МЕТОДИ РЕАЛІЗАЦІЇ КРИПТОГРАФІЧНИХ МЕХАНІЗМІВ
ЛАБОРАТОРНА РОБОТА №1
“Вибір та реалізація базових фреймворків та бібліотек”.

Недождій Максим, Буржимський Ростислав

ФІ-42мн

1 Мета роботи

Вибір базових бібліотек/сервісів для подальшої реалізації криптосистеми.

2 Постановка задачі

Підгрупа 2C. Порівняння бібліотек OpenSSL, Crypto++, CryptoLib, PyCrypto для розробки гібрідної криптосистеми під Android/MacOs/IOs платформу.
Примітка. Бібліотека PyCrypto замінена на PyCryptodome.

3 Хід роботи

Обрано бібліотеки, знайдено їх реалізації на C++ та Python, обрано усі функції, написано усі функції, протестовано усі функції, замірян час, оформлені результати і звіт.

4 Опис функцій бібліотеки реалізації основних криптографічних примітивів обраної бібліотеки, з описом алгоритму, вхідних та вихідних даних, кодів повернення

5 Контрольний приклад роботи з функціями

SHA-256: 1.000.000 запусків, довжина випадкових текстів 500 символів.

ChaCha20: 1.000.000 запусків, довжина випадкових текстів 500 символів.

DSA 10.000 запусків, довжина випадкових текстів 500 символів.

Таблиця загального зайнятого часу у секундах для кожної функції на кожній бібліотеці. Виклики відбувались згідно зазначених умов, час брався середній з 5 вимірів.

Algo \ Lib	OpenSSL	Crypto++	PyCryptodome
SHA-256	3.17341	4.04513	5.3394
ChaCha20	1.71733	6.32638	8.8799
DSA keygen	0.288028	0.346091	1.421512
DSA signver	5.89138	8.4481	21.66875

Примітка. Генерація ключів у PyCryptodome працює з дещо нестабільною швидкістю

```c
int SHA256_Init(SHA256_CTX *c)
{
    memset(c, 0, sizeof(*c));
    c->h[0] = 0x6a09e667UL;
    c->h[1] = 0xbb67ae85UL;
    c->h[2] = 0x3c6ef372UL;
    c->h[3] = 0xa54ff53aUL;
    c->h[4] = 0x510e527fUL;
    c->h[5] = 0x9b05688cUL;
    c->h[6] = 0x1f83d9abUL;
    c->h[7] = 0x5be0cd19UL;
    c->md_len = SHA256_DIGEST_LENGTH;
    return 1;
}
```

Figure 1: SHA_INIT

```c
int HASH_UPDATE(HASH_CTX *c, const void *data_, size_t len)
{
    const unsigned char *data = data_;
    unsigned char *p;
    HASH_LONG l;
    size_t n;

    if (len == 0)
        return 1;

    l = (c->Nl + (((HASH_LONG) len) << 3)) & 0xffffffffUL;
    if (l < c->Nl)              /* overflow */
        c->Nh++;
    c->Nh += (HASH_LONG) (len >> 29); /* might cause compiler warning on
                                       * 16-bit */
    c->Nl = l;

    n = c->num;
    if (n != 0) {
        p = (unsigned char *)c->data;

        if (len >= HASH_CBLOCK || len + n >= HASH_CBLOCK) {
            memcpy(p + n, data, HASH_CBLOCK - n);
            HASH_BLOCK_DATA_ORDER(c, p, 1);
            n = HASH_CBLOCK - n;
            data += n;
            len -= n;
            c->num = 0;
            /*
             * We use memset rather than OPENSSL_cleanse() here deliberately.
             * Using OPENSSL_cleanse() here could be a performance issue. It
             * will get properly cleansed on finalisation so this isn't a
             * security problem.
             */
            memset(p, 0, HASH_CBLOCK); /* keep it zeroed */
        } else {
            memcpy(p + n, data, len);
            c->num += (unsigned int)len;
            return 1;
        }
    }

    n = len / HASH_CBLOCK;
    if (n > 0) {
        HASH_BLOCK_DATA_ORDER(c, data, n);
        n *= HASH_CBLOCK;
        data += n;
        len -= n;
    }

    if (len != 0) {
        p = (unsigned char *)c->data;
        c->num = (unsigned int)len;
        memcpy(p, data, len);
    }
    return 1;
}
```

