ДОДАТОК А

Слайди презентації

Рисунок А.1 – Слайд 1

Рисунок А.2 – Слайд 2

Content in three columns

1990-ті

У 1994 році Пітер Шор відкрив поліномно-часовий квантовий алгоритм для цілочисельної факторизації, чого почали серйозно проводитися дослідження побудови великомасштабних квантових комп'ютерів

2019

У січні 2019-го року на виставці CES був представлений перший комерційний квантовий компютер. У другій половині 2019-го року планується відкрити Quantum Computation Center, що відкриє доступ до IBM Q й дозволить розвивати сферу квантових обчислень

Перспективи

Припускається, що квантовий комп'ютер, здатний в лічині години атакувати RSA-2048, може бути побудований до 2030 року, що вже є серйозною довгостроковою загрозою для криптосистем.

Рисунок А.3 – Слайд 3

Вплив квантових комп'ютерів

Алгоритм	Тип	Призначення алгоритму	Квантова загроза
AES	Симетричний ключ	Шифрування	Необхідно збільшити розмір ключа
SHA-2, SHA-3	-	Хеш-функція	Необхідно збільшити розмір вхідного повідомлення
RSA	Відкритий ключ	Цифровий підпис	Ненадійний
ECDSA, ECDH (еліптичні криві)	Відкритий ключ	Цифровий підпис	Ненадійний
DSA (скінчене поле, поле Галуа)	Відкритий ключ	Цифровий підпис	Ненадійний

Рисунок А.4 – Слайд 4

Математичні моделі, стійкі до квантових атак

Алгебраїчні решітки

Принцип побудови асиметричних алгоритмів шифрування з використанням задач теорії решіток, тобто задач оптимізації на дискретних адитивних підгрупах, заданих на множині R^n

MQ-криптографія (мультиваріативне квадратичне перетворення)

Асиметричні алгоритми, побудовані на вирішення рівнянь, що базуються на багатовимірних поліномах над кінцевим полем F

HB-криптографія

Механіз ЕП грунтується на використанні стандартної криптографічної хеш-функції H

CB-криптографія

Криптографічне перетворення з використанням збиткових кодів

На основі ізогеній еліптичних кривих

Криптографічна система базується на властивостях суперсингулярних. еліптичних кривих і графіків суперсингулярної ізогенії

Рисунок А.5 – Слайд 5

Потенційні постквантові алгоритми

У 2016 році NIST США (National Institute of Standards and Technology) ініціював початок стандартизації постквантових алгоритмів. У січні 2019 року було оголошено 9 алгоритмів, що пройшли до другого раунду й є потенційними постквантовими алгоритмами:

- CRYTAL-DILITHIUM (базується на алгебраїчних решітках)
- FALCON (базується на алгебраїчних решітках)
- qTesla (базується на алгебраїчних решітках)
- GeMess (MQ-криптографія)
- LUOV (MQ-криптографія)
- MQDSS (MQ-криптографія)
- Rainbow (MQ-криптографія)
- Picnic (HB-криптографія)
- SPHINKS+ (HB-криптографія)

Рисунок А.6 – Слайд 6

Порівняння постквантових алгоритмів

Схема підпису	Безпека	Захищенність від timing attack	Цикли підписання	Цикли перевірки підпису	Розмір публічного ключа, Bytes	Розмір підпису, Bytes
Dilithium	125	Так	789	209	1472	2701
FALCON	>> 128	Ні	-	-	1792	1200
qTESLA	98	Так	143402	19284	12582912	2444
GeMSS	128	Так	1497	15	83100	61
LUOV	128	Так	659000	290000	34100	421
MQDSS	128	Так	8510	5752	72	40952
Rainbow	128	Так	68	22	145500	48
Picnic	128	Так	1034	194	64	195458
SPHINCS+	128	Так	51636	1451	1056	41000

