

The VP1 Protocol for Voice Privacy Devices

Version 1.2

Eric A. Blossom
Communication Security Corporation

Revised: August 7, 1997

1 Introduction

This document specifies the protocol used in Communication Security Corporation's line of voice privacy devices. The protocol is designed to be robust, easy to implement, and applicable to both synchronous communication channels, and with minor modifications, to packetized networks.

Intended Audience

This paper is targeted at implementors and evaluators of cryptographic systems. It is intended that there be enough information here to construct a system that interoperates with Communication Security Corporation's line of voice privacy devices.

A fairly advanced knowledge of cryptography and data communications is assumed. There are several good textbooks on cryptography that provide the needed background. Try *Applied Cryptography* [5], *Cryptography Theory and Practice* [6], or the somewhat dated *Cryptography and Data Security* [1].

2 Overview

The protocol is asymmetric. The two roles are called the *initiator* and the *responder*.

The basic objective is to agree upon a key establishment scheme, generate a unique session key, and agree upon a mutually satisfactory *profile* which specifies the traffic encryption algorithm and mode to be used as well as the speech compression method and framing. Once the key and profile have been agreed upon, arbitrary data and control payloads may be encrypted and passed in both directions.

The initiator is in charge of the key establishment protocol, and causes a set of three messages and replies to be exchanged. The initiator sends a message

and waits for a reply. If no reply is received within a reasonable time, the initiator resends the message.

2.1 Cryptographic Overview

A unique session key is generated for each conversation using the Diffie-Hellman exponential key exchange [6, 5]¹. The Diffie-Hellman public parameters, g , the generator, and p , the modulus, are specified by the *scheme* that is selected during the first phase of the key establishment protocol.

The Diffie-Hellman key exchange has two important properties for our application. Foremost is that no advanced exchange of secret keying material is required. This means that each telephone privacy device may be identical, requires no tamper-resistant storage for secrets, yet can generate a shared secret key, unique to each conversation, with any device or system running this protocol. In addition, the key exchange has the property of “perfect forward secrecy”, which means that the generated key is used only once and then is destroyed. In the event that the key used in a particular conversation is determined by an adversary, that key is of no use to the adversary for decrypting either past or future conversations.

Once a shared secret has been determined using Diffie-Hellman, a portion of that secret is used as keying material for the traffic encryption algorithm selected by the *profile*. When available, 168-bit 3-key Triple-DES is used (See section 3.3.1).

The *man-in-the-middle attack* [5, pages 48–49] is defended against by deriving a *key ID* from the public exponentials g^x and g^y and displaying this to the user for voice verification over the secure channel. If the key IDs on both sides match, there is a high degree of assurance that no man-in-the-middle attack is underway.

3 Details

3.1 Low Level Packet Framing

The underlying communications medium is assumed to provide a synchronous bit stream. Packets are prefixed with a 31 bit barker code to establish packet framing. Following this is a fixed length header of 192 bits that is the result of performing an 8-way interleave on the rate 1/2 Golay forward error corrected header, `pkt_header.t`. The Golay error correction code is described in Appendix A.

The Barker code consists of the 31 bit pattern

MSB	0 0 0 1 1 0 1 1 1 0 1 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 1 1 1 1 1	LSB
-----	---	-----

¹US Patent No. 4,200,770, Licensed by Cylink Corporation.

where the least significant bit is the first bit transmitted. It should be considered detected whenever it is received with 1 or fewer bits in error.

```

/*
 * This defines the format of packets exchanged across the raw
 * synchronous channel.
 *
 * Packets are prefixed with a 31 bit barker code to establish packet
 * framing. Following this is a fixed length header of 192 bits that
 * is the result of performing an 8-way interleave on the rate 1/2
 * Golay(24,12) forward error corrected header, pkt_header_t.
 *
 * <barker code> <INTERLEAVE(FEC(pkt_header_t))> <payload>
 */

#define BARKER_CODE_LENGTH      4
#define PKT_HEADER_WIRE_LENGTH 24      /* octets */

#define MI_SIZE 8                /* 8 octets (64 bits) */

/*
 * this header is 12 octets == 96 bits --> 192 bits when rate 1/2 FEC'd
 */
typedef struct {
    unsigned short    payload_fmt;
    unsigned char     mi[MI_SIZE];
    unsigned short    fcs;          /* CCITT 16-bit Frame Check Sequence
                                     of payload_fmt + mi */
} pkt_header_t;

/*
 * payload format is broken up like this:
 *
 * 15                                     0
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 * | E | F | R | S | 0 | 0 |           Payload Length           |
 * +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
 *
 * E = payload is encrypted
 * F = payload is forward error corrected
 * R = realtime data; payloads are streaming.
 *   (contiguous payloads of same length, without intervening headers)
 * S = Resync request
 * 0 = reserved
 */

