0b3
                ed95eea64c2d402f4d734c8dc26f3ac5
   y[14]        91825daef01eae3c38e3328d00a77dc6
                57034f287ccb0f0e1c9a7cbdc828f627
   y[15]        205e4737b84b58376551d44c12c3c215
                c812a0970789c83de51d6ad787271963
   y[16]        327f0a5fbb6b5907dec02c9a90934af5
                a1c63b72c82653605d1dcce51596b3c2
   y[17]        b45696689f2eb382007497557692caac
                4d57b5de9f5569bc2ad0137fd47fb47e
   y[18]        664fcb6db4971f5b3e07aceda9ac130e
                9f38182de994cff192ec0e82fd6d4cb7
   y[19]        f3fe00812589b7a7ce51544045643301
                6b84a59bec6619a1c6c0b37dd1450ed4
   y[20]        f2d8b584410ceda8025f5d2d8dd0d217
                6fc1cf2cc06fa8c82bed4d944e71339e
   y[21]        ce780fd025bd41ec34ebff9d4270a322
                4e019fcb444474d482fd2dbe75efb203
   y[22]        89cc10cd600abb54c47ede93e08c114e
                db04117d714dc1d525e11bed8756192f
   y[23]        929d15462b939ff3f52f2252da2ed64d
                8fae88818b1efa2c7b08c8794fb1b214
   y[24]        aa233db3162833141ea4383f1a6f120b
                e1db82ce3630b3429114463157a64e91
   y[25]        234d475e2f79cbf05e4db6a9407d72c6
```

```
               bff7d1198b5c4d6aad2831db61274993
   y[26]       715a0182c7dc8089e32c8531deed4f74
               31c07c02195eba2ef91efb5613c37af7
   y[27]       ae0c066babc69369700e1dd26eddc0d2
               16c781d56e4ce47e3303fa73007ff7b9
   y[28]       49ef23be2aa4dbf25206fe45c20dd888
               395b2526391a724996a44156beac8082
   y[29]       12858792bf8e74cba49dee5e8812e019
               da87454bff9e847ed83db07af3137430
   y[30]       82f880a278f682c2bd0ad6887cb59f65
               2e155987d61bbf6a88d36ee93b6072e6
   y[31]       656d9ccbaae3d655852e38deb3a2dcf8
               058dc9fb6f2ab3d3b3539eb77b248a66
   y[32]       1091d05eb6e2f297774fe6053598457c
               c61908318de4b826f0fc86d4bb117d33
   y[33]       e865aa805009cc2918d9c2f840c4da43
               a703ad9f5b5806163d7161696b5a0adc
   ---------------------------------------------
   LMS type    00000005                        # LM_SHA256_M32_H5
   path[0]     d5c0d1bebb06048ed6fe2ef2c6cef305
               b3ed633941ebc8b3bec9738754cddd60
   path[1]     e1920ada52f43d055b5031cee6192520
               d6a5115514851ce7fd448d4a39fae2ab
   path[2]     2335b525f484e9b40d6a4a969394843b
               dcf6d14c48e8015e08ab92662c05c6e9
   path[3]     f90b65a7a6201689999f32bfd368e5e3
               ec9cb70ac7b8399003f175c40885081a
   path[4]     09ab3034911fe125631051df0408b394
               6b0bde790911e8978ba07dd56c73e7ee
```

Test Case 2 Private Key

```
   ---------------------------------------------
   (note: procedure in Appendix A is used)
   SEED        000102030405060708090a0b0c0d0e0f
               101112131415161718191a1b1c1d1e1f
   I           d08fabd4a2091ff0a8cb4ed834e74534
   ---------------------------------------------
   ---------------------------------------------
```

Test Case 2 Public Key

```
---------------------------------------------
HSS public key
levels      00000002
---------------------------------------------
LMS type    00000006                        # LM_SHA256_M32_H10
LMOTS type  00000003                        # LMOTS_SHA256_N32_W4
I           d08fabd4a2091ff0a8cb4ed834e74534
K           32a58885cd9ba0431235466bff9651c6
            c92124404d45fa53cf161c28f1ad5a8e
---------------------------------------------
---------------------------------------------
```

Test Case 2 Message

```
---------------------------------------------
Message     54686520656e756d65726174696f6e20 The enumeration
            696e2074686520436f6e737469747574 in the Constitut
            696f6e2c206f66206365727461696e20 ion, of certain
            7269676874732c207368616c6c206e6f rights, shall no
            7420626520636f6e7374727565642074 t be construed t
            6f2064656e79206f7220646973706172 o deny or dispar
            616765206f746865727320726574616 9 age others retai
            6e65642062792074686520706 56f706c ned by the peopl
            652e0a                           e..
---------------------------------------------
```

Test Case 2 Signature

```
---------------------------------------------
HSS signature
Nspk        00000001
sig[0]:
---------------------------------------------
LMS signature
q           00000003
---------------------------------------------
LMOTS signature
LMOTS type  00000003                        # LMOTS_SHA256_N32_W4
C           3d46bee8660f8f215d3f96408a7a64cf
            1c4da02b63a55f62c666ef5707a914ce
y[0]        0674e8cb7a55f0c48d484f31f3aa4af9
            719a74f22cf823b94431d01c926e2a76
y[1]        bb71226d279700ec81c9e95fb11a0d10
            d065279a5796e265ae17737c44eb8c59
y[2]        4508e126a9a7870bf4360820bdeb9a01
            d9693779e416828e75bddd7d8c70d50a
```

```
y[3]        0ac8ba39810909d445f44cb5bb58de73
            7e60cb4345302786ef2c6b14af212ca1
y[4]        9edeaa3bfcfe8baa6621ce88480df237
            1dd37add732c9de4ea2ce0dffa53c926
y[5]        49a18d39a50788f4652987f226a1d481
            68205df6ae7c58e049a25d4907edc1aa
y[6]        90da8aa5e5f7671773e941d805536021
            5c6b60dd35463cf2240a9c06d694e9cb
y[7]        54e7b1e1bf494d0d1a28c0d31acc7516
            1f4f485dfd3cb9578e836ec2dc722f37
y[8]        ed30872e07f2b8bd0374eb57d22c614e
            09150f6c0d8774a39a6e168211035dc5
y[9]        2988ab46eaca9ec597fb18b4936e66ef
            2f0df26e8d1e34da28cbb3af75231372
y[10]       0c7b345434f72d65314328bbb030d0f0
            f6d5e47b28ea91008fb11b05017705a8
y[11]       be3b2adb83c60a54f9d1d1b2f476f9e3
            93eb5695203d2ba6ad815e6a111ea293
y[12]       dcc21033f9453d49c8e5a6387f588b1e
            a4f706217c151e05f55a6eb7997be09d
y[13]       56a326a32f9cba1fbe1c07bb49fa04ce
            cf9df1a1b815483c75d7a27cc88ad1b1
y[14]       238e5ea986b53e087045723ce16187ed
            a22e33b2c70709e53251025abde89396
y[15]       45fc8c0693e97763928f00b2e3c75af3
            942d8ddaee81b59a6f1f67efda0ef81d
y[16]       11873b59137f67800b35e81b01563d18
            7c4a1575a1acb92d087b517a8833383f
y[17]       05d357ef4678de0c57ff9f1b2da61dfd
            e5d88318bcdde4d9061cc75c2de3cd47
y[18]       40dd7739ca3ef66f1930026f47d9ebaa
            713b07176f76f953e1c2e7f8f271a6ca
y[19]       375dbfb83d719b1635a7d8a138919579
            44b1c29bb101913e166e11bd5f34186f
y[20]       a6c0a555c9026b256a6860f4866bd6d0
            b5bf90627086c6149133f8282ce6c9b3
y[21]       622442443d5eca959d6c14ca8389d12c
            4068b503e4e3c39b635bea245d9d05a2
y[22]       558f249c9661c0427d2e489ca5b5dde2
            20a90333f4862aec793223c781997da9
y[23]       8266c12c50ea28b2c438e7a379eb106e
            ca0c7fd6006e9bf612f3ea0a454ba3bd
y[24]       b76e8027992e60de01e9094fddeb3349
            883914fb17a9621ab929d970d101e45f
y[25]       8278c14b032bcab02bd15692d21b6c5c
            204abbf077d465553bd6eda645e6c306
y[26]       5d33b10d518a61e15ed0f092c3222628
            1a29c8a0f50cde0a8c66236e29c2f310
```

