Crytographic Commitments in Punchscan

Aleks Essex

July 16, 2009

Abstract

A brief specification for the Punchscan/Scantegrity cryptographic commitment protocol is presented.

1 Introduction

This document briefly outlines the cryptographic commitment scheme employed by the joint Punchscan and Scantegrity software implementation[1] as originally described by Hosp and Popoveniuc in the original Punchscan note.[2]

1.1 Notation

- $\mathrm{E}_{(k)}(x)$: denotes the encryption of x under key k using the Advanced Encryption Standard `AES-128` in *Electronic Code Book* (ECB) mode.[3]
- $\mathrm{H}(x)$: denotes the hash of x using the Secure Hash Algorithm `SHA-256`.[4]
- $z = x \mid\mid y$: denotes concatenation of byte vectors such that $z = \{z(0), \cdots, z(m+n-1)\} = \{x(0), \cdots, x(m-1), y(0), \cdots, y(n-1)\}$.
- $x_{<64>}$: denotes the base64 encoded byte array x.[5]
- `DECODE`$(x_{<64>})$: denotes the decoding of a base64 string $x_{<64>}$ into a byte array x.
- `ENCODE`(x) : denotes the encoding of a byte array x into a base64 string $x_{<64>}$.

[1] Available: `http://www.scantegrity.org/software`, `http://www.punchscan.org/software`

[2] Ben Hosp and Stefan Popoveniuc. An Introduction to Punchscan. Proceedings of the IaVoSS Workshop on Trustworthy Elections (WOTE06), 2006. Available: `http://www.punchscan.org/learnmore.php`

[3] *Federal Information Processing Standards Publication* `FIPS 197` Advanced Encryption Standard (AES).

[4] *Federal Information Processing Standards Publication* `FIPS 180-2` Secure Hash Standard (SHS).

[5] `RFC 4648`.

2 Commitment Function

The algorithm `commit` described in **Algorithm 1** creates a cryptographic commitment to a (byte vector) message m using a 128-bit sub-key (referred to by Popoveniuc as "salt with key-material" (skm)), and an election-wide constant $const$. The skm value encrypts the constant to produce an intermediate value sak (referred to by Popoveniuc as "salt and key"). The message is then passed through a pair of `SHA-256` hashes, the resulting two 256-bit (base64 encoded) values concatenated to form the cryptographic commitment to m.

Algorithm 1: Punchscan `commit`

Input : $m, skm_{<64>}, const_{<64>}$
Result: $commitment_{<64>}$

$skm = \texttt{DECODE}(skm_{<64>})$
$const = \texttt{DECODE}(const_{<64>})$
$sak = \mathrm{E}_{(skm)}(const)$
$h1 = H(m \mid\mid sak)$
$h2 = H(m \mid\mid E_{(sak)}(h1))$
Return $\texttt{ENCODE}(h1 \mid\mid h2)$

3 Protocol

The Punchscan commitment protocol is an interactive 3-pass protocol involving two entities: a prover **P** and a verifier **V**.

1. **P** sends $commitment_{<64>} = \texttt{commit}(m, skm_{<64>}, const_{<64>})$ to **V** prior to commencement of election.
2. Election concludes. A selective challenge of commitments is made using a public random-coin (not discussed here, but typically implemented by a PRNG seeded by the future closing price of a sufficiently-sized portfolio of well-traded stock indices).
3. **P** responds to a challenge to decommit $commitment_{<64>}$ by sending to **V**: $\{m', skm'_{<64>}, const'_{<64>}\}$.
4. **V** computes: $commitment'_{<64>} = \texttt{commit}(m', skm'_{<64>}, const'_{<64>})$
5. **V** concludes $\{m', skm'_{<64>}, const'_{<64>}\} = \{m, skm_{<64>}, const_{<64>}\}$ iff $commitment'_{<64>} = commitment_{<64>}$ (i.e., the commitment was verified), otherwise the protocol terminates with a fail condition.

4 Test Vector

Input:
$m =$ `30 04 03 01 02 00 03 01 00 02 00 03 01 04 02 00 01`
$skm_{<64>} =$ `dWvJjTDof3YHWyOYvkIFoA==`
$const_{<64>} =$ `UHJpbmNldG9uRWxlY3Rpbw==`

Intermediate:
$skm =$ `75 6b c9 8d 30 e8 7f 76 07 5b 23 98 be 42 05 a0`
$const =$ `50 72 69 6e 63 65 74 6f 6e 45 6c 65 63 74 69 6f`
$sak =$ `c6 5c 8f 8b f7 bd 57 d5 86 53 e6 60 2f fb a3 ac`