Рисунок А.7 – Слайд 7

Нормалізовані значення критеріїв альтернатив

Схема підпису	Цикли підписання	Цикли перевірки підпису	Розмір публічного ключа, Bytes	Розмір підпису, Bytes
Dilithium	0.9989	0.9993	0.9903	0.9864
GeMSS	0.9978	1.0000	0.4291	0.9999
LUOV	0.0000	0.0000	0.7660	0.9981
MQDSS	0.9872	0.9802	0.9999	0.7907
Rainbow	1.0000	0.9999	0.0000	1.0000
Picnic	0.9985	0.9994	1.0000	0.0000
SPHINCS+	0.9217	0.9950	0.9932	0.7904

Рисунок А.8 – Слайд 8

Результат багатокритеріального аналізу

У результаті багатокритеріального аналізу, під час якого було нормалізовано критерії, з використанням лінійної адитивної згортки з ваговими коефіцієнтами було визначено, що Dilithium є найкращим з постквантових алгоритмів підпису.

Схема підпису	Згортка критеріїв
Dilithium	0.9757
GeMSS	0.8879
LUOV	0.2823
MQDSS	0.9357
Rainbow	0.8200
Picnic	0.8193
SPHINCS+	0.9179

Рисунок А.9 – Слайд 9

Dilithium

Схема роботи базується на підході “Fiat-Shamir with Aborts”

Основне конструктивне удосконалення алгоритму у порівнянні з класичною Схемою, полягає в тому, що розмір відкритого ключа зменшується приблизно в 2.5 рази за рахунок додання додаткових 100 байт до підпису.

Усунена жорстка детермінована природа цифрового підпису.

Є чотири рівні безпеки. В різних рівнях безпеки використовується більше/менше операцій над кільцем та більше/менше операцій XOF

