Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Фізико-технічний інститут

ЛАБОРАТОРНА РОБОТА №4

з дисципліни

«Криптографія»

на тему: «Побудова генератора псевдовипадкових послідовностей на лінійних регістрах зсуву (генератора Джиффі) та його кореляційний криптоаналіз»

Виконали:

студенти 3 курсу ФТІ

групи ФБ-74

Заіграєв Костянтин та Новіков Олексій

Перевірили:

Чорний О.

Савчук М. М.

Завадська Л. О.

Мета роботи :

Ознайомлення з деякими принципами побудови криптосистем на лінійних регістрах зсуву; практичне освоєння програмної реалізації лінійних регістрів зсуву (ЛРЗ); ознайомлення з методом кореляційного аналізу криптосистем на прикладі генератора Джиффі.

Порядок виконання роботи

0. Уважно прочитати методичні вказівки до виконання комп'ютерного практикуму.
1. За даними характеристичними многочленами написати програму роботи ЛРЗ L1 , L2, L3 і побудованого на них генератора Джиффі.
2. За допомогою формул (4) – (6) при заданому α визначити кількість знаків вихідної послідовності N* , необхідну для знаходження вірного початкового заповнення, а також поріг C для регістрів L1 та L2 .
3. Організувати перебір всіх можливих початкових заповнень L1 і обчислення відповідних статистик R з використанням заданої послідовності (z_i) , i=0,N*-1
4. Відбракувати випробувані варіанти за критерієм R > C і знайти всі кандидати на істинне початкове заповнення L1 .
5. Аналогічним чином знайти кандидатів на початкове заповнення L2 .
6. Організувати перебір всіх початкових заповнень L3 та генерацію відповідних послідовностей (s_i) .
7. Відбракувати невірні початкові заповнення L3 за тактами, на яких $x_i!=y_i$, де (x_i), (y_i) – послідовності, що генеруються регістрами L1 та L2 при знайдених початкових заповненнях.
8. Перевірити знайдені початкові заповнення ЛРЗ L1, L2 , L3 шляхом співставлення згенерованої послідовності (z_i) із заданою при i=0,N-1.

Вихідні дані:

Характеристичні многочлени:
– для L1 : $p(x) = x^{30} \oplus x^6 \oplus x^4 \oplus x \oplus 1$, що відповідає співвідношенню між членами послідовності $x_{i+30} = x_{i+6} \oplus x_{i+4} \oplus x_{i+1} \oplus x_i$;
– для L2 : $p(x) = x^{31} \oplus x^3 \oplus 1$, відповідна рекурента: $y_{i+31} = y_i \oplus y_{i+1}$;
– для L3 : $p(x) = x^{32} \oplus x^7 \oplus x^5 \oplus x^3 \oplus x^2 \oplus x \oplus 1$, відповідна рекурента: $s_{i+32} = s_i \oplus s_{i+1} \oplus s_{i+2} \oplus s_{i+3} \oplus s_{i+5} \oplus s_{i+7}$;
Імовірність помилки першого роду α = 0,01.

Результати

Варіант 7

000111110010000010110110100010011100110010111000101100111111011110101001100110001010100010110101101101001011001010001000
001011000111011100001001000011110101010000101100000000110001010101110000110000000011010000110111011111001010001111010100
001100111010110011111011011111011100110100001010111001011010101001100100010100110101011111011100010111001100100111001010
101000000000100010010101100100000111010100011000011100010010010011001101000100101100011100001111100011000001001101011000
010011010100110001011001011010101100011000011001100010000110000001111101001010101011111011100101001000011110110011001100110
010110010010111100000100001100101111011111011100000111110000010010100100010010101000011111111011100010110110101100010000
010111011111010100100001110011000000110011000111110101010000110011000101000000100000001001000111100100001111011100101101
110011100011101111101000111010101000101110010011010111010100011100101101101111000101000111010101100001011010010011010
101011000100101001011011000000111001101101010011010001110110111011001011000000011110010010110001001111101001010111101001
011100000000111101110011110001100110000111011001100101110100011100111110011010111100101110100010110011101101001001111001
101010110101001111110001011011111110001100011001101000101101011100100101110110001010000111111001010000111111001111001000
001111111011111101101000100011111100001111100000000110110111001110001010010101001100100001100111011010001111101010000010
100011110110011101101100011011111000100010101000010100111111110000110100111001100001111111011011000001000100010001000011
101001111110000001101011100101010110111001111010111010100010001110100010000110110001011011001000110000101100000010010111
110110100110111011000001010101001000011010011011000001100100101001000100000110011100001010100111011100101000110110001001
101110011011100111000011101010001101111111101111101010010000010011110111010010010011100000100011101110110011101110111110
000100101111111010101000000110010000010100001110000100011101111100101101001100110011111101101001110000010111100010001010
001011110010010010110000 1