```

```

#define PF_LENGTH_MASK 0x03ff
#define PF_LENGTH(x) ((x) & PF_LENGTH_MASK)
#define PF_ENCRYPTED 0x8000
#define PF_FEC 0x4000
#define PF_REALTIME 0x2000 /* else control data */
#define PF_RESYNC_RQST 0x1000

/*
 * the length field refers to the actual number of octets of information
 * in the payload, not the length of the payload on the wire.
 * If FEC is *not* being used, then they are the same. In either case,
 * the length of the payload on the wire is given by
 * PAYLOAD_WIRE_LENGTH (payload_fmt).
 */

inline static int
PAYLOAD_WIRE_LENGTH (unsigned short payload_fmt)
{
    if ((payload_fmt & PF_FEC) == 0)
        return PF_LENGTH (payload_fmt);

    /* partition into 24 octet blocks */
    return (((PF_LENGTH (payload_fmt) + 23) / 24) * 48;
}

```

3.2 Protocol Packets and Flow

The overall flow of packets during the key establishment protocol is shown below:

Initiator -----		Responder -----
-->	offer schemes	-->
<--	pick scheme commit g^x offer profiles	<--
-->	commit g^y pick profile	-->
<--	send g^x	<--
verify g^x vs commit g^x		

```

-->          send g^y          -->

compute shared secret          verify g^y vs commit g^y

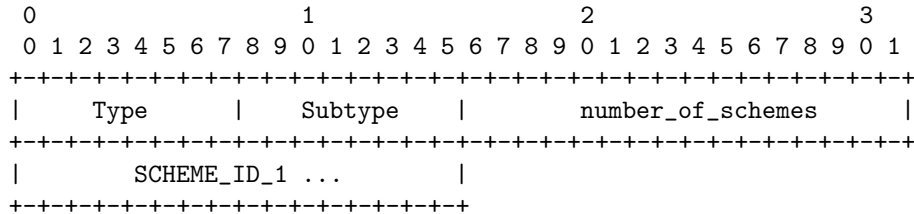
<--          ack g^y          <--

                                compute shared secret
    
```

In all cases, the initiator sends a message and waits for a reply from the responder. If no reply is received within a reasonable period of time, the initiator retransmits the message. Packets received with bad CRCs are ignored. Multi-octet fields are transmitted in “network standard order”, most significant octet first. For actual values of symbolic constants used below, see Appendix B.

3.2.1 Phase 1 / Initiator

The initiator begins the key exchange by sending a message which offers a list of *schemes* for the responder to select from. Each scheme specifies the type of key exchange algorithm to use, as well as any public parameters. At this time, all supported key exchange algorithms use the classic modular exponentiation form of Diffie-Hellman. They differ in the modulus and recommended generator used.



Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_MESSAGE_1

Number_of_schemes 16 bits, integer number of 16-bit SCHEME_ID's that follow.

3.2.2 Phase 1 / Responder

When the responder receives **Message_1** he examines the offered list of schemes for an acceptable value. If he fails to find one, he sends a **Reject** message (See 3.2.7) and terminates the negotiation.

Having found an acceptable scheme, the responder generates a random private integer exponent x , such that $0 < x < p-1$ and computes the integer public value $pv_R = g^x \text{ mod } p$ where g and p are the public parameters associated with the selected scheme.

A value representing a “commitment” to the public value pv_R is generated according to the method specified for the chosen scheme. The Secure Hash Algorithm (SHA-1) [?] is currently specified for all schemes.

To generate a commitment to pv_R using SHA-1, the integer pv_R is converted to an octet string M which is then hashed using SHA-1.