```
y[27]       a375cebda1dc6bb9a1a01dae6c7aba8e
            bedc6371a7d52aacb955f83bd6e4f84d
y[28]       2949dcc198fb77c7e5cdf6040b0f84fa
            f82808bf985577f0a2acf2ec7ed7c0b0
y[29]       ae8a270e951743ff23e0b2dd12e9c3c8
            28fb5598a22461af94d568f29240ba28
y[30]       20c4591f71c088f96e095dd98beae456
            579ebbba36f6d9ca2613d1c26eee4d8c
y[31]       73217ac5962b5f3147b492e8831597fd
            89b64aa7fde82e1974d2f6779504dc21
y[32]       435eb3109350756b9fdabe1c6f368081
            bd40b27ebcb9819a75d7df8bb07bb05d
y[33]       b1bab705a4b7e37125186339464ad8fa
            aa4f052cc1272919fde3e025bb64aa8e
y[34]       0eb1fcbfcc25acb5f718ce4f7c2182fb
            393a1814b0e942490e52d3bca817b2b2
y[35]       6e90d4c9b0cc38608a6cef5eb153af08
            58acc867c9922aed43bb67d7b33acc51
y[36]       9313d28d41a5c6fe6cf3595dd5ee63f0
            a4c4065a083590b275788bee7ad875a7
y[37]       f88dd73720708c6c6c0ecf1f43bbaada
            e6f208557fdc07bd4ed91f88ce4c0de8
y[38]       42761c70c186bfdafafc444834bd3418
            be4253a71eaf41d718753ad07754ca3e
y[39]       ffd5960b033698179572142680359 9ed
            5b2b7516920efcbe32ada4bcf6c73bd2
y[40]       9e3fa152d9adeca36020fdeeee1b7395
            21d3ea8c0da497003df1513897b0f547
y[41]       94a873670b8d93bcca2ae47e64424b74
            23e1f078d9554bb5232cc6de8aae9b83
y[42]       fa5b9510beb39ccf4b4e1d9c0f19d5e1
            7f58e5b8705d9a6837a7d9bf99cd1338
y[43]       7af256a8491671f1f2f22af253bcff54
            b673199bdb7d05d81064ef05f80f0153
y[44]       d0be7919684b23da8d42ff3effdb7ca0
            985033f389181f47659138003d712b5e
y[45]       c0a614d31cc7487f52de8664916af79c
            98456b2c94a8038083db55391e347586
y[46]       2250274a1de2584fec975fb09536792c
            fbfcf6192856cc76eb5b13dc4709e2f7
y[47]       301ddff26ec1b23de2d188c999166c74
            e1e14bbc15f457cf4e471ae13dcbdd9c
y[48]       50f4d646fc6278e8fe7eb6cb5c94100f
            a870187380b777ed19d7868fd8ca7ceb
y[49]       7fa7d5cc861c5bdac98e7495eb0a2cee
            c1924ae979f44c5390ebedddc65d6ec1
y[50]       1287d978b8df064219bc5679f7d7b264
            a76ff272b2ac9f2f7cfc9fdcfb6a5142
```

```
   y[51]       8240027afd9d52a79b647c90c2709e06
               0ed70f87299dd798d68f4fadd3da6c51
   y[52]       d839f851f98f67840b964ebe73f8cec4
               1572538ec6bc131034ca2894eb736b3b
   y[53]       da93d9f5f6fa6f6c0f03ce43362b8414
               940355fb54d3dfdd03633ae108f3de3e
   y[54]       bc85a3ff51efeea3bc2cf27e1658f178
               9ee612c83d0f5fd56f7cd071930e2946
   y[55]       beeecaa04dccea9f97786001475e0294
               bc2852f62eb5d39bb9fbeef75916efe4
   y[56]       4a662ecae37ede27e9d6eadfdeb8f8b2
               b2dbccbf96fa6dbaf7321fb0e701f4d4
   y[57]       29c2f4dcd153a2742574126e5eaccc77
               686acf6e3ee48f423766e0fc466810a9
   y[58]       05ff5453ec99897b56bc55dd49b99114
               2f65043f2d744eeb935ba7f4ef23cf80
   y[59]       cc5a8a335d3619d781e7454826df720e
               ec82e06034c44699b5f0c44a8787752e
   y[60]       057fa3419b5bb0e25d30981e41cb1361
               322dba8f69931cf42fad3f3bce6ded5b
   y[61]       8bfc3d20a2148861b2afc14562ddd27f
               12897abf0685288dcc5c4982f8260268
   y[62]       46a24bf77e383c7aacab1ab692b29ed8
               c018a65f3dc2b87ff619a633c41b4fad
   y[63]       b1c78725c1f8f922f6009787b1964247
               df0136b1bc614ab575c59a16d089917b
   y[64]       d4a8b6f04d95c581279a139be09fcf6e
               98a470a0bceca191fce476f9370021cb
   y[65]       c05518a7efd35d89d8577c990a5e1996
               1ba16203c959c91829ba7497cffcbb4b
   y[66]       294546454fa5388a23a22e805a5ca35f
               956598848bda678615fec28afd5da61a
   ---------------------------------------------
   LMS type    00000006                          # LM_SHA256_M32_H10
   path[0]     b326493313053ced3876db9d23714818
               1b7173bc7d042cefb4dbe94d2e58cd21
   path[1]     a769db4657a103279ba8ef3a629ca84e
               e836172a9c50e51f45581741cf808315
   path[2]     0b491cb4ecbbabec128e7c81a46e62a6
               7b57640a0a78be1cbf7dd9d419a10cd8
   path[3]     686d16621a80816bfdb5bdc56211d72c
               a70b81f1117d129529a7570cf79cf52a
   path[4]     7028a48538ecdd3b38d3d5d62d262465
               95c4fb73a525a5ed2c30524ebb1d8cc8
   path[5]     2e0c19bc4977c6898ff95fd3d310b0ba
               e71696cef93c6a552456bf96e9d075e3
   path[6]     83bb7543c675842bafbfc7cdb88483b3
               276c29d4f0a341c2d406e40d4653b7e4
```