Алгоритм було релізовано на мові Golang

Рисунок А.10 – Слайд 10

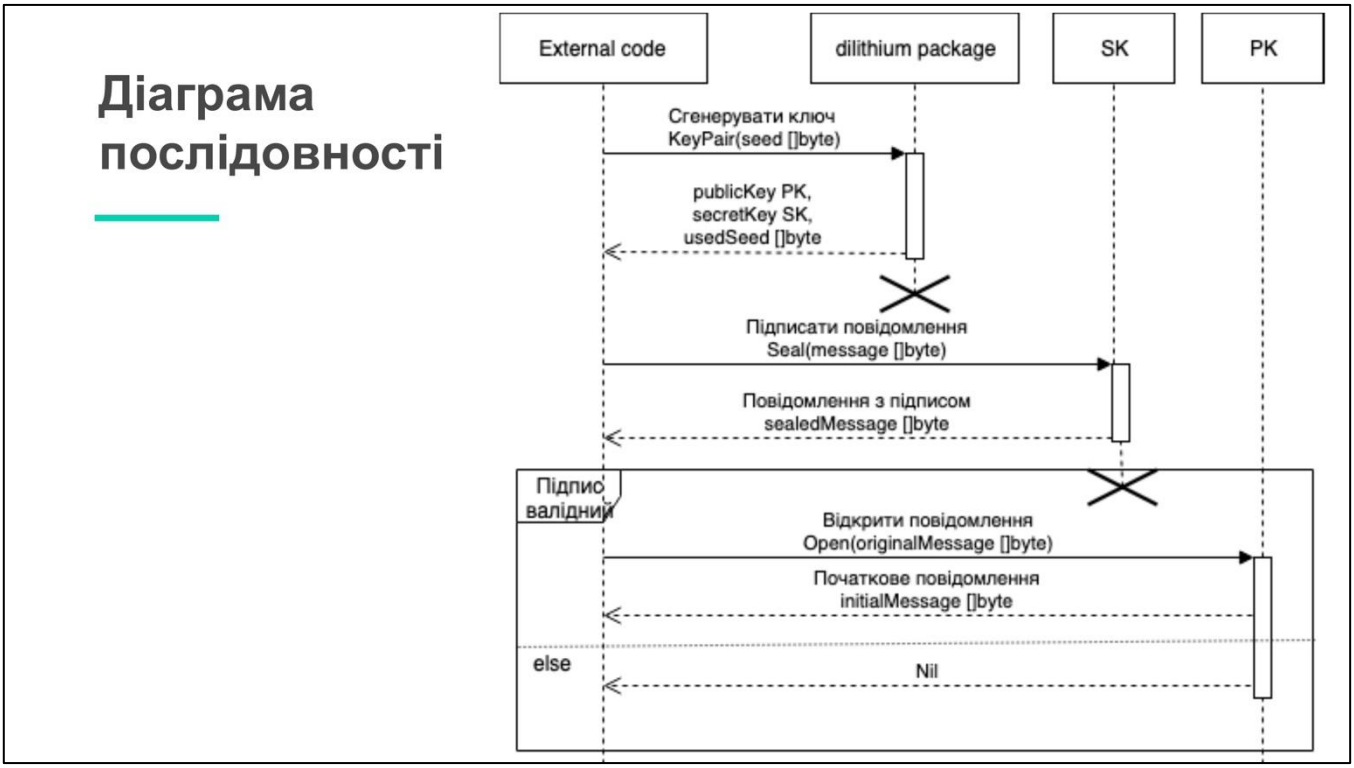

Рисунок А.11 – Слайд 11

Рисунок А.12 – Слайд 12

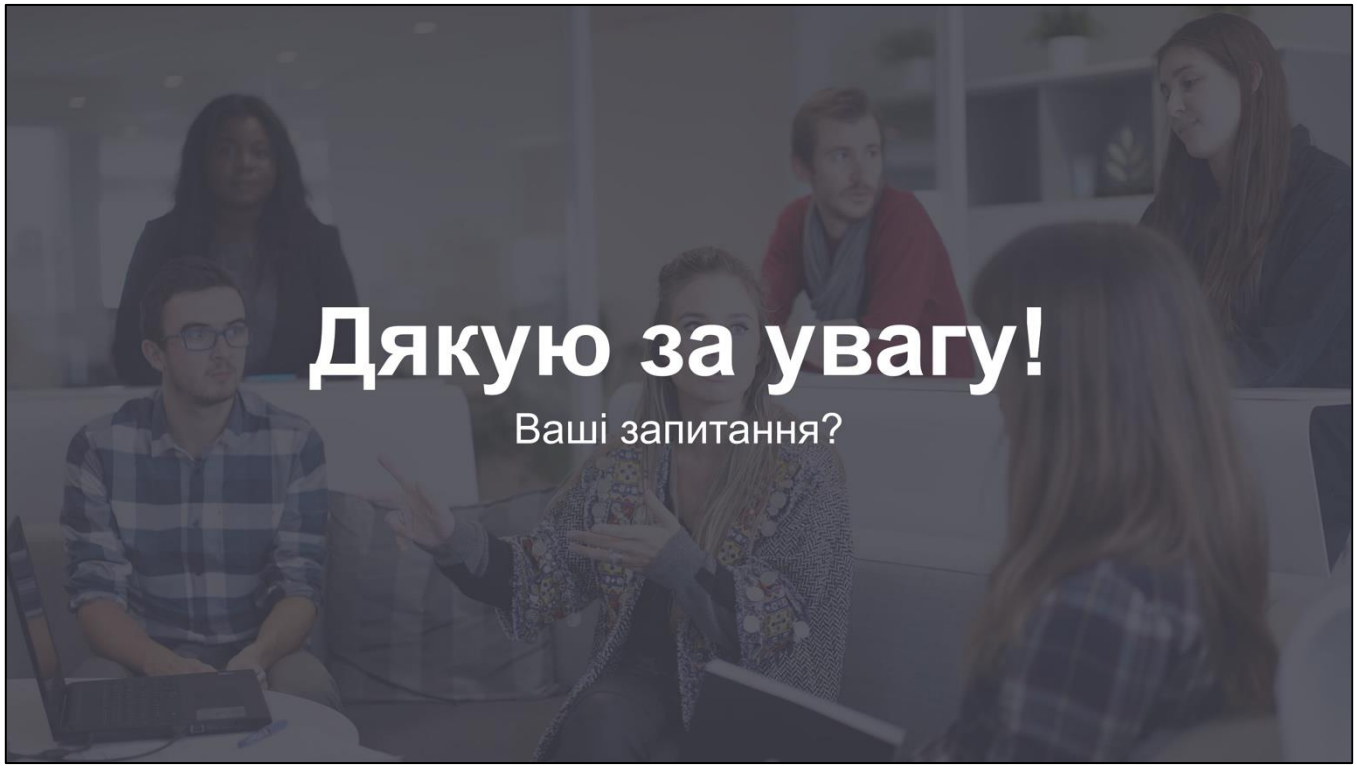

Рисунок А.13– Слайд 13

ДОДАТОК Б

Лістинг коду

```
package diploma_dilithium


import "crypto/aes"

import "archive/zip"

import "net/http"

import "testing"

import "encoding/hex"

import "io/ioutil"

import "io"

import "strings"

import "bytes"

import "bufio"

import "fmt"

import "path"


var GOLDEN_ZIP = "https://pq-crystals.org/dilithium/data/dilithium-
submission-nist-updated.zip"

var GOLDEN_KAT = fmt.Sprintf("PQCsignKAT_%d.rsp", SK_SIZE_PACKED)


func TestParams(t *testing.T) {

     t.Logf("Current params yield: %d public key size, %d signature
size\n", PK_SIZE_PACKED, SIG_SIZE_PACKED)

}


func TestSignVerifyDetached(t *testing.T) {

     pk, sk, _ := KeyPair(nil)

     msg := []byte("hello")

     sig := sk.Sign(msg)

     t.Logf("detached sig size %d (%d bytes saved)\n", len(sig),
SIG_SIZE_PACKED-len(sig))

     if !pk.Verify(msg, sig) {
```

```
            t.Fatal("basic detached signing failed")
        }
}

func TestVerify_ShouldPass_IfSignIsValid(t *testing.T) {
        pk, sk, _ := KeyPair(nil)
        msg := []byte("hello")
        sealed := sk.Seal(msg)
        if bytes.Compare(pk.Open(sealed), msg) != 0 {
            t.Fatal("basic signing failed")
        }
}

func TestSeal_ShouldReturnNil_WhenSignIsInvalid(t *testing.T) {
        pk, sk, _ := KeyPair(nil)
        msg := []byte("hello")
        sealed := sk.Seal(msg)
        sealed[5] ^= 1
        sealed[6] ^= 1
        sealed[7] ^= 1
        if bytes.Compare(pk.Open(sealed), msg) == 0 {
            t.Fatal("broken signature accepted")
        }
}

func TestBadSignDetached(t *testing.T) {
        pk, sk, _ := KeyPair(nil)
        msg := []byte("hello")
        sig := sk.Sign(msg)
        sig[5] ^= 1
        sig[6] ^= 1
        sig[7] ^= 1

        if pk.Verify(msg, sig) {
```

```
            t.Fatal("broken detached signature accepted")
        }
}

func TestSeed(t *testing.T) {
        pk, sk, seed := KeyPair(nil)
        pk2, sk2, _ := KeyPair(seed)

        if bytes.Compare(pk.Bytes()[:], pk2.Bytes()[:]) != 0 {
            t.Fatal("pk mismatch")
        }
        if bytes.Compare(sk.Bytes()[:], sk2.Bytes()[:]) != 0 {
            t.Fatal("sk mismatch")
        }
}

// input 48 bytes seed, output 32 bytes seed
func seedNIST(t *testing.T, seed []byte) []byte {
        var key [32]byte
        var V [16]byte
        var temp [48]byte
        var res [32]byte

        incV := func() {