$h1 =$ `11 a6 1e d8 14 e8 ab 9d bd bb 35 7b 45 ed af 31` \\
`d9 6a 87 7f 16 c7 7b 23 6d cb e7 13 fe ea 89 60`
$h2 =$ `ba 6d ed 72 b4 f1 b3 40 3b 0e a2 d0 14 c7 68 e4`\\
`3a 95 53 85 0a 3d ce af 91 47 00 c8 40 6c 6b 8c`

Output:
$commitment_{<64>} =$ `EaYe2BToq529uzV7Re2vMdlqh38Wx3sjbcvnE/7qiWC6`\\
`be1ytPGzQDsOotAUx2jkOpVThQo9zq+RRwDIQGxrjA==`

A Example Implementation – C#

```
/////////////////////////////////////////////////////
// Punchscan Commitment Function - C#
// Copyright Aleks Essex, 2006
/////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Security.Cryptography;

namespace punchscanAudit
{
    class cryptoSuite
    {
        public static byte[] Encrypt(byte[] plainTextBytes, byte[] keyBytes, byte[] initVectorBytes)
        {
            // Create uninitialized Rijndael encryption object.
            RijndaelManaged key = new RijndaelManaged();

            key.Mode = CipherMode.ECB;
            key.Padding = PaddingMode.None;

            ICryptoTransform encryptor = key.CreateEncryptor(keyBytes, initVectorBytes);

            // Define memory stream which will be used to hold encrypted data.
            MemoryStream memoryStream = new MemoryStream();

            // Define cryptographic stream (always use Write mode for encryption).
            CryptoStream cryptoStream = new CryptoStream(memoryStream,
                                                         encryptor,
                                                         CryptoStreamMode.Write);
            // Start encrypting.
            cryptoStream.Write(plainTextBytes, 0, plainTextBytes.Length);

            // Finish encrypting.
            cryptoStream.FlushFinalBlock();

            // Convert our encrypted data from a memory stream into a byte array.
            byte[] cipherTextBytes = memoryStream.ToArray();

            // Close both streams.
            memoryStream.Close();
            cryptoStream.Close();

            // Return encrypted string.
            return cipherTextBytes;
        }


        /////////////////////////////////////////////////////////////
        public static byte[] hash(byte[] plainTextBytes) {

            SHA256Managed hashValue = new SHA256Managed();
            byte[] hashBytes = hashValue.ComputeHash(plainTextBytes);
            return hashBytes;
        }


        /////////////////////////////////////////////////////////////
        public static byte[] Base64Decode(string str)
        {
            byte[] decbuff = Convert.FromBase64String(str);
            return decbuff;
        }

        /////////////////////////////////////////////////////////////
        public static string commit(byte[] messageBytes, string skm_base64, string electionConstant_base64){

            //Convert parameters from base64 ASCII encoding to byte array
            byte[] skm = cryptoSuite.Base64Decode(skm_base64);
            byte[] electionConstant = cryptoSuite.Base64Decode(electionConstant_base64);

            //Creates a zero-vector to be used when an IV is required
            byte[] iv = new byte[skm.Length];

            //Compute salt and key (sak)
```

```
            byte[] sak = cryptoSuite.Encrypt(electionConstant, skm, iv);

            //Concatenate messageBytes || sak
            byte[] messageBytes_and_sak = new byte[messageBytes.Length + sak.Length];
            for (int i = 0; i < messageBytes.Length; i++)
                messageBytes_and_sak[i] = messageBytes[i];
            for (int i = 0; i < sak.Length; i++)
                messageBytes_and_sak[messageBytes.Length + i] = sak[i];

            //Compute h1
            byte[] h1 = cryptoSuite.hash(messageBytes_and_sak);

            //Encrypt E_sak( h1 )
            byte[] h1Encypted_sak = cryptoSuite.Encrypt(h1, sak, iv);

            //Concatenate messageBytes || h1Encrypted
            byte[] messageBytes_and_h1Encrypted = new byte[messageBytes.Length + h1.Length];
            for (int i = 0; i < messageBytes.Length; i++)
                messageBytes_and_h1Encrypted[i] = messageBytes[i];
            for (int i = 0; i < h1Encypted_sak.Length; i++)
                messageBytes_and_h1Encrypted[messageBytes.Length + i] = h1Encypted_sak[i];

            //compute h2
            byte[] h2 = cryptoSuite.hash(messageBytes_and_h1Encrypted);

            //concatenate h1 || h2
            byte[] result = new byte[h1.Length + h2.Length];
            for (int i = 0; i < h1.Length; i++)
                result[i] = h1[i];
            for (int i = 0; i < h2.Length; i++)
                result[h1.Length + i] = h2[i];

            return Convert.ToBase64String(result);

        }//commit
    }//cryptosuite
}//auditapp
```