```
path[7]     d045851acf6a0a0ea9c710b805cced46
            35ee8c107362f0fc8d80c14d0ac49c51
path[8]     6703d26d14752f34c1c0d2c4247581c1
            8c2cf4de48e9ce949be7c888e9caebe4
path[9]     a415e291fd107d21dc1f084b11582082
            49f28f4f7c7e931ba7b3bd0d824a4570
---------------------------------------------
LMS public key
LMS type    00000005                        # LM_SHA256_M32_H5
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
I           215f83b7ccb9acbcd08db97b0d04dc2b
K           a1cd035833e0e90059603f26e07ad2aa
            d152338e7a5e5984bcd5f7bb4eba40b7
---------------------------------------------
final_signature:
---------------------------------------------
LMS signature
q           00000004
---------------------------------------------
LMOTS signature
LMOTS type  00000004                        # LMOTS_SHA256_N32_W8
C           0eb1ed54a2460d512388cad533138d24
            0534e97b1e82d33bd927d201dfc24ebb
y[0]        11b3649023696f85150b189e50c00e98
            850ac343a77b3638319c347d7310269d
y[1]        3b7714fa406b8c35b021d54d4fdada7b
            9ce5d4ba5b06719e72aaf58c5aae7aca
y[2]        057aa0e2e74e7dcfd17a0823429db629
            65b7d563c57b4cec942cc865e29c1dad
y[3]        83cac8b4d61aacc457f336e6a10b6632
            3f5887bf3523dfcadee158503bfaa89d
y[4]        c6bf59daa82afd2b5ebb2a9ca6572a60
            67cee7c327e9039b3b6ea6a1edc7fdc3
y[5]        df927aade10c1c9f2d5ff446450d2a39
            98d0f9f6202b5e07c3f97d2458c69d3c
y[6]        8190643978d7a7f4d64e97e3f1c4a08a
            7c5bc03fd55682c017e2907eab07e5bb
y[7]        2f190143475a6043d5e6d5263471f4ee
            cf6e2575fbc6ff37edfa249d6cda1a09
y[8]        f797fd5a3cd53a066700f45863f04b6c
            8a58cfd341241e002d0d2c0217472bf1
y[9]        8b636ae547c1771368d9f317835c9b0e
            f430b3df4034f6af00d0da44f4af7800
y[10]       bc7a5cf8a5abdb12dc718b559b74cab9
            090e33cc58a955300981c420c4da8ffd
y[11]       67df540890a062fe40dba8b2c1c548ce
            d22473219c534911d48ccaabfb71bc71
y[12]       862f4a24ebd376d288fd4e6fb06ed870
```

```
                5787c5fedc813cd2697e5b1aac1ced45
   y[13]        767b14ce88409eaebb601a93559aae89
                3e143d1c395bc326da821d79a9ed41dc
   y[14]        fbe549147f71c092f4f3ac522b5cc572
                90706650487bae9bb5671ecc9ccc2ce5
   y[15]        1ead87ac01985268521222fb9057df7e
                d41810b5ef0d4f7cc67368c90f573b1a
   y[16]        c2ce956c365ed38e893ce7b2fae15d36
                85a3df2fa3d4cc098fa57dd60d2c9754
   y[17]        a8ade980ad0f93f6787075c3f680a2ba
                1936a8c61d1af52ab7e21f416be09d2a
   y[18]        8d64c3d3d8582968c2839902229f85ae
                e297e717c094c8df4a23bb5db658dd37
   y[19]        7bf0f4ff3ffd8fba5e383a48574802ed
                545bbe7a6b4753533353d73706067640
   y[20]        135a7ce517279cd683039747d218647c
                86e097b0daa2872d54b8f3e508598762
   y[21]        9547b830d8118161b65079fe7bc59a99
                e9c3c7380e3e70b7138fe5d9be255150
   y[22]        2b698d09ae193972f27d40f38dea264a
                0126e637d74ae4c92a6249fa103436d3
   y[23]        eb0d4029ac712bfc7a5eacbdd7518d6d
                4fe903a5ae65527cd65bb0d4e9925ca2
   y[24]        4fd7214dc617c150544e423f450c99ce
                51ac8005d33acd74f1bed3b17b7266a4
   y[25]        a3bb86da7eba80b101e15cb79de9a207
                852cf91249ef480619ff2af8cabca831
   y[26]        25d1faa94cbb0a03a906f683b3f47a97
                c871fd513e510a7a25f283b196075778
   y[27]        496152a91c2bf9da76ebe089f4654877
                f2d586ae7149c406e663eadeb2b5c7e8
   y[28]        2429b9e8cb4834c83464f079995332e4
                b3c8f5a72bb4b8c6f74b0d45dc6c1f79
   y[29]        952c0b7420df525e37c15377b5f09843
                19c3993921e5ccd97e097592064530d3
   y[30]        3de3afad5733cbe7703c5296263f7734
                2efbf5a04755b0b3c997c4328463e84c
   y[31]        aa2de3ffdcd297baaaacd7ae646e44b5
                c0f16044df38fabd296a47b3a838a913
   y[32]        982fb2e370c078edb042c84db34ce36b
                46ccb76460a690cc86c302457dd1cde1
   y[33]        97ec8075e82b393d542075134e2a17ee
                70a5e187075d03ae3c853cff60729ba4
   ---------------------------------------------
   LMS type     00000005                        # LM_SHA256_M32_H5
   path[0]      4de1f6965bdabc676c5a4dc7c35f97f8
                2cb0e31c68d04f1dad96314ff09e6b3d
   path[1]      e96aeee300d1f68bf1bca9fc58e40323
```

```
                36cd819aaf578744e50d1357a0e42867
   path[2]      04d341aa0a337b19fe4bc43c2e79964d
                4f351089f2e0e41c7c43ae0d49e7f404
   path[3]      b0f75be80ea3af098c9752420a8ac0ea
                2bbb1f4eeba05238aef0d8ce63f0c6e5
   path[4]      e4041d95398a6f7f3e0ee97cc1591849
                d4ed236338b147abde9f51ef9fd4e1c1
```

Authors' Addresses

David McGrew
Cisco Systems
13600 Dulles Technology Drive
Herndon, VA 20171
USA

Email: mcgrew@cisco.com

Michael Curcio
Cisco Systems
7025-2 Kit Creek Road
Research Triangle Park, NC 27709-4987
USA

Email: micurcio@cisco.com

Scott Fluhrer
Cisco Systems
170 West Tasman Drive
San Jose, CA
USA

Email: sfluhrer@cisco.com

Internet Research Task Force (IRTF)
Internet-Draft
Intended status: Informational
Expires: December 16, 2017

B. Viguier
Radboud University

June 14, 2017

KangarooTwelve
draft-viguier-kangarootwelve-00

Abstract

This document defines the KangarooTwelve eXtendable Output Function (XOF), a hash function with arbitrary output length. It provides an efficient and secure hashing primitive, which is able to exploit the parallelism of the implementation in a scalable way. It uses tree hashing over a round-reduced version of SHAKE128 as underlying primitive.

This document builds up on the definitions of the permutations and of the sponge construction in [FIPS 202], and is meant to serve as a stable reference and an implementation guide.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

This document defines the KangarooTwelve eXtendable Output Function (XOF) [K12], i.e. a generalization of a hash function that can return arbitrary output length. KangarooTwelve is based on a Keccak-p permutation specified in [FIPS202] and aims at higher speed than SHAKE and SHA-3.

The SHA-3 functions process data in a serial manner and unable to optimally exploit parallelism available in modern CPU architectures. KangarooTwelve splits the input message in fragments and applies an inner hash function F on each of them separately. It then applies F again on the concatenation of the digests. It makes use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The inner hash function F is a sponge function and uses a round-reduced version of the permutation used in Keccak. Its security builds up on the scrutiny that Keccak has received since its publication [KECCAK_CRYPTANALYSIS].

1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

The following notations are used throughout the document:

‘...‘ denotes a bit-string. For example, ‘1010101‘.

A 8 bit string ‘b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7‘ is a byte represented by an integer value v following the LSB 0 convention, i.e.

