# PV204 Security technologies

## Trusted element, side channels attacks

**Petr Švenda** *svenda@fi.muni.cz* *@rngsec*
Centre for Research on Cryptography and Security, Masaryk University

CROCS
Centre for Research on
Cryptography and Security

# The masterplan for this lab

1. Project, teams
2. Implementation of modular exponentiation (RSA)
3. Understand naïve and square&multiply algorithm
   - Toy example with integers (32 bits)
4. Understand how to measure operation (clock())
   - Pre-prepared functions – console or file output
   - Visualization of multiple measurements (R, http://plot.ly...)
   - What can be inferred from measurements
5. Use large datatype MPI instead of `int` (> $10^2$ bits)
6. Understand blinding as protection technique
7. Assignment

# Project, teams, first deadline next week!

- Form group, exchange contact, agree periodic joint meetings
- Make sure you know what to do for the next deadline
  - Form team, pick three certificates, book with me (9.3.2020)
  - Analyze certificates, prepare report and presentation: (19.3.2020)

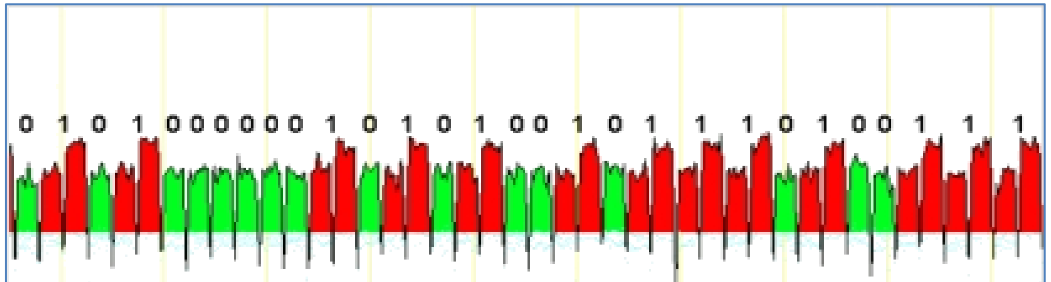
from lecture

# Faster modexp: Square and multiply algorithm

Executed always

Executed only when d_j == 1

```
// M = C^d mod N
// Square and multiply algorithm
x = C                          // start with ciphertext
for j = 1 to n {               // process all bits of private exponent
    x = x*x mod N              // shift to next bit by x * x (always)
    if (d_j == 1) {            // j-th bit of private exponent d
      x = x*C mod N            // if 1 then multiple by Ciphertext
    }
}
return x        // plaintext M
```



*Gilbert Goodwill, http://www.embedded.com/print/4408435*

- How to measure?
  - Exact detection from simple power trace
  - Extraction from overall time of multiple measurements

# Naïve vs. square and multiply algorithm

- `SideChannelExercise.zip` source code from IS
- Inspect naïve and square&multiply algorithm
  - Limited to integers (unsigned long) for simplicity
- Measure timings
  - Pre-prepared measurement functions
    - measureExponentiation()
      - clock() used for measurement (usually 1ms granularity)
    - measureExponentiationRepeat()
      - Make defined number of repeats and stores results into file
- Identify dependency of algorithm on secret value

# Setup

- Create new Visual Studio 2015 Project
  - File->New->Project->VisualC++->Win32 Console app
  - Turn off ‘Precompiled header’ and ‘SDL checks’
- Paste SideChannelExercise.cpp from IS instead of project’s main file
- Copy all remaining files into directory where stdafx.cpp is
- Add bignum.c into compilation
  - Solution->Source files->Add->Existing item
- Try to compile
- Insert breakpoint (begin of main()) – F9
- Run program in debug mode – F5
- Execute next step of program – F10

# Naïve modular exponentiation algorithm

```
typedef unsigned long ULONG;
const int ULONG_LENGTH = sizeof(ULONG);

ULONG naiveExponentiation(ULONG message, ULONG exponent, ULONG modulus) {
	ULONG result = message;
	for (int i = 1; i < exponent; i++) {
		// result = (result * message) % modulus; // this may cause type overflow
		result *= message;
		result %= modulus;
	}

	return result;
}
```

- What is disadvantage of this algorithm?
- Is algorithm vulnerable to timing side-channel?
- Is algorithm vulnerable to another side-channel?

```
typedef unsigned long ULONG;
const int ULONG_LENGTH = sizeof(ULONG);

ULONG squareAndMultiply(ULONG message, ULONG exponent, ULONG modulus) {
	// Obtain effective length of exponent in bits
	int sizeExponent = ULONG_LENGTH;
	ULONG mask = 1;
	ULONG bit = 0;
	for (int i = 0; i < ULONG_LENGTH * 8; i++) {
		bit = exponent & mask;
		if (bit != 0) { sizeExponent = i + 1; }
		mask <<= 1;
	}
	// Compute square and multiply algorithm
	ULONG result = 1;
	for (int i = sizeExponent - 1; i >= 0; i--) {
		//result = (result * result) % modulus; // this may cause type overflow
		result *= result;
		result %= modulus;
		if ((exponent & (1 << i)) != 0) { // given bit is not 0
			// result = (result * message) % modulus; // this may cause type overflow
			result *= message;
			result %= modulus;
		}
	}
	return result;
}
```

# Pair activity: Analysis of square&multiply algo

- Form pairs (e.g., with your neighbour) [21 minutes]
- Look and code together (before ready to answer the question)
- Two roles:
  - Educator – explains the answer to the given question to his/her pair
  - Sceptic – tries to find any flaw or weak point in Educator’s reasoning
- Educator keep explaining until Sceptic can’t find any flaw
  - not more than 3 mins per question
  - Sceptic notes down interesting issues raised
- Switch roles after every question (from next slide)

# Pair activity: Analysis of square&multiply algori

Pre-prepared function **`squareAndMultiply()`**

1. What is advantage of this algorithm with respect to naïve algorithm?
2. Is int (ULONG) enough for cryptographic security?
3. Is algorithm vulnerable to timing side-channel?
4. Which part of code is dependent on secret value?

