PV204 Security technologies

Trusted element, side channels attacks

Petr Švenda svenda@fi.muni.cz
Faculty of Informatics, Masaryk University

CROCS
Centre for Research on Cryptography and Security

The masterplan for this lab

- Implementation of modular exponentiation (RSA)

1. Understand naïve and square&multiply algorithm
   - Toy example with integers (32 bits)
2. 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
3. Use large datatype MPI instead of int (> 10^2 bits)
4. Understand blinding as protection technique
5. Homework

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 (1ms granularity)
- Identify dependency of algorithm on secret value

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;
}
```

Square&multiply algorithm

- Pre-prepared function squareAndMultiply()
- What is advantage of this algorithm?
- Is algorithm vulnerable to timing side-channel?
- Which part of code is dependent on secret value?
- measureExponentiation(65535, 65535, 10000003L, SQUAREANDMULTIPLY);
  - What is the time required to complete operation?
  - What are implication for attacker's ability to mount timing attack?
- How to mask dependency on secret exponent?
- Is int (ULONG) enough for cryptographic security?

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 $b = c * r^e \bmod N$
  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)

Homework

- Finalize implementation of blinding of argument with MPI
- Create unit tests that will verify identical functionality
  - squareAndMultiplyMPI and squareAndMultiplyBlindedMPI
- Perform analysis of blinded and non-blinded version
  - Timing results for 1000 measurements, visualized 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 scenarios that will demonstrate interesting dependency not covered by previous ones)
- Describe your testing setup (methodology, sw/hw used, analysis…)
- Discuss the differences observed
- Discuss the feasibility of attack against non-blinded and blinded version

Homework – what to submit

- Source code of your blinded operation
- Test showing that it computes correctly
- 2 pages of text and figures
  - Description of setup
  - Visualized measurements (histograms, 4 scenarios)
  - Discussion of difference observed
  - Discussing of feasibility of attack against blinded/non-blinded implementation
- Submit **before 3.3. 23:59am** into IS HW vault