Figure 2: SHA_UPDATE

```
int HASH_FINAL(unsigned char *md, HASH_CTX *c)
{
    unsigned char *p = (unsigned char *)c->data;
    size_t n = c->num;

    p[n] = 0x80;                /* there is always room for one */
    n++;

    if (n > (HASH_CBLOCK - 8)) {
        memset(p + n, 0, HASH_CBLOCK - n);
        n = 0;
        HASH_BLOCK_DATA_ORDER(c, p, 1);
    }
    memset(p + n, 0, HASH_CBLOCK - 8 - n);

    p += HASH_CBLOCK - 8;
# if   defined(DATA_ORDER_IS_BIG_ENDIAN)
    (void)HOST_l2c(c->Nh, p);
    (void)HOST_l2c(c->Nl, p);
# elif defined(DATA_ORDER_IS_LITTLE_ENDIAN)
    (void)HOST_l2c(c->Nl, p);
    (void)HOST_l2c(c->Nh, p);
# endif
    p -= HASH_CBLOCK;
    HASH_BLOCK_DATA_ORDER(c, p, 1);
    c->num = 0;
    OPENSSL_cleanse(p, HASH_CBLOCK);

# ifndef HASH_MAKE_STRING
#  error "HASH_MAKE_STRING must be defined!"
# else
    HASH_MAKE_STRING(c, md);
# endif

    return 1;
}
```

Figure 3: SHA_FINAL

```
int DSA_generate_parameters_ex(DSA *dsa, unsigned bits, const uint8_t *seed_in,
                               size_t seed_len, int *out_counter,
                               unsigned long *out_h, BN_GENCB *cb) {
  int ok = 0;
  unsigned char seed[SHA256_DIGEST_LENGTH];
  unsigned char md[SHA256_DIGEST_LENGTH];
  unsigned char buf[SHA256_DIGEST_LENGTH], buf2[SHA256_DIGEST_LENGTH];
  BIGNUM *r0, *W, *X, *c, *test;
  BIGNUM *g = NULL, *q = NULL, *p = NULL;
  BN_MONT_CTX *mont = NULL;
  int k, n = 0, m = 0;
  unsigned i;
  int counter = 0;
  int r = 0;
  BN_CTX *ctx = NULL;
  unsigned int h = 2;
  unsigned qsize;
  const EVP_MD *evpmd;

  evpmd = (bits >= 2048) ? EVP_sha256() : EVP_sha1();
  qsize = EVP_MD_size(evpmd);

  if (bits < 512) {
    bits = 512;
  }

  bits = (bits + 63) / 64 * 64;

  if (seed_in != NULL) {
    if (seed_len < (size_t)qsize) {
      return 0;
    }
    if (seed_len > (size_t)qsize) {
      /* Only consume as much seed as is expected. */
      seed_len = qsize;
    }
    memcpy(seed, seed_in, seed_len);
  }
```

Figure 4: DSA_generate_params_ex

```c
int DSA_generate_key(DSA *dsa) {
  int ok = 0;
  BN_CTX *ctx = NULL;
  BIGNUM *pub_key = NULL, *priv_key = NULL;
  BIGNUM prk;

  ctx = BN_CTX_new();
  if (ctx == NULL) {
    goto err;
  }

  priv_key = dsa->priv_key;
  if (priv_key == NULL) {
    priv_key = BN_new();
    if (priv_key == NULL) {
      goto err;
    }
  }

  do {
    if (!BN_rand_range(priv_key, dsa->q)) {
      goto err;
    }
  } while (BN_is_zero(priv_key));

  pub_key = dsa->pub_key;
  if (pub_key == NULL) {
    pub_key = BN_new();
    if (pub_key == NULL) {
      goto err;
    }
  }
```