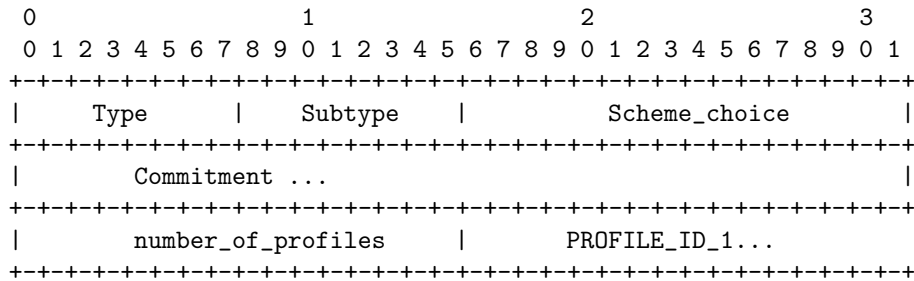
The integer public value pv_R shall be converted to an octet string M of length k . The octet string M shall satisfy

$$pv_R = \sum_{i=1}^k 2^{8(k-i)} M_i$$

where M_1, \dots, M_k are the octets of M from first to last.

In other words, the first octet of M has the most significance in the integer and the last octet of M has the least significance.

Finally, the responder constructs a Message_1_Reply message containing the selected SCHEME_ID, commitment, and a list of supported *profiles*. Each profile defines a configuration that specifies the encryption algorithm and mode to be used to encrypt the traffic, the speech compression algorithm to be used, and the framing used. The resulting message is then transmitted to the initiator.



Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_MESSAGE_1_REPLY

Scheme_choice 16 bits, value determines interpretation of Commitment

Commitment Variable length field. 16-bit length in octets, followed by that many octets.

Number_of_profiles 16 bits, integer number of 16-bit PROFILE_ID's that follow.

3.2.3 Phase 2 / Initiator

When the initiator receives a **Message_1_Reply**, he checks to ensure that the selected scheme is valid and that there is an offered profile that is acceptable. If not, he sends a **Reject** message and terminates the negotiation.

If the message is valid, he stores the commitment for later comparison with the actual $pv_R = g^x \text{ mod } p$ received.

The initiator generates a random private integer exponent y , such that $0 < y < p - 1$ and computes the integer public value $pv_I = g^y \text{ mod } p$ where g and p are the public parameters associated with the selected scheme.

A value representing a “commitment” to the public value pv_I is generated according to the method specified for the chosen scheme. The Secure Hash Algorithm (SHA-1) is currently specified for all schemes.

To generate a commitment to pv_I using SHA-1, the integer pv_I is converted to an octet string M which is then hashed using SHA-1.

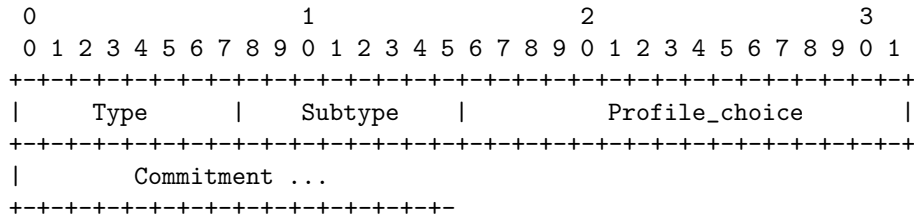
The integer public value pv_I shall be converted to an octet string M of length k . The octet string M shall satisfy

$$pv_I = \sum_{i=1}^k 2^{8(k-i)} M_i$$

where M_1, \dots, M_k are the octets of M from first to last.

In other words, the first octet of M has the most significance in the integer and the last octet of M has the least significance.

The initiator then builds and sends a **Message_2** containing the selected profile, and the commitment to pv_I , $\text{SHA-1}(M)$.



Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_MESSAGE_2

Profile_choice 16 bits, which profile was selected (defines crypto and speech coder)

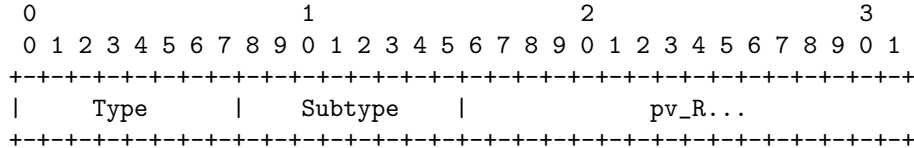
Commitment Variable length field. 16-bit length in octets, followed by that many octets.

3.2.4 Phase 2 / Responder

When the responder receives **Message_2**, the profile specified is checked for validity. If invalid, a **Reject** message is sent and the negotiation is terminated.

The commitment is saved for later use in verifying pv_I .

The responder then builds and sends a **Message_2_Reply** containing the responder's public value $pv_R = g^x \bmod p$.



Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_MESSAGE_2_REPLY

pv_R Variable length unsigned integer. 16-bit length in octets, followed by that many octets.

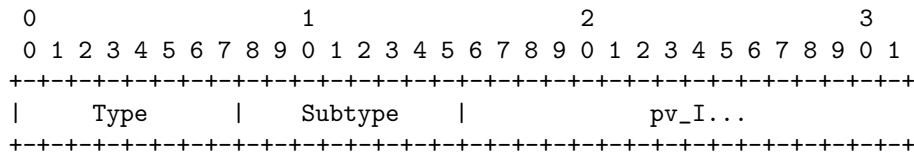
3.2.5 Phase 3 / Initiator

The initiator enters the third phase of the negotiation upon receiving a **Message_2_Reply**. The received $pv_R = g^x \bmod p$ is hashed and compared with the previously received commitment. If they fail to match, a **Reject** message is sent and the negotiation terminated.

If they match, the initiator builds and sends a **Message_3** containing pv_I , and then computes the integer shared secret ss as follows:

$$ss = (pv_R)^y \bmod p$$

Note that y is the initiator's secret exponent and p is given by the selected scheme.



Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_MESSAGE_3

pv_I Variable length unsigned integer. 16-bit length in octets, followed by that many octets.

3.2.6 Phase 3 / Responder

When the responder receives a `Message_3` the public value pv_I is hashed and compared to the previously received commitment. If they fail to match, a `Reject` message is sent and the negotiation is terminated.

If the hash matches the commitment, the responder calculates the integer shared secret ss as follows:

$$ss = (pv_I)^x \text{ mod } p$$

Note that pv_I is $g^y \text{ mod } p$. Hence the responder computes

$$ss = (g^y)^x \text{ mod } p$$

Likewise, the initiator has computed

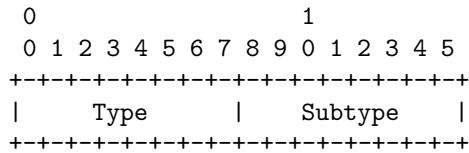
$$ss = (pv_R)^y \text{ mod } p$$

and since $pv_R = g^x \text{ mod } p$, the initiator's value is

$$ss = (g^x)^y \text{ mod } p$$

Therefore, since $(g^x)^y \equiv (g^y)^x$, both parties compute the same value for ss .

As a final step, after computing the shared secret, the responder builds and sends a `Message_3_Reply` to complete the negotiation. At this time, both sides may begin encrypting data and/or control packets using the shared secret ss and the encryption algorithm specified in the negotiated profile.

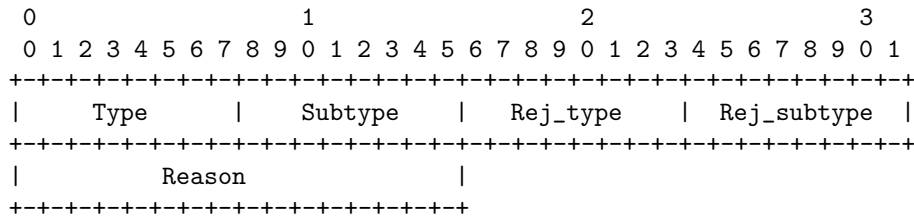


Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_MESSAGE_3_REPLY

3.2.7 Other Messages

There are three other message types that are occasionally used.



Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_REJECT

Rej_type 8 bits, the Type field of the rejected message.

Rej_subtype 8 bits, the Subtype field of the rejected message.

Reason 16 bits. Why the message was rejected.

```

0                               1                               2
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+++++
|   Type   |   Subtype   |   Why   |
+++++

```

Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_GO_CLEAR

Why 8 bits. Reason for going clear.

```

0                               1
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+++++
|   Type   |   Subtype   |
+++++

```

Type 8 bits, value = CATFISH_1

Subtype 8 bits, value = ST_GO_CLEAR_REPLY

3.3 Mapping Shared Secrets to Traffic Keys

The key exchange algorithm results in the creation of an integer shared secret *ss* that is between 512 and 2048-bits long, depending on the *scheme* agreed to at the beginning of the negotiation. This section explains how that shared secret is converted to a session key for encrypting voice and data traffic. The method used depends on the encryption specified in the negotiated *profile*.