            for j := 15; j >= 0; j-- {
                if V[j] == 0xff {
                    V[j] = 0
                } else {
                    V[j]++
                    break
                }
            }
        }
```

```
	state, _ := aes.NewCipher(key[:])
	for i := byte(0); i < 3; i++ {
		incV()
		state.Encrypt(temp[16*i:], V[:])
	}
	for i := 0; i < 48; i++ {
		temp[i] ^= seed[i]
	}
	copy(key[:], temp[:32])
	copy(V[:], temp[32:])
	state, _ = aes.NewCipher(key[:])
	incV()
	state.Encrypt(res[0:16], V[:])
	incV()
	state.Encrypt(res[16:32], V[:])
	return res[:]
}

func TestPQCSignKAT(t *testing.T) {
	os.Mkdir("testdata", 0755)
	cached := "testdata/" + path.Base(GOLDEN_ZIP)
	zipfile, err := zip.OpenReader(cached)
	if err != nil {
		t.Logf("Retrieving golden KAT zip from %s", GOLDEN_ZIP)
		resp, _ := http.Get(GOLDEN_ZIP)
		defer resp.Body.Close()
		body, _ := ioutil.ReadAll(resp.Body)
		ioutil.WriteFile(cached, body, 0644)
		zipfile, _ = zip.OpenReader(cached)
	}

	var katfile io.ReadCloser
	gotkat := false
	for _, f := range zipfile.File {
```

```
        if strings.HasSuffix(f.Name, GOLDEN_KAT) {
                katfile, _ = f.Open()
                gotkat = true
                break
        }
}

if !gotkat {
        t.Fatal("failed to get golden KAT data")
}

r := bufio.NewReader(katfile)

smlen := 0
mlen := 0
var opk, pk PK
var osk, sk SK
var msg []byte
for {
        line, err := r.ReadString('\n')
        if err != nil {
                break
        }
        fields := strings.Split(line, " ")
        if len(fields) != 3 {
                continue
        }
        val := strings.TrimSpace(fields[2])
        bval := []byte(val)
        hval := make([]byte, hex.DecodedLen(len(bval)))
        hex.Decode(hval, bval)
        switch fields[0] {
        case "smlen":
                smlen, _ = strconv.Atoi(val)
```

```
            case "mlen":
                mlen, _ = strconv.Atoi(val)
            case "msg":
                if len(hval) != mlen {
                    t.Fatal("mlen != len(msg)")
                }
                msg = hval
                _ = msg
            case "seed":
                {
                    //var seed [32]byte
                    if len(hval) != 48 {
                        t.Fatal("expected 48 byte seed")
                    }
                    opk, osk, _ = KeyPair(seedNIST(t, hval))
                }
            case "sk":
                if len(hval) != SK_SIZE_PACKED {
                    t.Fatal("sk size mismatch")
                }
                copy(sk.Bytes()[:], hval)