```
v = sum for i=0..7 of 2^i * b_i
```

For example, ‘11100000‘ = 7. The following diagram represents the byte "07" with value 7 (decimal).

```
 Significance of Bits
MSB 7 6 5 4 3 2 1 0 LSB
   +-+-+-+-+-+-+-+-+
   |0 0 0 0 0 1 1 1|
   +-+-+-+-+-+-+-+-+
hex:   0       7
```

"..." denotes a string of bytes given in hexadecimal. For example, "0B 80", which can be also be seen as a bit-string : ‘11010000 00000001‘.

|s| denotes the length of a byte string "s". For example, |"FF FF"| = 2.

‘0^b‘ denotes the repetition of bit ‘0‘ b times. For example, ‘0^4‘ = ‘0000‘.

‘0^0‘ denotes the empty bit-string.

‘1^b‘ denotes the repetition of bit ‘1‘ b times. For example, ‘1^3‘ = ‘111‘.

"00^b" denotes the b times the repetition of byte "00". For example, "00^7" = "00 00 00 00 00 00 00".

a||b denotes the concatenation of two strings ’a’ and ’b’. For example, ‘10‘||‘01‘ = ‘1001‘

s[n:m] denotes the selection of bytes from n to m exclusive of a string ’s’. For example, for s = "A5 C6 D7", s[0:1] = "A5" and s[1:3] = "C6 D7".

2. Specifications

KangarooTwelve is an eXtendable Output Function (XOF). It takes as an input a pair of byte-strings (M, C) and a positive integer L where

M byte-string, is the Message and

C byte-string, is a Customization string and

L positive integer, the length of the output in bytes.

The Customization string serves as domain separation. It is typically a short string such as a name or an identifier (e.g. URI, ODI...)

2.1. Inner function: F

The inner function F makes use of the permutation Keccak-p[1600,n_r=12], i.e., a version of the one used in SHAKE and SHA-3 instances reduced to n_r=12 rounds and specified in FIPS 202 [FIPS202]. F is a sponge function calling this permutation, multi-rate padding pad10*1 and with a rate of 168 bytes (= 1344 bits):

```
F = Sponge[Keccak-p[1600,n_r=12], pad10*1, r=1344]
```

It follows that F has a capacity of 1600 - 1344 = 256 bits.

The sponge function F takes as an input a bit-string S and a positive integer L where

S bit-string, is the input String and

L positive integer, the Length of the output in bytes

The input string S SHOULD be represented as a pair (Sbytes, dS), where Sbytes contains only bytes and where dS is the delimited suffix representing the trailing bits.

First, let S = Sbytes || Sbits, where Sbytes contains only bytes and Sbits contains at most 7 bits. Then, convert Sbits into the delimited suffix dS by appending a bit ‘1‘ and as many bits ‘0‘ as necessary so that dS is a byte. The numerical value of dS is thus:

```
dS = 2^|Sbits| + sum for i=0..|Sbits|-1 of 2^i*Sbits_i
```

Notice that the most significant bit ‘1‘ of dS coincides with the first bit of padding in the multi-rate padding rule pad10*1. The implementation of F therefore SHOULD add dS to the state and then the second bit of padding. Appendix A.2 provides a pseudo code version.

In the table below, here are some examples of values, including those that are used in this document:

Sbits	bit-string	value (dec)	delimited Suffix (dS)
‘‘	‘10000000‘	1	"01"
‘01‘	‘01100000‘	6	"06"
‘11‘	‘11100000‘	7	"07"
‘110‘	‘11010000‘	11	"0B"

2.2. Tree hashing over F

On top of the sponge function F, KangarooTwelve uses a Sakura-compatible tree hash mode [SAKURA]. First, merge M and C to a single input string S in a reversible way. right_encode(|C|) gives the length in bytes of C as a byte-string. See Section 2.3.

```
S = M || C || right_encode( |C| )
```

Then, split S into n chunks of 8192 bytes.

```
S = S_0 || .. || S_n-1
  |S_0| = .. = |S_n-2| = 8192 bytes
  |S_n-1| <= 8192 bytes
```

From S_1 .. S_n-1, compute the 32-bytes hashes CV_0 .. CV_n-2. This computation SHOULD exploit the parallelism available on the platform in order to be optimally efficient.

```
Node_i  = S_i+1 || ‘110‘
CV_i    = F( Node_i, 32 )
```

Compute the final node: Node*.

o If |S| <= 8192 bytes, then Node* = S || ‘11‘

o Otherwise compute Node* as follow:

```
Node* = S_0 || "03 00 00 00 00 00 00 00"
Node* = Node* || CV_0
        ..
Node* = Node* ||  CV_n-2
Node* = Node* || right_encode(n-1)
Node* = Node* || "FF FF" || ‘01‘
```

Finally, KangarooTwelve output is retrieved from F(Node*).

```
KangarooTwelve( M, C, L ) = F( Node*, L )
```

For |S| > 8192 bytes, KangarooTwelve computation flow is as follow:

```
                                +--------------+
                                |      S_0     |
                                +--------------+
                                       ||
                                +--------------+
                                | ‘11‘||‘0^62‘ |
                                +--------------+
                                       ||
  +-------------------+   F   +--------------+
  |    S_1   || ‘110‘ |------>|     CV_0     |
  +-------------------+       +--------------+
                                       ||
  +-------------------+   F   +--------------+
  |    S_2   || ‘110‘ |------>|     CV_1     |
  +-------------------+       +--------------+
                                       ||
          ...                         ...
                                       ||
  +-------------------+   F   +--------------+
  |   S_n-1  || ‘110‘ |------>|    CV_n-2    |
  +-------------------+       +--------------+
                                       ||
                                +--------------+
                                |   r_e(n-1)   |
                                +--------------+
                                       ||
                                +------------------+     F
                                |  "FF FF" || ‘01‘ |---------->  output
                                +------------------+
```

We provide a pseudo code version in Appendix A.3.

2.3. right_encode(x)

The function right_encode takes as inputs a non negative integer x < 256^255 and outputs a string of bytes x_n || .. || x_0 || n where

```
x = sum from i=0..n of 256^i * x_i
```

A pseudo code version is as follow.

```
right_encode(x):
  S = 0^0

  while x > 0
      S = x % 256 || S
      x = x / 256

  S = S || length(S)

  return S
  end
```

3. Test vectors

Test vectors are based on the repetition of pattern the "00 01 .. FA" with a specific length. ptn(n) defines a string by repeating the pattern "00 01 .. FA" as many times as necessary and truncated to n bytes e.g.