Початкові значення послідовностей:

L1: 00101111101001000000000011001100
L2: 00011111001000111011111010101000
L3: 11000110001100110000100010111000

Повні послідовності:

L1 gamma:
0010111110100100000000001100110110001110101100001111101110110111010100111000000101101001111000001011001100
1111010111000000011000101001110011001001011110001010010101101110010010001010101110001011000100011110011010
0110111010000101010111100100111111101010001101100101011101010000010000101010100101100011100100010001110011
0101011110011110010111011101010110011011010000010011100100000111001001100111101001101010110010100100111011
0010101000010110100110001101010101101011110100000100010111001111110101011001011110000101011000010000100010
0001100000011111000110110000011010111000010110010111001100111001011011001000000110100010000110010111011111
0000010000011110000100010100000100101001000111011101111001101110101101100011000001111111101010011010101 00
1000100010000000010101100101110011000110010000000010000001100010111110110010100111111000110110011101011001
1011101100101000001010000111111011010111110000111000001101101111110100100111000110100010111001110011011110
0000011100001001011111000000111011101111010011010001110100011000010110010000011011000010111011001110101101
0111110100101100000101011111110001111010110000000111101101110110010011010110011110011101101101010110000011
0110000100001101000110011010111111000010000000110110111111110011100110111010001011001111001011111111100111
1000011010000111000111101100010100001000111111011111100101110110101001101001111100000101011000001001011001
0000000000110001101010011101100000111100100010000100011110010010101111000011010101010100110011000010100 10
1111110000010100111001011001111000000010010100001000010010001010110111110110001110011110000101010011110 11
1101011000110001000111110001001001010100000101000101001000011010111000011011010101000100111001010101001101
1010101110100010111000011001101010111001110001010111011010101000100111010001010101101101010111001100110011
0010001101010001101010101011111000010010000010111110111010011110010100010110010111110111010011101111001000
1100101111101000000010111001110001100010111010010000100111101000110011111010011111111010000110010000110110
100100101001111110001010010110 0011

L2 gamma:
0001111100100011101111101010100111001100011111001001011111001111010111110011000001010011011010110100110101
1001011001000001100010010011001001001000100110111000000101000000000110001000111100010110100000001101110011 1
1011110100010100011000111011010010101011011011101111110000100011111000000011001000111010011110011100001101
0001111101001101101111110110010111100101001000000100100010010101100110000100010000011000011100101011010011 0
0100110110111110011100010010100001000000100110111111000011010100100010001000100011101100111000011001100110
0111110001011111011010101010101100111101010010000111111111100101101011100001011100000000110010001101111010 1
0111100001101000111000101011110101110110010111111110111010101101100010010100000100110111100000111000011010
0100010001011100111111011001000011001101011101100010001000101101010011011000111001100110100011100100001111
1110101010010111111000101110000101111100010100011110101111010101001111011011110101101010111110011010000010
1011000111101001101001010010111001111101010010010001100010111011001011100000000111011101011000100101111000 0
1111001101100111000010101110111011010000101111101011101100110000101010101001011011000101011010111111110001
0000011101110001101000001111001001111001111110010100111011000011011011000110011001110001101100000000111101 0
1010111111100001100001110101111110100001110110110111101101000101010111100000000010100001011011111010111000 0
0101101010100001001011011110010100011111010100001000010110011011110010111010100101010010100010110010110111
0001111000110110100100100001111111011111000000100000000101110000100100111001001000010101111010000000111100000
0101011101010101000111011100010111011011111111011111001111101011000001000010010011011001011001100100101000 0
0010000100100101010000011010001001010000000111101001100101100001101000011101010010100100110110010101111011
1000110000010000100111010100111111011001001010001110111001100010001000001101111110011101011100110010011000 1
0011011101101110101000010111000100011000001101110101010111110011101100110001101111110100110111000101011110
00100010100100011111011101011110 01