                if bytes.Compare(osk.Bytes()[:], sk.Bytes()[:]) != 0
{
                    t.Fatal("sk mismatch")
                }
            case "pk":
                {
                    if len(hval) != PK_SIZE_PACKED {
                        t.Fatal("pk size mismatch")
                    }
                    copy(pk.Bytes()[:], hval)
                    if bytes.Compare(opk.Bytes()[:], pk.Bytes()[:])
!= 0 {
                        t.Fatal("pk mismatch")
```

```
                    }
               }
          case "sm":
               if len(hval) != smlen {
                    t.Fatal("smlen != len(sm)")
               }
               if bytes.Compare(osk.Seal(msg), hval) != 0 {
                    t.Fatal("signed data mismatch")
               }
               if pk.Open(hval) == nil {
                    t.Fatal("failed to validate")
               }
          }
     }
}

func TestSignVerify1000(t *testing.T) {
     var sk [SK_SIZE_PACKED]byte
     var pk [PK_SIZE_PACKED]byte
     msg := []byte("hello there")
     for i := 0; i < 1000; i++ {
          crypto_sign_keypair(nil, &pk, &sk)
          msg[0] = byte(i)
          msg[1] = byte(i >> 8)
          msg[2] = byte(i >> 16)
          msg[3] = byte(i >> 24)
          signed := crypto_sign(msg, &sk)
          if crypto_sign_open(signed, &pk) == nil {
               t.Fatal("failed to verify")
          }
     }
}

func BenchmarkKeyPair(b *testing.B) {
```

```
	var seed [32]byte
	for i := 0; i < b.N; i++ {
		KeyPair(seed[:])
	}
}
func BenchmarkSign(b *testing.B) {
	var msg [32]byte
	_, sk, _ := KeyPair(nil)
	b.ResetTimer()
	for i := 0; i < b.N; i++ {
		msg[0] = byte(i)
		msg[1] = byte(i >> 8)
		msg[2] = byte(i >> 16)
		sk.Sign(msg[:])
	}
}
func BenchmarkVerify(b *testing.B) {
	var msg [32]byte
	sigs := make([][]byte, b.N)
	pk, sk, _ := KeyPair(nil)
	for i := 0; i < b.N; i++ {
		msg[0] = byte(i)
		msg[1] = byte(i >> 8)
		msg[2] = byte(i >> 16)
		sigs[i] = sk.Sign(msg[:])
	}
	b.ResetTimer()
	for i := 0; i < b.N; i++ {
		msg[0] = byte(i)
		msg[1] = byte(i >> 8)
		msg[2] = byte(i >> 16)

		if !pk.Verify(msg[:], sigs[i]) {
			b.Fatal("verify failed")
```

```
          }
     }
}
type SK [SK_SIZE_PACKED]byte

type PK [PK_SIZE_PACKED]byte

func KeyPair(seed []byte) (publicKey PK, secretKey SK, usedSeed
[]byte) {
     usedSeed = crypto_sign_keypair(seed, publicKey.Bytes(),
secretKey.Bytes())
     return
}

func (k *SK) Bytes() (rawSecretKeyBytes *[SK_SIZE_PACKED]byte) {
     return (*[SK_SIZE_PACKED]byte)(k)
}

func (k *SK) Seal(message []byte) (sealedMessage []byte) {
     return crypto_sign(message, k.Bytes())