Pre-prepared function **`measureExponentiation(65535, 65535, 1000003L, SQUAREANDMULTIPLY);`**

1. How is measured time depending on a secret value?
2. Is this algorithm easier or harder for attacker to mount timing attack wrt naïve exp?
3. How to mask dependency on secret exponent?

# Big integers (MPI from mbedTLS library)

- 32 bits are not enough, 4096 is recommended (RSA)
  - No native type in C/C++, use mbedTLS's MPI

```
void squareAndMultiplyMPI(const mpi* message, const mpi* exponent, const mpi* modulus,
                                                      mpi* result) {
        // Obtain length of exponent in bits
        int sizeExponent = 0;
        int maxBitLength = mpi_size(exponent) * 8;
        for (int i = 0; i < maxBitLength; i++) {
                if (mpi_get_bit(exponent, i) != 0) { sizeExponent = i + 1; }
        }
        // Compute square and multiply algorithm
        mpi_lset(result, 1);
        for (int i = sizeExponent - 1; i >= 0; i--) {
                mpi_mul_mpi(result, result, result); // result *= result;
                mpi_mod_mpi(result, result, modulus); // result %= modulus;
                if (mpi_get_bit(exponent, i) != 0) { // given bit is not 0
                    mpi_mul_mpi(result, result, message);
                    mpi_mod_mpi(result, result, modulus);
                }
        }}
```

# Create large (pseudo-)random MPI

- generateRNG() is function callback to fill single int

```
mpi message; mpi_init(&message);
mpi exponent; mpi_init(&exponent);
mpi modulus; mpi_init(&modulus);

// Cryptographically large number (2048b)
const int NUMBER_SIZE = 256;
// Init with pseudorandom values (prng will always start with same value)
mpi_fill_random(&message, NUMBER_SIZE, generateRNG, NULL);
mpi_fill_random(&exponent, NUMBER_SIZE, generateRNG, NULL);
mpi_fill_random(&modulus, NUMBER_SIZE, generateRNG, NULL);
// Fix MSb and LSb of modulus to 1
modulus.p[0] |= 1; mpi_set_bit(&modulus, 1, 1);

measureExponentiationMPI(&message, &exponent, &modulus, SQUAREANDMULTIPLY);
```

# Measure times with MPI

- Operation with large MPI can be measured
  - 100s-1000s ms
- Visualize histogram of multiple measurements
  - Pre-prepared measurements functions with file output
    - measureExponentiationRepeat()
  - https://plot.ly (Histogram, Traces→Range/bins 1)
  - pyplot, R…
- Try repeated measurement with the same data
- Try repeated measurement with the different data
- Are measured times constant? Why?

# Fix: Blinding

- Create squareAndMultiplyBlindedMPI() as improved version of squareAndMultiplyMPI()
  1. Generate random value *r* and compute $r^e$ mod N
  2. Compute blinded ciphertext *b = c * $r^e$ mod N*
  3. Decrypt *b* and then divide result by *r*

$$(r^e \cdot c)^d \cdot r^{-1} \mod n = r^{ed} \cdot r^{-1} \cdot c^d \mod n = r \cdot r^{-1} \cdot c^d \mod n = m.$$

- (r is random number, but invertible mod N) from lecture

# Defense introduced by OpenSSL

- RSA blinding: RSA_blinding_on()
  - https://www.openssl.org/news/secadv_20030317.txt
- Decryption without protection: $M = c^d \bmod N$
- Blinding of ciphertext *c* before decryption
  1. Generate random value *r* and compute $r^e \bmod N$
  2. Compute blinded ciphertext $b = c * r^e \bmod N$
  3. Decrypt *b* and then divide result by *r*
     - *r* is removed and only decrypted plaintext remains

$$(r^e \cdot c)^d \cdot r^{-1} \mod n = r^{ed} \cdot r^{-1} \cdot c^d \mod n = r \cdot r^{-1} \cdot c^d \mod n = m.$$

# Assignment 2: Protection via bogus branch

- Modify squareAndMultiplyMPI() code (use version without blinding)
  - When exponent's bit is not 1, add "bogus" branch as a protection against leakage

```
if (mpi_get_bit(exponent, i) != 0) { // given bit is not 0)
```

- Perform analysis on the original and protected version
  - Timing measurements for 1000 measurements, visualize as histograms
  - Scenario 1: Same data, same exponent
  - Scenario 2: Same exponent, low hamming weight of data
  - Scenario 3: Same exponent, high hamming weight of data
  - Scenario 4: Low/high hw exponent and random data
  - Optional: add 1-2 other sensible scenarios
- Compile and evaluate in Debug and Release profiles
  - Can you observe any difference? Why? What are security implications?
- IMPORTANT: Think! Some scenarios make sense. Some not.
  - Make explicit claim in discussion of every scenario if it makes sense from security point of view

# Assignment – what to submit

- Source code of your protected operation
- 2 pages of text and figures
  - Describe how bogus branch is removing the dependency of execution time on the secret exponent
  - Description of setup (methodology, sw, hw)
  - Visualized measurements (histograms, 4 scenarios)
  - Discussion of difference observed
  - Discussion of attack feasibility against original/protected implementation
- Submit **before 12.3. 23:59am** into IS HW vault
  - Soft deadline: -1.5 points for every started 24 hours

# Example vizualization (credits: J. Masarik)