```
Pattern for a length of 17 bytes:
ptn(17) =
  "00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10"
```

```
Pattern for a length of 17^2 bytes:
ptn(17^2) =
  "00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
   20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
   30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
   40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
   50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
   60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
   70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
   A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
   B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
   C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
   D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
   E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
   F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
   20 21 22 23 24 25"

KangarooTwelve(M=0^0, C=0^0, 32):
  "1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5"

KangarooTwelve(M=0^0, C=0^0, 64):
  "1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5
   42 69 C0 56 B8 C8 2E 48 27 60 38 B6 D2 92 96 6C
   C0 7A 3D 46 45 27 2E 31 FF 38 50 81 39 EB 0A 71"

KangarooTwelve(M=0^0, C=0^0, 10032), last 32 bytes:
  "E8 DC 56 36 42 F7 22 8C 84 68 4C 89 84 05 D3 A8
   34 79 91 58 C0 79 B1 28 80 27 7A 1D 28 E2 FF 6D"

KangarooTwelve(M=ptn(1 bytes), C=0^0, 32):
  "2B DA 92 45 0E 8B 14 7F 8A 7C B6 29 E7 84 A0 58
   EF CA 7C F7 D8 21 8E 02 D3 45 DF AA 65 24 4A 1F"

KangarooTwelve(M=ptn(17 bytes), C=0^0, 32):
  "6B F7 5F A2 23 91 98 DB 47 72 E3 64 78 F8 E1 9B
   0F 37 12 05 F6 A9 A9 3A 27 3F 51 DF 37 12 28 88"

KangarooTwelve(M=ptn(17^2 bytes), C=0^0, 32):
  "0C 31 5E BC DE DB F6 14 26 DE 7D CF 8F B7 25 D1
   E7 46 75 D7 F5 32 7A 50 67 F3 67 B1 08 EC B6 7C"
```

```
KangarooTwelve(M=ptn(17^3 bytes), C=0^0, 32):
  "CB 55 2E 2E C7 7D 99 10 70 1D 57 8B 45 7D DF 77
   2C 12 E3 22 E4 EE 7F E4 17 F9 2C 75 8F 0D 59 D0"

KangarooTwelve(M=ptn(17^4 bytes), C=0^0, 32):
  "87 01 04 5E 22 20 53 45 FF 4D DA 05 55 5C BB 5C
   3A F1 A7 71 C2 B8 9B AE F3 7D B4 3D 99 98 B9 FE"

KangarooTwelve(M=ptn(17^5 bytes), C=0^0, 32):
  "84 4D 61 09 33 B1 B9 96 3C BD EB 5A E3 B6 B0 5C
   C7 CB D6 7C EE DF 88 3E B6 78 A0 A8 E0 37 16 82"

KangarooTwelve(M=ptn(17^6 bytes), C=0^0, 32):
  "3C 39 07 82 A8 A4 E8 9F A6 36 7F 72 FE AA F1 32
   55 C8 D9 58 78 48 1D 3C D8 CE 85 F5 8E 88 0A F8"

KangarooTwelve(M=0^0, C=ptn(1 bytes), 32):
  "FA B6 58 DB 63 E9 4A 24 61 88 BF 7A F6 9A 13 30
   45 F4 6E E9 84 C5 6E 3C 33 28 CA AF 1A A1 A5 83"

KangarooTwelve(M=0xff, C=ptn(41 bytes), 32):
  "D8 48 C5 06 8C ED 73 6F 44 62 15 9B 98 67 FD 4C
   20 B8 08 AC C3 D5 BC 48 E0 B0 6B A0 A3 76 2E C4"

KangarooTwelve(M=0xff ff ff, C=ptn(41^2), 32):
  "C3 89 E5 00 9A E5 71 20 85 4C 2E 8C 64 67 0A C0
   13 58 CF 4C 1B AF 89 44 7A 72 42 34 DC 7C ED 74"

KangarooTwelve(M=0xff ff ff ff ff ff ff, C=ptn(41^3 bytes), 32):
  "75 D2 F8 6A 2E 64 45 66 72 6B 4F BC FC 56 57 B9
   DB CF 07 0C 7B 0D CA 06 45 0A B2 91 D7 44 3B CF"
```

4. IANA Considerations

None.

5. Security Considerations

This document is meant to serve as a stable reference and an implementation guide for the KangarooTwelve eXtendable Output Function. It makes no assertion to its security and relies on the cryptanalysis of Keccak [KECCAK_CRYPTANALYSIS].

6. References

6.1. Normative References

[FIPS202] National Institute of Standards and Technology, "FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", WWW http://dx.doi.org/10.6028/NIST.FIPS.202, August 2015.

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

6.2. Informative References

[K12] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "KangarooTwelve: fast hashing based on Keccak-p", WWW http://eprint.iacr.org/2016/770.pdf, August 2016.

[KECCAK_CRYPTANALYSIS] Keccak Team, "Summary of Third-party cryptanalysis of Keccak", WWW https://www.keccak.team/third_party.html, 2017.

[SAKURA] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "Sakura: a flexible coding for tree hashing", WWW http://eprint.iacr.org/2013/231.pdf, April 2013.

Appendix A. Pseudo code

The sub-sections of this appendix contain pseudo code definitions of KangarooTwelve.

A.1. Keccak-p[1600] over 12 rounds

```
Keccak-p_1600_12(state):
  R = "D5"

  for x from 0 to 4
    for y from 0 to 4
      lanes[x][y] = state[8*(x+5*y):8*(x+5*y)+8]

  for round from 12 to 23
    # theta
    for x from 0 to 4
      C[x] = lanes[x][0]
      C[x] ^= lanes[x][1]
      C[x] ^= lanes[x][2]
      C[x] ^= lanes[x][3]
      C[x] ^= lanes[x][4]
    for x from 0 to 4
      D[x] = C[(x+4)%5] ^ ROL64(C[(x+1)%5], 1)
    for y from 0 to 4
      for x from 0 to 4
        lanes = lanes[x][y]^D[x]

    # rho and pi
    (x, y) = (1, 0)
    current = lanes[x][y]
    for t from 0 to 23
      (x, y) = (y, (2*x+3*y)%5)
      (current, lanes[x][y]) =
          (lanes[x][y], ROL64(current, (t+1)*(t+2)/2))

    # chi
    for y from 0 to 4
      for x from 0 to 4
        T[x] = lanes[x][y]
        for x from 0 to 4
          lanes[x][y] = T[x] ^((not T[(x+1)%5]) & T[(x+2)%5])

    # iota
    for j from 0 to 6
      R = ((R << 1) ^ ((R >> 7)* "71")) % 256
      if (R & 2)
        lanes[0][0] = lanes[0][0] ^ (1 << ((1<<j)-1))
```

```
    state = 0^0
    for x from 0 to 4
      for y from 0 to 4
        state = state || lanes[x][y]

    return state
    end
```

where ROL64(x, y) is a rotation of the ’x’ 64-bit word toward the bits with higher indexes by ’y’ bits.

A.2. Inner function F

```
  F(inputBytes, Suffix, outputByteLen):
    state = "00^200"
    blockSize = 0
    offset = 0

    # === Absorb inputBytes ===
    while offset < |inputBytes|
        blockSize = min( |inputBytes| - offset, 168)
        state ^= inputBytes[offset : offset + blockSize]
        offset = offset + blockSize

        if blockSize = 168
            state = Keccak-p_1600_12(state)
            blockSize = 0