}

func (k *SK) Sign(message []byte) (signature []byte) {
     var sig [SIG_SIZE_PACKED]byte
     return crypto_sign_detached(&sig, message, k.Bytes())
}

func (k *PK) Bytes() (rawPublicKeyBytes *[PK_SIZE_PACKED]byte) {
     return (*[PK_SIZE_PACKED]byte)(k)
}

func (k *PK) Open(sealedMessage []byte) (originalMessage []byte) {
     return crypto_sign_open(sealedMessage, k.Bytes())
}
```

```
func (k *PK) Verify(message []byte, signature []byte) bool {
     return crypto_verify_detached(signature, message, k.Bytes())
}
func ntt(p *[N]uint32) {
     var count, start, j, k uint
     var zeta, t uint32

     k = 1
     for count = 128; count > 0; count >>= 1 {
          for start = 0; start < N; start = j + count {
               zeta = zetas[k]
               k++
               for j = start; j < start+count; j++ {
                    t = montgomery_reduce(uint64(zeta) *
uint64(p[j+count]))
                    p[j+count] = p[j] + 2*Q - t
                    p[j] = p[j] + t
               }
          }
     }
}
func crypto_verify_detached(sm []byte, m []byte, pk
*[PK_SIZE_PACKED]byte) bool {
     var z polyvecl
     var h polyveck
     var c poly
     if !unpack_sig_detached(&z, &h, &c, sm) {
          return false
     }
     return crypto_verify_raw(&z, &h, &c, m, pk)
}
func crypto_sign_keypair(seed []byte, pk *[PK_SIZE_PACKED]byte, sk
*[SK_SIZE_PACKED]byte) []byte {
     var tr [CRHBYTES]byte // CRHBYTES - hash size = 48
```

```
var rho, rhoprime, key [SEEDBYTES]byte // 32
var s2, t, t1, t0 polyveck
var s1, s1hat polyvecl
var mat [K]polyvecl
var nonce uint16

if seed == nil {
     seed = make([]byte, SEEDBYTES)
     rand.Read(seed)
}
state := sha3.NewShake256()
state.Write(seed)
state.Read(rho[:])
state.Read(rhoprime[:])
state.Read(key[:])

expand_mat(&mat, &rho)

for i := 0; i < L; i++ {
     if nonce > 255 {
          panic("bad mode")
     }
     poly_uniform_eta(&s1.vec[i], &rhoprime, byte(nonce))
     nonce++
}
for i := 0; i < K; i++ {
     if nonce > 255 {
          panic("bad mode")
     }
     poly_uniform_eta(&s2.vec[i], &rhoprime, byte(nonce))
     nonce++
}

s1hat = s1
```

```
	polyvecl_ntt(&s1hat)
	for i := 0; i < K; i++ {
		polyvecl_pointwise_acc_invmontgomery(&t.vec[i], &mat[i],
&s1hat)
		poly_invntt_montgomery(&t.vec[i])
	}

	polyveck_add(&t, &t, &s2)

	polyveck_freeze(&t)
	polyveck_power2round(&t1, &t0, &t)
	pack_pk(pk, &rho, &t1)

	sha3.ShakeSum256(tr[:], pk[:])
	pack_sk(sk, &rho, &key, &tr, &s1, &s2, &t0)

	return seed
}
func crypto_sign(msg []byte, sk *[SK_SIZE_PACKED]byte) []byte {
	var sig [SIG_SIZE_PACKED]byte