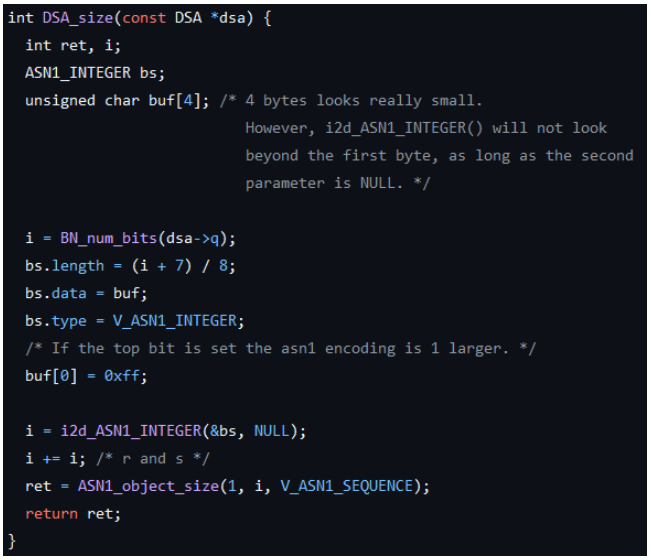
Figure 5: DSA_generate_key

```
void DSA_free(DSA *dsa) {
  if (dsa == NULL) {
    return;
  }

  if (!CRYPTO_refcount_dec_and_test_zero(&dsa->references)) {
    return;
  }

  CRYPTO_free_ex_data(&g_ex_data_class, dsa, &dsa->ex_data);

  BN_clear_free(dsa->p);
  BN_clear_free(dsa->q);
  BN_clear_free(dsa->g);
  BN_clear_free(dsa->pub_key);
  BN_clear_free(dsa->priv_key);
  BN_clear_free(dsa->kinv);
  BN_clear_free(dsa->r);
  BN_MONT_CTX_free(dsa->method_mont_p);
  CRYPTO_MUTEX_cleanup(&dsa->method_mont_p_lock);
  OPENSSL_free(dsa);
}
```

Figure 6: DSA_free

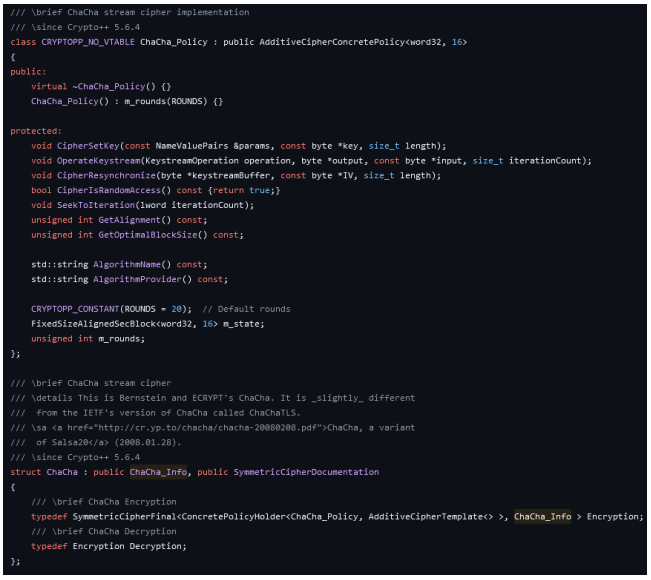
```
int DSA_sign(int type, const uint8_t *digest, size_t digest_len,
             uint8_t *out_sig, unsigned int *out_siglen, DSA *dsa) {
  DSA_SIG *s;

  s = DSA_do_sign(digest, digest_len, dsa);
  if (s == NULL) {
    *out_siglen = 0;
    return 0;
  }

  *out_siglen = i2d_DSA_SIG(s, &out_sig);
  DSA_SIG_free(s);
  return 1;
}

int DSA_verify(int type, const uint8_t *digest, size_t digest_len,
               const uint8_t *sig, size_t sig_len, const DSA *dsa) {
  int valid;
  if (!DSA_check_signature(&valid, digest, digest_len, sig, sig_len, dsa)) {
    return -1;
  }
  return valid;
}
```