    # === Absorb Suffix ===
    state ^= "00^blockSize" || Suffix
    if (Suffix & "80") != 0 and blockSize == 167
        state = Keccak-p_1600_12(state)
    state ^= "00^167" || "80"

    state = Keccak-p_1600_12(state)

    # === Squeeze ===
    while outputByteLen > 0
        blockSize = min(outputByteLen, 168)
        outputBytes = outputBytes || state[0:blockSize]
        outputByteLen = outputByteLen - blockSize

        if outputByteLen > 0
            state = Keccak-p_1600_12(state)

    return outputBytes
    end
```

A.3. KangarooTwelve

```
KangarooTwelve(inputMessage, customString, outputByteLen):
  S = inputMessage || customString
  S = S || right_encode( |customString| )

  if |S| <= 8192
      return F(S, "07", outputByteLen)
  else
      # === Kangaroo hopping ===
      Node* = S[0:8192] || "03 00^7"
      offset = 8192
      while offset < |inputBytes|
          blockSize = min( |inputBytes| - offset, 8192)
          CV = F(inputBytes[offset : offset + blockSize], "0B", 32)
          Node* = Node* || CV
          offset = offset + blockSize

      Node* = Node* || right_encode( |S| / 8192 ) || "FF FF"
      return F(Node*, "06", outputByteLen)
  end
```

Author’s Address

Benoit Viguier
Radboud University
Toernooiveld 212
Nijmegen
The Netherlands

EMail: b.viguier@cs.ru.nl

Internet Research Task Force (IRTF)
Internet-Draft
Intended status: Informational
Expires: August 11, 2019

B. Viguier
Radboud University

February 7, 2019

KangarooTwelve
draft-viguier-kangarootwelve-04

Abstract

This document defines the KangarooTwelve eXtendable Output Function (XOF), a hash function with output of arbitrary length. It provides an efficient and secure hashing primitive, which is able to exploit the parallelism of the implementation in a scalable way. It uses tree hashing over a round-reduced version of SHAKE128 as underlying primitive.

This document builds up on the definitions of the permutations and of the sponge construction in [FIPS 202], and is meant to serve as a stable reference and an implementation guide.

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 August 11, 2019.

Table of Contents

1. Introduction

This document defines the KangarooTwelve eXtendable Output Function (XOF) [K12], i.e. a generalization of a hash function that can return an output of arbitrary length. KangarooTwelve is based on a Keccak-p permutation specified in [FIPS202] and has a higher speed than SHAKE and SHA-3.

The SHA-3 functions process data in a serial manner and are unable to optimally exploit parallelism available in modern CPU architectures. Similar to ParallelHash [SP800-185], KangarooTwelve splits the input message in fragments to exploit available parallelism. It then applies an inner hash function F on each of them separately before applying F again on the concatenation of the digests. It makes use of Sakura coding for ensuring soundness of the tree hashing mode [SAKURA]. The inner hash function F is a sponge function and uses a round-reduced version of the permutation Keccak-f used in SHA-3, making it faster than ParallelHash. Its security builds up on the scrutiny that Keccak has received since its publication [KECCAK_CRYPTANALYSIS].

With respect to [FIPS202] and [SP800-185] functions, KangarooTwelve features the following advantages:

- o Unlike SHA3-224, SHA3-256, SHA3-384, SHA3-512, KangarooTwelve has an extendable output.

- o Unlike any [FIPS202] defined function, similarly to functions defined in [SP800-185], KangarooTwelve allows the use of a customization string.

- o Unlike any [FIPS202] and [SP800-185] functions but ParallelHash, KangarooTwelve splits the input message in fragments to exploit available parallelism.

- o Unlike ParallelHash, KangarooTwelve does not have overhead when processing short messages.

- o The Keccak-f permutation in KangarooTwelve has half the number of rounds of the one used in SHA3, making it faster than any function defined in [FIPS202] and [SP800-185].

1.1. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

The following notations are used throughout the document:

`...` denotes a string of bytes given in hexadecimal. For example, `0B 80`.

|s| denotes the length of a byte string `s`. For example, |`FF FF`| = 2.

`00`^b denotes a byte string consisting of the concatenation of b bytes `00`. For example, `00`^7 = `00 00 00 00 00 00 00`.

`00`^0 denotes the empty byte-string.

a||b denotes the concatenation of two strings a and b. For example, `10`||`F1` = `10 F1`

s[n:m] denotes the selection of bytes from n to m exclusive of a string s. For example, for s = `A5 C6 D7`, s[0:1] = `A5` and s[1:3] = `C6 D7`.

s[n:] denotes the selection of bytes from n to the end of a string s. For example, for s = `A5 C6 D7`, s[0:] = `A5 C6 D7` and s[2:] = `D7`.

In the following, x and y are byte strings of equal length:

x^=y denotes x takes the value x XOR y.

x & y denotes x AND y.

In the following, x and y are integers:

x+=y denotes x takes the value x + y.

x-=y denotes x takes the value x - y.

x**y denotes x multiplied by itself y times.

2. Specifications

KangarooTwelve is an eXtendable Output Function (XOF). It takes as input two byte-strings (M, C) and a positive integer L where

M byte-string, is the Message and

C byte-string, is an OPTIONAL Customization string and

L positive integer, the number of output bytes requested.

The Customization string MAY serve as domain separation. It is typically a short string such as a name or an identifier (e.g. URI, ODI...)

By default, the Customization string is the empty string. For an API that does not support a customization string input, C MUST be the empty string.

2.1. Inner function F

The inner function F makes use of the permutation Keccak-p[1600,n_r=12], i.e., a version of the permutation Keccak-f[1600] used in SHAKE and SHA-3 instances reduced to its last n_r=12 rounds and specified in FIPS 202, sections 3.3 and 3.4 [FIPS202]. KP denotes this permutation.

F is a sponge function calling this permutation KP with a rate of 168 bytes or 1344 bits. It follows that F has a capacity of 1600 - 1344 = 256 bits or 32 bytes.

The sponge function F takes:

input byte-string, the input bytes and

outputByteLen positive integer, the Length of the output in bytes

First the message is padded with zeroes to the closest multiple of 168 bytes. Then a byte '80' is XORed to the last byte of the padded message. and the resulting string is split into a sequence of 168-byte blocks.

As defined by the sponge construction, the process operates on a state and consists of two phases.

In the absorbing phase the state is initialized to all-zero. The message blocks are XORed into the first 168 bytes of the state. Each block absorbed is followed with an application of KP to the state.

In the squeezing phase output is formed by taking the first 168 bytes of the state, repeated as many times as necessary until outputByteLen bytes are obtained, interleaved with the application of KP to the state.

This definition of the sponge construction assumes a at least one-byte-long input where the last byte is in the '01'-'7F' range. This is the case in KangarooTwelve.

A pseudo-code version is available as follows:

```
F(input, outputByteLen):
  offset = 0
  state = '00'^200

  # === Absorb complete blocks ===
  while offset < |input| - 168
      state ^= inputBytes[offset : offset + 168] || '00'^32
      state = KP(state)
      offset += 168

  # === Absorb last block and treatment of padding ===
  LastBlockLength = |input| - offset
  state ^= inputBytes[offset:] || '00'^(200-LastBlockLength)
  state ^= '00'^167 || '80' || '00'^32
  state = KP(state)

  # === Squeeze ===
  output = '00'^0
  while outputByteLen > 168
      output = output || state[0:168]
      outputByteLen -= 168
      state = KP(state)

  output = output || state[0:outputByteLen]

  return output
  end
```

2.2. Tree hashing over F

On top of the sponge function F, KangarooTwelve uses a Sakura-compatible tree hash mode [SAKURA]. First, merge M and the OPTIONAL C to a single input string S in a reversible way. length_encode(|C|) gives the length in bytes of C as a byte-string. length_encode(x) may be abbreviated as l_e(x). See Section 2.3.