	crypto_sign_attached(&sig, msg, sk)
	return append(sig[:], msg...)
}
const (
	// Settings for the current strength mode "recommended".
	K        = 5
	L        = 4
	ETA      = 5
	SETABITS = 4
	BETA     = 275
	OMEGA    = 96
)
const (
	// Public key buffer size. Always exact this size.
```

```
    PK_SIZE_PACKED  = (SEEDBYTES + K*POLT1_SIZE_PACKED)


    // Private key buffer size. Always exact this size.
    SK_SIZE_PACKED  = (2*SEEDBYTES + (L+K)*POLETA_SIZE_PACKED +
CRHBYTES + K*POLT0_SIZE_PACKED)


    // Signature size. Attached signatures are exactly this big.
    // Detached signatures tend to be a bit smaller, this being the
upper bound.
    SIG_SIZE_PACKED = (L*POLZ_SIZE_PACKED + (OMEGA + K) + (N/8 +
8))


    // These are all private for porting reasons, disregard the
upper case.
    SHAKE256_RATE = 136
    SHAKE128_RATE = 168
    SEEDBYTES     = 32
    CRHBYTES      = 48         // hash of public key
    N             = 256
    Q             = 8380417
    QINV          = 4236238847 // -q^(-1) mod 2^32
    MONT          = 4193792    // 2^32 % Q
    QBITS         = 23
    ROOT_OF_UNITY = 1753
    D             = 14
    GAMMA1        = ((Q - 1) / 16)
    GAMMA2        = (GAMMA1 / 2)
    ALPHA         = (2 * GAMMA2)


    // Polynomial sizes
    POL_SIZE_PACKED    = ((N * QBITS) / 8)
    POLT1_SIZE_PACKED  = ((N * (QBITS - D)) / 8)
    POLT0_SIZE_PACKED  = ((N * D) / 8)
    POLETA_SIZE_PACKED = ((N * SETABITS) / 8)
    POLZ_SIZE_PACKED   = ((N * (QBITS - 3)) / 8)
```

```
    POLW1_SIZE_PACKED  = ((N * 4) / 8)

    POLVECK_SIZE_PACKED = (K * POL_SIZE_PACKED)
    POLVECL_SIZE_PACKED = (L * POL_SIZE_PACKED)
)

var zetas = [N]uint32{
    0, 25847, 5771523, 7861508, 237124, 7602457, 7504169, 466468,
    1826347, 2353451, 8021166, 6288512, 3119733, 5495562, 3111497,
2680103,
    2725464, 1024112, 7300517, 3585928, 7830929, 7260833, 2619752,
6271868,
    6262231, 4520680, 6980856, 5102745, 1757237, 8360995, 4010497,
280005,
    2706023, 95776, 3077325, 3530437, 6718724, 4788269, 5842901,
3915439,
    4519302, 5336701, 3574422, 5512770, 3539968, 8079950, 2348700,
7841118,
    6681150, 6736599, 3505694, 4558682, 3507263, 6239768, 6779997,
3699596,
    811944, 531354, 954230, 3881043, 3900724, 5823537, 2071892,
5582638,
    4450022, 6851714, 4702672, 5339162, 6927966, 3475950, 2176455,
6795196,
    7122806, 1939314, 4296819, 7380215, 5190273, 5223087, 4747489,
126922,
    3412210, 7396998, 2147896, 2715295, 5412772, 4686924, 7969390,
5903370,
    7709315, 7151892, 8357436, 7072248, 7998430, 1349076, 1852771,
6949987,