Figure 7: DSA_sign and DSA_Verify



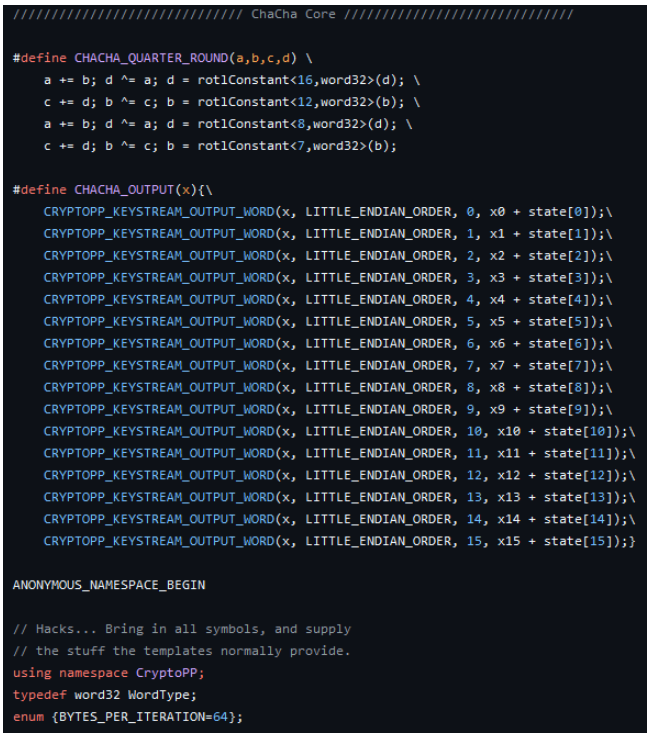
```
int DSA_size(const DSA *dsa) {
  int ret, i;
  ASN1_INTEGER bs;
  unsigned char buf[4]; /* 4 bytes looks really small.
                           However, i2d_ASN1_INTEGER() will not look
                           beyond the first byte, as long as the second
                           parameter is NULL. */

  i = BN_num_bits(dsa->q);
  bs.length = (i + 7) / 8;
  bs.data = buf;
  bs.type = V_ASN1_INTEGER;
  /* If the top bit is set the asn1 encoding is 1 larger. */
  buf[0] = 0xff;

  i = i2d_ASN1_INTEGER(&bs, NULL);
  i += i; /* r and s */
  ret = ASN1_object_size(1, i, V_ASN1_SEQUENCE);
  return ret;
}
```

Figure 8: DSA_size



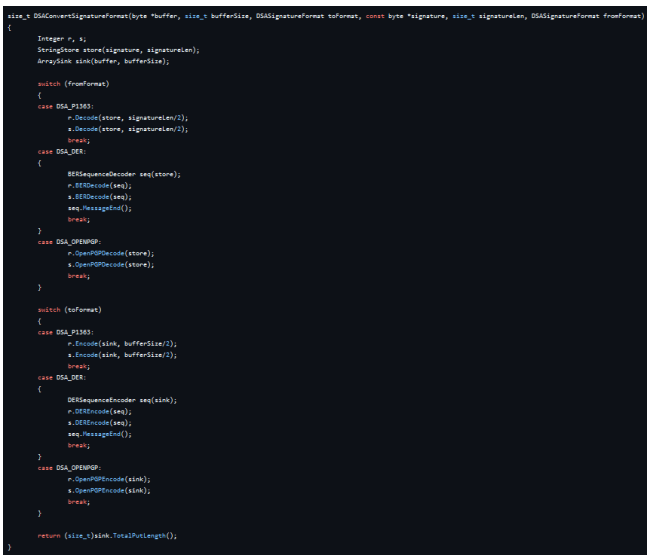
```
/// \brief ChaCha stream cipher implementation
/// \since Crypto++ 5.6.4
class CRYPTOPP_NO_VTABLE ChaCha_Policy : public AdditiveCipherConcretePolicy<word32, 16>
{
public:
    virtual ~ChaCha_Policy() {}
    ChaCha_Policy() : m_rounds(ROUNDS) {}

protected:
    void CipherSetKey(const NameValuePairs &params, const byte *key, size_t length);
    void OperateKeystream(KeystreamOperation operation, byte *output, const byte *input, size_t iterationCount);
    void CipherResynchronize(byte *keystreamBuffer, const byte *IV, size_t length);
    bool CipherIsRandomAccess() const {return true;}
    void SeekToIteration(lword iterationCount);
    unsigned int GetAlignment() const;
    unsigned int GetOptimalBlockSize() const;

    std::string AlgorithmName() const;
    std::string AlgorithmProvider() const;

    CRYPTOPP_CONSTANT(ROUNDS = 20);  // Default rounds
    FixedSizeAlignedSecBlock<word32, 16> m_state;
    unsigned int m_rounds;
};

/// \brief ChaCha stream cipher
/// \details This is Bernstein and ECRYPT's ChaCha. It is _slightly_ different
///  from the IETF's version of ChaCha called ChaChaTLS.
/// \sa <a href="http://cr.yp.to/chacha/chacha-20080208.pdf">ChaCha, a variant
///  of Salsa20</a> (2008.01.28).
/// \since Crypto++ 5.6.4
struct ChaCha : public ChaCha_Info, public SymmetricCipherDocumentation
{
    /// \brief ChaCha Encryption
    typedef SymmetricCipherFinal<ConcretePolicyHolder<ChaCha_Policy, AdditiveCipherTemplate<> >, ChaCha_Info > Encryption;
    /// \brief ChaCha Decryption
    typedef Encryption Decryption;
};
```