```
S = M || C || length_encode( |C| )
```

Then, split S into n chunks of 8192 bytes.

```
S = S_0 || .. || S_n-1
  |S_0| = .. = |S_n-2| = 8192 bytes
  |S_n-1| <= 8192 bytes
```

From S_1 .. S_n-1, compute the 32-bytes Chaining Values CV_1 .. CV_n-1. This computation SHOULD exploit the parallelism available on the platform in order to be optimally efficient.

```
CV_i    = F( S_i||`0B`, 32 )
```

Compute the final node: FinalNode.

- o If |S| <= 8192 bytes, FinalNode = S

- o Otherwise compute FinalNode as follows:

```
FinalNode = S_0 || `03 00 00 00 00 00 00 00`
FinalNode = FinalNode || CV_1
      ..
FinalNode = FinalNode || CV_n-1
FinalNode = FinalNode || length_encode(n-1)
FinalNode = FinalNode || `FF FF`
```

Finally, KangarooTwelve output is retrieved:

- o If |S| <= 8192 bytes, from F(FinalNode||`07`, L)

```
KangarooTwelve( M, C, L ) = F( FinalNode||`07`, L )
```

- o Otherwise from F(FinalNode||`06`, L)

```
KangarooTwelve( M, C, L ) = F( FinalNode||`06`, L )
```

The following figure illustrates the computation flow of KangarooTwelve for |S| <= 8192 bytes:

```
+--------------+  F(..||`07`, L)
|      S       |----------------->  output
+--------------+
```

The following figure illustrates the computation flow of KangarooTwelve for |S| > 8192 bytes:

```
                                +--------------+
                                |     S_0      |
                                +--------------+
                                      ||
                                +--------------+
                                | '03'||'00'^7 |
                                +--------------+
                                      ||
   +---------+  F(..||'0B',32)  +--------------+
   |   S_1   |----------------->|     CV_1     |
   +---------+                  +--------------+
                                      ||
   +---------+  F(..||'0B',32)  +--------------+
   |   S_2   |----------------->|     CV_2     |
   +---------+                  +--------------+
                                      ||
             ...                      ...
                                      ||
   +---------+  F(..||'0B',32)  +--------------+
   |  S_n-1  |----------------->|    CV_n-1    |
   +---------+                  +--------------+
                                      ||
                                +--------------+
                                |  l_e( n-1 )  |
                                +--------------+
                                      ||
                                +------------+  F(..||'06', L)
                                |   'FF FF'  |----------------->  output
                                +------------+
```

We provide a pseudo code version in Appendix A.2.

In the table below are gathered the values of the domain separation bytes used by the tree hash mode:

Type	Byte
SingleNode	'07'
IntermediateNode	'0B'
FinalNode	'06'

2.3. length_encode(x)

The function length_encode takes as inputs a non negative integer x < 256**255 and outputs a string of bytes x_n-1 || .. || x_0 || n where

```
x = sum from i=0..n-1 of 256**i * x_i
```

and where n is the smallest non-negative integer such that x < 256**n. n is also the length of x_n-1 || .. || x_0.

As example, length_encode(0) = `00`, length_encode(12) = `0C 01` and length_encode(65538) = `01 00 02 03`

A pseudo code version is as follows.

```
length_encode(x):
  S = `00`^0

  while x > 0
      S = x mod 256 || S
      x = x / 256

  S = S || length(S)

  return S
  end
```

3. Test vectors

Test vectors are based on the repetition of the pattern `00 01 .. FA` with a specific length. ptn(n) defines a string by repeating the pattern `00 01 .. FA` as many times as necessary and truncated to n bytes e.g.

```
Pattern for a length of 17 bytes:
ptn(17) =
  `00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 10`
```

```
Pattern for a length of 17**2 bytes:
ptn(17**2) =
  `00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
   20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
   30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
   40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F
   50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
   60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6D 6E 6F
   70 71 72 73 74 75 76 77 78 79 7A 7B 7C 7D 7E 7F
   80 81 82 83 84 85 86 87 88 89 8A 8B 8C 8D 8E 8F
   90 91 92 93 94 95 96 97 98 99 9A 9B 9C 9D 9E 9F
   A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 AA AB AC AD AE AF
   B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 BA BB BC BD BE BF
   C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 CA CB CC CD CE CF
   D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF
   E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 EA EB EC ED EE EF
   F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 FA
   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
   10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
   20 21 22 23 24 25`

KangarooTwelve(M=`00`^0, C=`00`^0, 32):
  `1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5`

KangarooTwelve(M=`00`^0, C=`00`^0, 64):
  `1A C2 D4 50 FC 3B 42 05 D1 9D A7 BF CA 1B 37 51
   3C 08 03 57 7A C7 16 7F 06 FE 2C E1 F0 EF 39 E5
   42 69 C0 56 B8 C8 2E 48 27 60 38 B6 D2 92 96 6C
   C0 7A 3D 46 45 27 2E 31 FF 38 50 81 39 EB 0A 71`

KangarooTwelve(M=`00`^0, C=`00`^0, 10032), last 32 bytes:
  `E8 DC 56 36 42 F7 22 8C 84 68 4C 89 84 05 D3 A8
   34 79 91 58 C0 79 B1 28 80 27 7A 1D 28 E2 FF 6D`

KangarooTwelve(M=ptn(1 bytes), C=`00`^0, 32):
  `2B DA 92 45 0E 8B 14 7F 8A 7C B6 29 E7 84 A0 58
   EF CA 7C F7 D8 21 8E 02 D3 45 DF AA 65 24 4A 1F`

KangarooTwelve(M=ptn(17 bytes), C=`00`^0, 32):
  `6B F7 5F A2 23 91 98 DB 47 72 E3 64 78 F8 E1 9B
   0F 37 12 05 F6 A9 A9 3A 27 3F 51 DF 37 12 28 88`

KangarooTwelve(M=ptn(17**2 bytes), C=`00`^0, 32):
  `0C 31 5E BC DE DB F6 14 26 DE 7D CF 8F B7 25 D1
   E7 46 75 D7 F5 32 7A 50 67 F3 67 B1 08 EC B6 7C`
```

```
KangarooTwelve(M=ptn(17**3 bytes), C=`00`^0, 32):
  `CB 55 2E 2E C7 7D 99 10 70 1D 57 8B 45 7D DF 77
   2C 12 E3 22 E4 EE 7F E4 17 F9 2C 75 8F 0D 59 D0`

KangarooTwelve(M=ptn(17**4 bytes), C=`00`^0, 32):
  `87 01 04 5E 22 20 53 45 FF 4D DA 05 55 5C BB 5C
   3A F1 A7 71 C2 B8 9B AE F3 7D B4 3D 99 98 B9 FE`