3.3.1 Triple-DES

Let ss be the integer shared secret that results from running the key exchange algorithm. Then,

$$K1 = (ss/2^{128}) \bmod 2^{64}$$

$$K2 = (ss/2^{64}) \bmod 2^{64}$$

$$K3 = ss \bmod 2^{64}$$

$$ciphertext = E_{K3}(D_{K2}(E_{K1}(plaintext)))$$

$$plaintext = D_{K1}(E_{K2}(D_{K3}(ciphertext)))$$

Where E_{Ki} denotes DES [3] encryption with key Ki and D_{Ki} denotes DES decryption with key Ki .

3.4 Diffie's Counter Mode Variant

Proposed Counter Mode for Triple DES

Whitfield Diffie

11 October 1995

The following describes a counter-driven mode for triple-DES. The cost of generating the keystream is three DES encryptions plus the cost of incrementing the counter. The mode does not appear to be vulnerable to a meet in the middle attack for any reasonable amount of traffic.

The sender generates an initialization vector and initializes a counter:

- (1) Decrypt the initialization vector in ECB mode to get an initial value for the counter.
(Regardless of the size of the initialization vector, the decryption guarantees that the initial counter value will be a uniformly distributed 64 bit quantity.)
- (2) Decrypt the initial value in ECB mode and if the result is even add 1 to it. This is the increment.
(Like the initial counter value, this quantity will be uniformly distributed, except for the low order bit, which has been forced to be 1.)

The sender is now prepared to encrypt traffic:

- (3) Add the increment to the counter modulo 2^{64} .
(Since the increment has been forced to be odd, the counter will always go through all 2^{64} states before repeating.)
- (4) ECB encrypt the counter to get a block of keystream.

- (5) Exclusive-OR the keystream block bitwise with a block of plaintext to produce a block of ciphertext.

and repeat steps 3 through 5 as needed.

For the decryptor, the process is the same except that the initialization vector is typically received as part of the message rather than “generated” and step 5 is replaced by step:

- (5') Exclusive-OR the keystream block bitwise with a block of ciphertext to produce a block of plaintext.

Note that the initialization vector must — at least with high probability, be non-repeating — but there is no apparent need for it to be unpredictable, because it will be decrypted under the secret key before being used. This means that the initialization vector may either be produced with a random number generator or may simply be a non-repeating counter, a clock, or the number of bits, bytes, blocks, or packets exchanged so far.

3.5 Derivation of the Key ID Displayed to the User

The *key ID* is derived from the public exponentials that are exchanged during the Diffie-Hellman key exchange, and provides a means of ensuring that both end-points of the communication are using the same keying material.

The key ID is computed as follows. Let $a = \min(pv_I, pv_R)$ and $b = \max(pv_I, pv_R)$. Convert a and b to octet strings A and B . Let H be the result of hashing the concatenation of A and B using SHA-1. The key ID is the first 6 digits that result from converting the first 32-bit word of H to hexadecimal.

Once the key exchange is complete, the key ID is displayed to the user. Users should then verify that their displayed values are identical by speaking over the encrypted voice channel.

In the event that the key IDs fail to match, the most likely explanation is that there is a sophisticated man-in-the-middle attack underway. Your conversation is not secure. Go clear, and attempt to reestablish secure mode.

3.6 Payload Format When Using GSM

The GSM 06.10 Full rate speech transcoder [7] specifies a detailed mapping between input blocks of 160 speech samples in 13-bit uniform PCM format to encoded blocks of 260 bits and from encoded blocks of 260 bits to output blocks of 160 reconstructed speech samples. The sampling rate is 8000 samples/s leading to an average bit rate for the encoded stream of 13,000 bits/s. The coding scheme is the so-called Regular Pulse Excitation — Long Term prediction — Linear Predictive Coder, referred to as RPE-LTP.

A block of 160 speech samples at 8000 samples/s spans 20 ms. Using a V.32bis modem at 14,400 bits/s data rate, there are $14,400 - 13,000 = 1400$

bits/s available for framing and other uses. This may be evenly divided up into 28 “extra bits” per frame.

The encoded speech is packed into frames of 288 bits. The final bit in each frame is the “winking bit” which starts at 0 and alternates in each subsequent frame. The winking bit is used to check for loss of synchronization.

The mapping of the 260 bits of GSM information into the 288 bits of available channel is currently undergoing a change. FEC will be added to the perceptually most significant bits, and the resulting code words will be distributed throughout the frame to provide resistance to burst errors.