Figure 9: Crypto++ ChaCha20 Encryption and Decryption



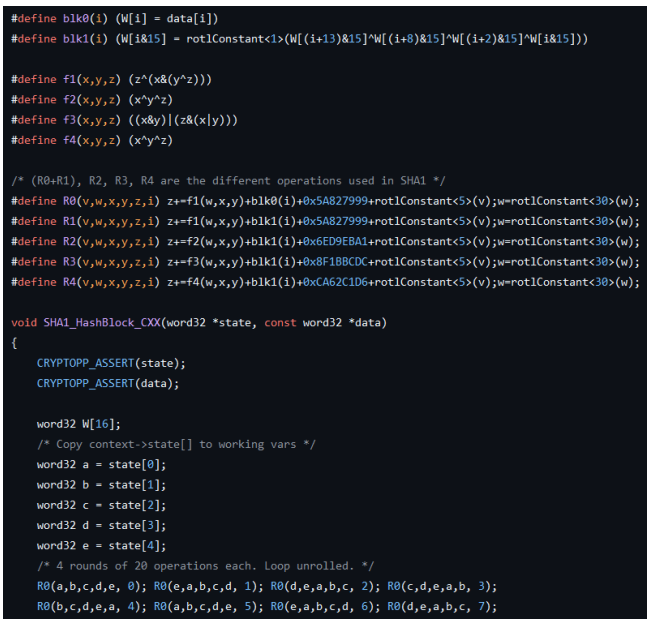
```
////////////////////////////// ChaCha Core //////////////////////////////

#define CHACHA_QUARTER_ROUND(a,b,c,d) \
    a += b; d ^= a; d = rotlConstant<16,word32>(d); \
    c += d; b ^= c; b = rotlConstant<12,word32>(b); \
    a += b; d ^= a; d = rotlConstant<8,word32>(d); \
    c += d; b ^= c; b = rotlConstant<7,word32>(b);

#define CHACHA_OUTPUT(x){\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 0, x0 + state[0]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 1, x1 + state[1]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 2, x2 + state[2]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 3, x3 + state[3]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 4, x4 + state[4]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 5, x5 + state[5]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 6, x6 + state[6]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 7, x7 + state[7]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 8, x8 + state[8]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 9, x9 + state[9]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 10, x10 + state[10]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 11, x11 + state[11]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 12, x12 + state[12]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 13, x13 + state[13]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 14, x14 + state[14]);\
    CRYPTOPP_KEYSTREAM_OUTPUT_WORD(x, LITTLE_ENDIAN_ORDER, 15, x15 + state[15]);}

ANONYMOUS_NAMESPACE_BEGIN

// Hacks... Bring in all symbols, and supply
// the stuff the templates normally provide.
using namespace CryptoPP;
typedef word32 WordType;
enum {BYTES_PER_ITERATION=64};
```