L3 gamma:
1100011000110011000010001011100010111100111001000011010110100111001111111000000001010000100011010000100101 10
0001011000000111011101110100110110001010011110101011000111111000100101101011111001011001011101101101110000 1
0111011010000001000011011001110000110010011010000011000110101010011101010100011111111001011100001110101111 1
0101110110011101111010100010001100000000101110110100100100001000110011000011011100010010000000011100110000 10
1011010101111101010010010101010010110000100101010011010011000100110011110111111011010010100001100100011100
1111011011101001001010100101010011011110011001101110010111111100000101111010100111010111100111011111101011
0000101010011101111100000010110101111100011110000100001010111000000100101100000101101001111111111101001011 0
1010010010010000100101001101001010000000100101111110111011101110010011110010011001010101000101111110001100 10
0001001010101101010101000100010111101011001010001010010110111111000111001110110000011110100010000101101001 1
0100000111000110111010010110101000010000110011100100110100001100100010101111011010101100100101111100011011
1101111111101001000010110000100111101001011100111000110010111011010000100111000100001000001100111010101001
1110100110011010100111011101101100101110000011000110100100101010011111011111111111110011011110101110111100 0
0011110010001100110011001011000010000100001111111100110011000101001001010101111011111010010001001101010011
0011010101111100001000011101110000110000000000001001111110011010110001000010110110101100001010001110111011 00
1110101101001111011101000000100100000010011000011011010011101010111000001100010101000001111011111110111100 11
1100000000000111001010001101110101000011010111011110100111110001000101110011011100010100111101101010000101
0101010010110011111000011101100000110001010001110011111100111101110010111100101001011101000000111011000101000

0000101101010111001000101001110010101011011010011110010011110010110100101001110001111100111100110110100100001100111101101001101011110000101001001110101100110101000100001011000100110100000010011011100000100010101010011001101011111100111110011110