A Golay Rate 1/2 Forward Error Correction

A.1 Forward Error Correction

The FEC method requires the receiver to detect and automatically correct errors in a received block of information. The number of errors the receiver can detect and correct depends on the coding method. The information bits (k) are separated into blocks that contain both information bits and code bits. The length of the block, including the information and code bits, is (n). The code is described as (n,k) , where n is the length of the block and k is the number of information bits in the block.

A.2 Golay Code

The Golay code is a linear, block, perfect, and cyclic (23,12) code capable of correcting any combination of three or fewer errors in a block of 23 digits. The generator polynomial for this code is

$$g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$$

where $g(x)$ is a factor of $x^{23} + 1$.

A.3 Half-rate Golay Code

The half-rate Golay code (24,12) is formed by adding a 0 fill-bit to the Golay (23,12) code. The (24,12) code is preferable to the (23,12) code because it has a code rate of exactly one-half. This code rate simplifies system timing.

A.4 Implementation

The Golay code may be implemented in either hardware or software. A software implementation typically uses a generator matrix and conversion table. A good introduction to error correcting codes is [2]. For a more theoretical treatment see [4].

B protocol.h

```

/* --C--
*****
*
* File:          protocol.h
* Description:   constants for protocol fields.
* Author:       Eric Blossom
* Created:      Sat Jun 22 18:25:06 1996
* Modified:     Tue Oct 15 23:26:03 1996 (eric) eb@comsec.com
* Language:     C
* Package:     N/A
* Status:      Experimental (Do Not Distribute)
*
* (C) Copyright 1996, Communication Security Corp., all rights reserved.
*
*****
*/

/*
* =====
* Message types (8 bits)
* =====
*/
#define MT_CATFISH_1          0x8b

/*
* =====
* Message subtypes (8 bits)
* =====
*/
#define ST_ECHO                0x00
#define ST_ECHO_REPLY         0x01
#define ST_MESSAGE_1          0x02
#define ST_MESSAGE_1_REPLY    0x03
#define ST_MESSAGE_2          0x04
#define ST_MESSAGE_2_REPLY    0x05
#define ST_MESSAGE_3          0x06
#define ST_MESSAGE_3_REPLY    0x07
#define ST_GO_CLEAR           0x08
#define ST_GO_CLEAR_REPLY     0x09
#define ST_REJECT              0x0a

#define ST_LAST_USED_VALUE    ST_REJECT

/*

```

```

* =====
* Scheme ID's (16 bits)
*
* These define the key exchange protocol and parameters used.
* =====
*/

/*
* 1024 bit Diffie-Hellman, using Z_p. Commits use SHA-1.
*
* [from draft-simpson-photuris-11.txt, June 1996]
*
* (1024-2)
*   A 1024-bit strong prime (p), expressed in hex:
*
*       97f6 4261 cab5 05dd 2828 e13f 1d68 b6d3
*       dbd0 f313 047f 40e8 56da 58cb 13b8 a1bf
*       2b78 3a4c 6d59 d5f9 2afc 6cff 3d69 3f78
*       b23d 4f31 60a9 502e 3efa f7ab 5e1a d5a6
*
*       5e55 4313 828d a83b 9ff2 d941 dee9 5689
*       fada ea09 36ad df19 71fe 635b 20af 4703
*       6460 3c2d e059 f54b 650a d8fa 0cf7 0121
*       c747 99d7 5871 32be 9b99 9bb9 b787 e8ab
*
*   The recommended generator (g) for this prime is 2.
*
*   This prime modulus was randomly generated by a freely available
*   program written by Phil Karn, verified using the
*   mpz_probab_prime() function Miller-Rabin test in the Gnu Math
*   Package (GMP) version 1.3.2; and also verified with GMP on
*   other platforms by Wei Dai and Frank A Stevenson, as well as
*   independently developed test libraries by Eric Young (Miller-
*   Rabin test), and Rich Schroepel (complete Elliptic Curve
*   test).
*
*   Currently estimated to provide 80 (pessimistic) through 98
*   (optimistic) bit-equivalents of cryptographic strength. Expo-
*   nent lengths of 160 to 256 bits (or more) are recommended.
*/
#define SID_DH_MODEXP_1024_A    0

/*
* 2048 bit Diffie-Hellman, using Z_p. Commits use SHA-1.
*
*   A 2048-bit strong prime (p), expressed in hex:

```

```

*
*      f488 33ba b45d c12c 4022 ad67 de65 bc20
*      35b6 7f8f 59c8 5ae8 c33d 4fd9 dde2 42e9
*      a530 b391 965a 7e09 6796 fa99 bd6d 4ce6
*      2e43 6cc7 cd2b 5ba4 521d 42d5 1827 0c29
*      ae69 69f7 c9e5 d5c0 230e 9afb cbed 5abd
*      0f02 f1e1 264d 130e bfa9 7f77 82eb 2122
*      1b25 d78d 2b13 7c0a b077 40c1 103d e884
*      887e e868 9a29 d216 daf3 992f f7ee 810b
*      c63c 7c00 f274 16c4 9b8e 31bd a4a6 3bbf
*      9f96 ed16 ecbb 4535 6217 21a6 abe1 f901
*      54ce af57 7bb4 6c59 b3a4 f2a0 223f 8389
*      2c51 fe42 22f5 ee03 15af c163 4352 c433
*      2268 4b74 388a 46cd 48d7 41f3 55b5 bca6
*      755d 06cf 3846 814d dad1 5476 e8f6 bd4f
*      01e0 7fb1 cfd6 0ac3 f21d 241d 5743 4f54
*      26d1 f449 ae21 30ce 6724 27a6 6f20 715b
*
*      The recommended generator (g) for this prime is 2.
*/
#define SID_DH_MODEXP_2048_A    1

/*
* 512 bit Diffie-Hellman, using Z_p. Commits use SHA-1.
*
*      A 512-bit strong prime (p), expressed in hex:
*
*      8d34 d0fb 66a5 68ae d30c 99ac 6aa3 45cb
*      393a 9134 a9e1 659d 6658 f6ef cb69 13b4
*      0088 2d0e 732f 96de 1709 6504 f36c 90a6
*      6e99 9706 ada3 7c5d 933c bb6c 4fa9 61cb
*
*      The recommended generator (g) for this prime is 2.
*
* NOTE!!! This modulus is too small for any reasonable
* level of security. Use this only when required because
* of export restrictions.
*/
#define SID_DH_MODEXP_512_A    2

/*
* =====
* Profile ID's (16 bits)
*
* These define the speech encoder, decoder, framing and crypto.

```

```
* This gives us a simple way to negotiate something today, but
* still leaving room for future enhancements.
* =====
*/

/*
* Synchronous communication at 14400, full rate GSM encoder, expanded
* to 288 bits / frame. Crypto uses triple DES in Diffie's modified
* counter mode.
*/
#define PID_SYNC_GSM_14400_3DES_COUNTER      0

/*
* Synchronous communication at 14400, full rate GSM encoder, expanded
* to 288 bits / frame. Crypto uses triple DES in Diffie's 96 bit
* shift register mode.
*/
#define PID_SYNC_GSM_14400_3DES_96_BIT_SHIFT  1

/*
* Synchronous communication at 14400, full rate GSM encoder, expanded
* to 288 bits / frame. Crypto uses single DES with 40-bit effective
* key length in Diffie's modified counter mode.
*
* NOTE!!! This profile is not recommended for general use.
* Use only when required by export restrictions.
*/
#define PID_SYNC_GSM_14400_40_BIT_DES_COUNTER  2

/*
* =====
* Reject Reasons (16 bits)
* =====
*/

#define RR_BAD_TYPE          0
#define RR_BAD_SUBTYPE      1
#define RR_NO_SCHEME        2
#define RR_NO_PROFILE       3
#define RR_BAD_COMMIT       4
#define RR_BAD_MSG_FORMAT   5
```

References

- [1] Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.
- [2] Shu Lin and Daniel J. Costello, Jr. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, 1983.
- [3] National Bureau of Standards. *Data Encryption Standard*. U.S. Department of Commerce, January 1977. FIPS PUB 46-1.
- [4] W. Wesley Peterson and E. J. Weldon, Jr. *Error-Correcting Codes*. MIT Press, second edition, 1972.
- [5] Bruce Schneier. *Applied Cryptography*. Wiley, second edition, 1996.
- [6] Douglas R. Stinson. *Cryptography Theory and Practice*. CRC Press, 1995.
- [7] ETSI TC-SMG. European digital cellular telecommunications system (phase 2); full rate speech transcoding (gsm 06.10). European Telecommunication Standard ETS 300 580-2, European Telecommunications Standards Institute, +33 92 94 42 00, September 1994. GSM 06.10.