    5037034, 264944, 508951, 3097992, 44288, 7280319, 904516,
3958618,
    4656075, 8371839, 1653064, 5130689, 2389356, 8169440, 759969,
7063561,
    189548, 4827145, 3159746, 6529015, 5971092, 8202977, 1315589,
1341330,
    1285669, 6795489, 7567685, 6940675, 5361315, 4499357, 4751448,
3839961,
    2091667, 3407706, 2316500, 3817976, 5037939, 2244091, 5933984,
4817955,
```

```
    266997, 2434439, 7144689, 3513181, 4860065, 4621053, 7183191,
5187039,

    900702, 1859098, 909542, 819034, 495491, 6767243, 8337157,
7857917,

    7725090, 5257975, 2031748, 3207046, 4823422, 7855319, 7611795,
4784579,

    342297, 286988, 5942594, 4108315, 3437287, 5038140, 1735879,
203044,

    2842341, 2691481, 5790267, 1265009, 4055324, 1247620, 2486353,
1595974,

    4613401, 1250494, 2635921, 4832145, 5386378, 1869119, 1903435,
7329447,

    7047359, 1237275, 5062207, 6950192, 7929317, 1312455, 3306115,
6417775,

    7100756, 1917081, 5834105, 7005614, 1500165, 777191, 2235880,
3406031,

    7838005, 5548557, 6709241, 6533464, 5796124, 4656147, 594136,
4603424,

    6366809, 2432395, 2454455, 8215696, 1957272, 3369112, 185531,
7173032,

    5196991, 162844, 1616392, 3014001, 810149, 1652634, 4686184,
6581310,

    5341501, 3523897, 3866901, 269760, 2213111, 7404533, 1717735,
472078,

    7953734, 1723600, 6577327, 1910376, 6712985, 7276084, 8119771,
4546524,

    5441381, 6144432, 7959518, 6094090, 183443, 7403526, 1612842,
4834730,

    7826001, 3919660, 8332111, 7018208, 3937738, 1400424, 7534263,
1976782,

}


var zetas_inv = [N]uint32{

    6403635, 846154, 6979993, 4442679, 1362209, 48306, 4460757,
554416,

    3545687, 6767575, 976891, 8196974, 2286327, 420899, 2235985,
2939036,

    3833893, 260646, 1104333, 1667432, 6470041, 1803090, 6656817,
426683,

    7908339, 6662682, 975884, 6167306, 8110657, 4513516, 4856520,
3038916,
```

```
    1799107, 3694233, 6727783, 7570268, 5366416, 6764025, 8217573,
3183426,

    1207385, 8194886, 5011305, 6423145, 164721, 5925962, 5948022,
2013608,

    3776993, 7786281, 3724270, 2584293, 1846953, 1671176, 2831860,
542412,

    4974386, 6144537, 7603226, 6880252, 1374803, 2546312, 6463336,
1279661,

    1962642, 5074302, 7067962, 451100, 1430225, 3318210, 7143142,
1333058,

    1050970, 6476982, 6511298, 2994039, 3548272, 5744496, 7129923,
3767016,

    6784443, 5894064, 7132797, 4325093, 7115408, 2590150, 5688936,
5538076,

    8177373, 6644538, 3342277, 4943130, 4272102, 2437823, 8093429,
8038120,

    3595838, 768622, 525098, 3556995, 5173371, 6348669, 3122442,
655327,

    522500, 43260, 1613174, 7884926, 7561383, 7470875,

0,

}
```