Код програми

```
#include "compute.cuh"

#include <bitset>//num to bin for answer
#pragma comment (lib, "cudart64_102.dll")


using namespace std;


double getBeta(pPolynom poly, double percent) {
	double Border = double(1) / pow(2, poly->polynom[0]);
	return Border * percent;
}

void getParams(pPolynom poly, pcrackParams params, int *C, int *Ns) {
	boost::math::normal dist(0.0, 1.0);
	double beta = getBeta(poly, params->betaPercent);
	double QuantileAlpha = quantile(dist, 1 - params->alpha);
	double QuantileBeta  = quantile(dist, 1 - beta);
	double NsDouble = pow((QuantileBeta + ((QuantileAlpha * sqrt(params->p1 * (1 - params->p1))) / sqrt(params->p2 * (1 - params->p2)))) * (sqrt(params->p2 * (1 - params->p2)) / (params->p2 - params->p1)), 2);
	*C = (NsDouble * params->p1 + QuantileAlpha * sqrt(NsDouble * params->p1 * (1 - params->p1))) + 1;
	*Ns = NsDouble + 1;
}


double getCuda_optimalSize(pPolynom poly, int Ns, int z_size) {
	int maxBlocks, maxThreads;
	unsigned long long memoryByte, mem_per_thread, mem_shared, mem_total;

	mem_per_thread = 2048 * sizeof(unsigned int) + 2 * 64 * sizeof(unsigned int) + sizeof(unsigned char) + 3 * sizeof(unsigned int) + 10 * sizeof(int) + 32 * sizeof(unsigned int);

			//poly2->polynom = new unsigned int[3]{ 31, 3, 0 };
	//poly2->size = 3;
	poly2->polynom = new unsigned int[5]{ 26, 6, 2, 1, 0 };
	poly2->size = 5;
	pPolynom poly3 = new Polynom;
	//poly3->polynom = new unsigned int[7]{ 32, 7, 5, 3, 2, 1, 0 };
	//poly3->size = 7;
	poly3->polynom = new unsigned int[5]{ 27, 5, 2, 1, 0 };
	poly3->size = 5;
	vector<unsigned int> L1_probable, L2_probable, result;
	plfstrHistory lfstr1_init = new lfstrHistory;
	plfstrHistory lfstr2_init = new lfstrHistory;

	int NsInt, C;
	getParams(poly1, params, &C, &NsInt);

	lfstr1_init->parts = unsigned int(getCuda_optimalSize(poly1, NsInt, C));

	unsigned int period;

	cout << "LFSTR1" << endl;
	cout << "CPU calculation of init vectors for CUDA parallelism" << endl;
	lfstr(1, poly1, &period, lfstr1_init);

	cuda_lfstrCrack(lfstr1_init, poly1, z, NsInt, C, &L1_probable);
	cout << "Num of probable vectors: " << L1_probable.size() << endl;
	/*
	for (int i = 0; i < L1_probable.size();i++) {
		cout << L1_probable[i] << endl;
	}
	*/









	cout << "LFSTR2" << endl;
	cout << "CPU calculation of init vectors for CUDA parallelism" << endl;
	getParams(poly2, params, &C, &NsInt);
	lfstr2_init->parts = unsigned int(getCuda_optimalSize(poly2, NsInt, C));
	lfstr(1, poly2, &period, lfstr2_init);
	mem_shared = 40000 * sizeof(unsigned int) + 64 * sizeof(unsigned int) + poly->size * sizeof(unsigned int) + sizeof(unsigned int) + 100 * sizeof(unsigned int);
	mem_total = mem_per_thread + mem_shared;

	cudaDeviceProp deviceProp;
	cudaGetDeviceProperties(&deviceProp, 0);
	cout << "Found: " << deviceProp.name << endl;
	maxBlocks = deviceProp.maxGridSize[0];
	maxThreads = deviceProp.maxThreadsDim[0];
	memoryByte = deviceProp.totalGlobalMem;
	//memoryByte -= pow(2, 28);
	memoryByte = (double)(memoryByte) * 0.9;
	memoryByte -= mem_shared;


	return ((double)(memoryByte / mem_per_thread)) * 0.95;
}


int main() {
	vector <unsigned int> z;
	//int N = getZ("test_big", &z);
	int N = getZ("test_small", &z);
	pcrackParams params = new crackParams;
	params->alpha = 0.01;
	params->p1 = 0.25;
	params->p2 = 0.5;
	params->betaPercent = 0.99;

	pPolynom poly1 = new Polynom;
	//poly1->polynom = new unsigned int[5]{ 30, 6, 4, 1, 0 };
	//poly1->size = 5;
	poly1->polynom = new unsigned int[3]{ 25, 3, 0 };
	poly1->size = 3;
	pPolynom poly2 = new Polynom;

	system("pause");

	return 0;
}


int getZ(string file, vector <unsigned int>* z)
{