Figure 10: Crypto++ ChaCha20 Core



```
size_t DSAConvertSignatureFormat(byte *buffer, size_t bufferSize, DSASignatureFormat toFormat, const byte *signature, size_t signatureLen, DSASignatureFormat fromFormat)
{
	Integer r, s;
	StringStore store(signature, signatureLen);
	ArraySink sink(buffer, bufferSize);

	switch (fromFormat)
	{
	case DSA_P1363:
		r.Decode(store, signatureLen/2);
		s.Decode(store, signatureLen/2);
		break;
	case DSA_DER:
	{
		BERSequenceDecoder seq(store);
		r.BERDecode(seq);
		s.BERDecode(seq);
		seq.MessageEnd();
		break;
	}
	case DSA_OPENPGP:
		r.OpenPGPDecode(store);
		s.OpenPGPDecode(store);
		break;
	}

	switch (toFormat)
	{
	case DSA_P1363:
		r.Encode(sink, bufferSize/2);
		s.Encode(sink, bufferSize/2);
		break;
	case DSA_DER:
	{
		DERSequenceEncoder seq(sink);
		r.DEREncode(seq);
		s.DEREncode(seq);
		seq.MessageEnd();
		break;
	}
	case DSA_OPENPGP:
		r.OpenPGPEncode(sink);
		s.OpenPGPEncode(sink);
		break;
	}

	return (size_t)sink.TotalPutLength();
}
```

Figure 11: Crypto++ DSA Core



```
#define blk0(i) (W[i] = data[i])
#define blk1(i) (W[i&15] = rotlConstant<1>(W[(i+13)&15]^W[(i+8)&15]^W[(i+2)&15]^W[i&15]))

#define f1(x,y,z) (z^(x&(y^z)))
#define f2(x,y,z) (x^y^z)
#define f3(x,y,z) ((x&y)|(z&(x|y)))
#define f4(x,y,z) (x^y^z)

/* (R0+R1), R2, R3, R4 are the different operations used in SHA1 */
#define R0(v,w,x,y,z,i) z+=f1(w,x,y)+blk0(i)+0x5A827999+rotlConstant<5>(v);w=rotlConstant<30>(w);
#define R1(v,w,x,y,z,i) z+=f1(w,x,y)+blk1(i)+0x5A827999+rotlConstant<5>(v);w=rotlConstant<30>(w);
#define R2(v,w,x,y,z,i) z+=f2(w,x,y)+blk1(i)+0x6ED9EBA1+rotlConstant<5>(v);w=rotlConstant<30>(w);
#define R3(v,w,x,y,z,i) z+=f3(w,x,y)+blk1(i)+0x8F1BBCDC+rotlConstant<5>(v);w=rotlConstant<30>(w);
#define R4(v,w,x,y,z,i) z+=f4(w,x,y)+blk1(i)+0xCA62C1D6+rotlConstant<5>(v);w=rotlConstant<30>(w);

void SHA1_HashBlock_CXX(word32 *state, const word32 *data)
{
    CRYPTOPP_ASSERT(state);
    CRYPTOPP_ASSERT(data);

    word32 W[16];
    /* Copy context->state[] to working vars */
    word32 a = state[0];
    word32 b = state[1];
    word32 c = state[2];
    word32 d = state[3];
    word32 e = state[4];
    /* 4 rounds of 20 operations each. Loop unrolled. */
    R0(a,b,c,d,e, 0); R0(e,a,b,c,d, 1); R0(d,e,a,b,c, 2); R0(c,d,e,a,b, 3);
    R0(b,c,d,e,a, 4); R0(a,b,c,d,e, 5); R0(e,a,b,c,d, 6); R0(d,e,a,b,c, 7);
```

Figure 12: Crypto++ SHA256 Core

6 Обґрунтування вибору бібліотеки

Провівши перевірку, найкращим варіантом по швидкості є OpenSSL. Найкращий по зручності використання є Crypto++. OpenSSL також використовується у ряді Python бібліотек, як швидка реалізація рішення деяких задач. OpenSSL є більшою бібліотекою з більшим різноманіттям методів, на відміну від Crypto++ та PyCryptodome. PyCryptoDome виявився найдовшим і генерація ключів для DSA показує нестабільні результати. Також вибір методів є дуже обмеженим.