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

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.

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:

- A collective public key A generated from the public keys of participants N.
- A subset of participants M of N who actively participate in the signature creation. The size of M is denoted by m.
- 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:

- A list of public keys A_i of all participants i in N.
- The collective public key A.
- The statement S.
- The signature sig = R || s || Z.
- 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:

- The leader multicasts a message to his direct child nodes.
- 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