	fstream zFile;
	char* read = new char[1];
	zFile.open(file, ios::in | ios::binary | ios::ate);
	streampos fileLen = zFile.tellg();
	zFile.seekg(0, ios::beg);
	for (int i = 0; i < fileLen; i++)
	{
		if (i % 32 == 0)
			z->push_back(0);
		zFile >> read[0];
		(*z)[i / 32] |= (int(read[0]) - 48) << (31 - (i % 32));
	}
	delete[] read;
	zFile.close();
	return fileLen;
}



void lfstr(unsigned int init, pPolynom polynom, unsigned int* period, plfstrHistory history) {
	unsigned char new_bit;
	unsigned int curr = init;
	unsigned int bit_length = polynom->polynom[0];
	*period = 0;
	unsigned int max_iters = (1 << polynom->polynom[0]) - 1;
	if (polynom->polynom[0] == 32)
		max_iters = 4294967295;
	unsigned int modulo;
	//cout << max_iters << endl;

	if (history != nullptr) {
		if (history->parts == 0)
			modulo = 1;
		else
			modulo = max_iters / history->parts;
	}
```

```
		cuda_lfstrCrack(lfstr2_init, poly2, z, NsInt, C, &L2_probable);

		cout << "Num of probable vectors: " << L2_probable.size() << endl;
		/*for (int i = 0; i < L2_probable.size(); i++) {
			//cout << L2_probable[i] << endl;
		}*/



		cout << "LFSTR3" << endl;


		cout << "CPU preparations for CUDA" << endl;

		cuda_GeffeCrack(poly1, poly2, poly3, z, L1_probable, L2_probable,
&result);


		cout << "\n------------------------------------------------------------
\nRESULT:" << endl;

		for (int i = 0; i < result.size(); i++) {
			string bits = std::bitset<32>(result[i]).to_string();
			reverse(bits.begin(), bits.end());
			cout << "L" << i+1 << ": " << bits << endl;
		}

		cout << "L1 gamma:" << endl;
		vector<unsigned int> arr = get_lfstr_res(result[0], poly1);
		for (int i = 0; i < arr.size(); i++)
			cout << std::bitset<32>(arr[i]).to_string();
		cout << "\n" << endl;

		cout << "L2 gamma:" << endl;
		arr = get_lfstr_res(result[1], poly2);
		for (int i = 0; i < arr.size(); i++)
			cout << std::bitset<32>(arr[i]).to_string();
		cout << "\n" << endl;

		cout << "L3 gamma:" << endl;
		arr = get_lfstr_res(result[2], poly3);
		for (int i = 0; i < arr.size(); i++)
			cout << std::bitset<32>(arr[i]).to_string();
		cout << "\n" << endl;

		cout << "EO CRACKING!" << endl;

			R = 0;

			for (int i = 1; i < polynom->size; i++) {
				new_bit ^= (curr & (1 << polynom-
>polynom[i])) >> polynom->polynom[i];
			}


			temp_bit = (x[vectorSize - 1] & (1 << 31)) >> 31;
			x[vectorSize - 1] <<= 1;
			x[vectorSize - 1] |= curr & 1;
			for (int i = vectorSize-2; i >= 0; i--) {
				temp_bit2 = (x[i] & (1 << 31)) >> 31;
				x[i] <<= 1;
				x[i] |= temp_bit;
				temp_bit = temp_bit2;
			}


			if (history.size() == Ns-1) {
				for (int i = 0; i < Ns; i++) {
					R += ((z[i / 32] & (1 << (31 -
(i % 32)))) >> (31 - (i % 32))) ^ ((x[(2048 - Ns + i) / 32] & (1 << (31 -
((2048 - Ns + i) % 32)))) >> (31 - ((2048 - Ns + i) % 32)));
				}
				if (R < C) {
					probable-
>push_back(history[0]);
				}
				history.erase(history.begin());
			}
			history.push_back(curr);


			curr >>= 1;
			curr |= new_bit << (bit_length - 1);

			/*
			if (final_Countdown) {
				*final_Countdown--;
				if (*final_Countdown == 0)
					return;
			}

			if (curr == init) {
				final_Countdown = new int(Ns);
			}
		while (true) {
			new_bit = 0;

			if (history != nullptr && *period % modulo == 0)
				history->history.push_back(curr);

			*period += 1;

			for (int i = 1; i < polynom->size; i++) {
				new_bit ^= (curr & (1 << polynom-
>polynom[i])) >> polynom->polynom[i];
			}

			curr >>= 1;
			curr |= new_bit << (bit_length - 1);

			if (curr == init || *period >= max_iters) {
				if (history != nullptr)
					history->size = history-
>history.size();
				return;