KangarooTwelve(M=ptn(17**5 bytes), C=`00`^0, 32):
  `84 4D 61 09 33 B1 B9 96 3C BD EB 5A E3 B6 B0 5C
   C7 CB D6 7C EE DF 88 3E B6 78 A0 A8 E0 37 16 82`

KangarooTwelve(M=ptn(17**6 bytes), C=`00`^0, 32):
  `3C 39 07 82 A8 A4 E8 9F A6 36 7F 72 FE AA F1 32
   55 C8 D9 58 78 48 1D 3C D8 CE 85 F5 8E 88 0A F8`

KangarooTwelve(M=`00`^0, C=ptn(1 bytes), 32):
  `FA B6 58 DB 63 E9 4A 24 61 88 BF 7A F6 9A 13 30
   45 F4 6E E9 84 C5 6E 3C 33 28 CA AF 1A A1 A5 83`

KangarooTwelve(M=`FF`, C=ptn(41 bytes), 32):
  `D8 48 C5 06 8C ED 73 6F 44 62 15 9B 98 67 FD 4C
   20 B8 08 AC C3 D5 BC 48 E0 B0 6B A0 A3 76 2E C4`

KangarooTwelve(M=`FF FF FF`, C=ptn(41**2), 32):
  `C3 89 E5 00 9A E5 71 20 85 4C 2E 8C 64 67 0A C0
   13 58 CF 4C 1B AF 89 44 7A 72 42 34 DC 7C ED 74`

KangarooTwelve(M=`FF FF FF FF FF FF FF`, C=ptn(41**3 bytes), 32):
  `75 D2 F8 6A 2E 64 45 66 72 6B 4F BC FC 56 57 B9
   DB CF 07 0C 7B 0D CA 06 45 0A B2 91 D7 44 3B CF`
```

4. IANA Considerations

None.

5. Security Considerations

This document is meant to serve as a stable reference and an implementation guide for the KangarooTwelve eXtendable Output Function. It relies on the cryptanalysis of Keccak [KECCAK_CRYPTANALYSIS] and provides with the same security strength as SHAKE128, i.e., 128 bits of security against all attacks

To achieve 128-bit security strength, the output L must be chosen long enough so that there are no generic attacks that violate 128-bit security. So for 128-bit (second) preimage security the output should be at least 128 bits, for 128-bit of security against multi-

target preimage attacks with T targets the output should be at least 128+log_2(T) bits and for 128-bit collision security the output should be at least 256 bits.

Furthermore, when the output length is at least 256 bits, KangarooTwelve achieves NIST's post-quantum security level 2 [NISTPQ].

6. References

6.1. Normative References

[FIPS202] National Institute of Standards and Technology, "FIPS PUB 202 - SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", WWW http://dx.doi.org/10.6028/NIST.FIPS.202, August 2015.

[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>.

[SP800-185] National Institute of Standards and Technology, "NIST Special Publication 800-185 SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", WWW https://doi.org/10.6028/NIST.SP.800-185, December 2016.

6.2. Informative References

[K12] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "KangarooTwelve: fast hashing based on Keccak-p", WWW http://eprint.iacr.org/2016/770.pdf, August 2016.

[KECCAK_CRYPTANALYSIS] Keccak Team, "Summary of Third-party cryptanalysis of Keccak", WWW https://www.keccak.team/third_party.html, 2017.

[NISTPQ] National Institute of Standards and Technology, "Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process", WWW https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf, December 2016.

[SAKURA] Bertoni, G., Daemen, J., Peeters, M., and G. Van Assche, "Sakura: a flexible coding for tree hashing", WWW http://eprint.iacr.org/2013/231.pdf, April 2013.

[XKCP] Bertoni, G., Daemen, J., Peeters, M., Van Assche, G., and R. Van Keer, "eXtended Keccak Code Package", WWW https://github.com/XKCP/XKCP, September 2018.

Appendix A. Pseudo code

The sub-sections of this appendix contain pseudo code definitions of KangarooTwelve. A standalone Python version is also available in the Keccak Code Package [XKCP] and in [K12]

A.1. Keccak-p[1600,n_r=12]

```
KP(state):
  RC[0]  = `8B 80 00 80 00 00 00 00`
  RC[1]  = `8B 00 00 00 00 00 00 80`
  RC[2]  = `89 80 00 00 00 00 00 80`
  RC[3]  = `03 80 00 00 00 00 00 80`
  RC[4]  = `02 80 00 00 00 00 00 80`
  RC[5]  = `80 00 00 00 00 00 00 80`
  RC[6]  = `0A 80 00 00 00 00 00 00`
  RC[7]  = `0A 00 00 80 00 00 00 80`
  RC[8]  = `81 80 00 80 00 00 00 80`
  RC[9]  = `80 80 00 00 00 00 00 80`
  RC[10] = `01 00 00 80 00 00 00 00`
  RC[11] = `08 80 00 80 00 00 00 80`

  for x from 0 to 4
    for y from 0 to 4
      lanes[x][y] = state[8*(x+5*y):8*(x+5*y)+8]

  for round from 0 to 11
    # theta
    for x from 0 to 4
      C[x] = lanes[x][0]
      C[x] ^= lanes[x][1]
      C[x] ^= lanes[x][2]
      C[x] ^= lanes[x][3]
      C[x] ^= lanes[x][4]
    for x from 0 to 4
      D[x] = C[(x+4) mod 5] ^ ROL64(C[(x+1) mod 5], 1)
    for y from 0 to 4
      for x from 0 to 4
        lanes[x][y] = lanes[x][y]^D[x]

    # rho and pi
    (x, y) = (1, 0)
    current = lanes[x][y]
    for t from 0 to 23
      (x, y) = (y, (2*x+3*y) mod 5)
      (current, lanes[x][y]) =
          (lanes[x][y], ROL64(current, (t+1)*(t+2)/2))
```

```
      # chi
      for y from 0 to 4
        for x from 0 to 4
          T[x] = lanes[x][y]
        for x from 0 to 4
          lanes[x][y] = T[x] ^((not T[(x+1) mod 5]) & T[(x+2) mod 5])

      # iota
      lanes[0][0] ^= RC[round]

    state = `00`^0
    for x from 0 to 4
      for y from 0 to 4
        state = state || lanes[x][y]

    return state
    end
```

where ROL64(x, y) is a rotation of the 'x' 64-bit word toward the bits with higher indexes by 'y' positions. The 8-bytes byte-string x is interpreted as a 64-bit word in little-endian format.

A.2. KangarooTwelve

```
KangarooTwelve(inputMessage, customString, outputByteLen):
  S = inputMessage || customString
  S = S || length_encode( |customString| )

  if |S| <= 8192
      return F(S || `07`, outputByteLen)
  else
      # === Kangaroo hopping ===
      FinalNode = S[0:8192] || `03` || `00`^7
      offset = 8192
      numBlock = 0
      while offset < |S|
          blockSize = min( |S| - offset, 8192)
          CV = F(S[offset : offset + blockSize] || `0B`, 32)
          FinalNode = FinalNode || CV
          numBlock += 1
          offset   += blockSize

      FinalNode = FinalNode || length_encode( numBlock ) || `FF FF`

      return F(FinalNode || `06`, outputByteLen)
  end
```

Author's Address

Benoit Viguier
Radboud University
Toernooiveld 212
Nijmegen
The Netherlands

EMail: b.viguier@cs.ru.nl