			}
		}
}



void lfstr_Crack_old(unsigned int init, pPolynom polynom, vector <unsigned int>
z, int Ns, int C, vector <unsigned int>* probable) {
		unsigned char new_bit, temp_bit, temp_bit2;
		unsigned int curr = init;
		unsigned int bit_length = polynom->polynom[0];
		int* final_Countdown=nullptr;

		unsigned int max_iters = (1 << polynom->polynom[0]) - 1;
		cout << max_iters << endl;

		vector<unsigned int> history, x = z;
		int vectorSize = x.size();

		int R;
		for (int iter = 0; iter < max_iters + Ns; iter++) {
			if (iter % 10000000 == 0)
				cout << iter << endl;


			new_bit = 0;
				return;
			printf("%d) %d\n", iter, curr);*/

			/*
			if (iter == 15648) {
				for (unsigned int i = 0; i < vectorSize; i++)
{
					printf("%d) \t%d\n", i, x[i]);
				}
				return;
			}
			*/

			if (history.size() == Ns - 1) {
				for (int i = 0; i < Ns; i++) {
					R += ((z[i / 32] & (1 << (31 -
(i % 32)))) >> (31 - (i % 32))) ^ ((x[(2048 - Ns + i) / 32] & (1 << (31 - ((2048 -
Ns + i) % 32)))) >> (31 - ((2048 - Ns + i) % 32)));
				}
				if (R < C) {
					probable-
>push_back(history[0]);
				}
				history.erase(history.begin());
			}
			history.push_back(curr);


			curr >>= 1;
			curr |= new_bit << (bit_length - 1);

		}
		history.clear();
		history.shrink_to_fit();
		x.clear();
		x.shrink_to_fit();
		z.clear();
		z.shrink_to_fit();
}

void cuda_lfstrCrack(plfstrHistory init_arr, pPolynom polynom, vector <unsigned
int> z, int Ns, int C, vector<unsigned int>* probable) {
		vector<unsigned int> polynom_vector;
		for (int i = 0; i < polynom->size; i++)
			polynom_vector.push_back(polynom->polynom[i]);

		kernel_crackLFSTR(init_arr, polynom_vector, z, Ns, C, probable);
```

```
                        */
                }
}


void lfstr_Crack(unsigned int init, pPolynom polynom, vector <unsigned int> z,
int Ns, int C, vector <unsigned int>* probable) {
                unsigned char new_bit, temp_bit, temp_bit2;
                unsigned int curr = init;
                unsigned int bit_length = polynom->polynom[0];

                unsigned int max_iters = (1 << polynom->polynom[0]) - 1;
                cout << max_iters << endl;

                vector<unsigned int> history, x = z;
                int vectorSize = x.size();
                for (int i = 0; i < x.size(); i++)
                        x[i] = 0;

                int R;
                for (int iter = 0; iter < max_iters + Ns; iter++) {
                        if (iter % 10000000 == 0)
                                cout << iter << endl;



                        new_bit = 0;
                        R = 0;

                        for (int i = 1; i < polynom->size; i++) {
                                new_bit ^= (curr & (1 << polynom-
>polynom[i])) >> polynom->polynom[i];
                        }


                        temp_bit = (x[vectorSize - 1] & (1 << 31)) >> 31;
                        x[vectorSize - 1] <<= 1;
                        x[vectorSize - 1] |= curr & 1;
                        for (int i = vectorSize - 2; i >= 0; i--) {
                                temp_bit2 = (x[i] & (1 << 31)) >> 31;
                                x[i] <<= 1;
                                x[i] |= temp_bit;
                                temp_bit = temp_bit2;
                        }

                        /*
                        if (iter == 1489)
void cuda_GeffeCrack(pPolynom polynom1, pPolynom polynom2, pPolynom
polynom3, vector <unsigned int> z, vector <unsigned int> lfstr1, vector
<unsigned int> lfstr2, vector<unsigned int>* result) {
                //unsigned int z;
                vector<unsigned int> lfstr1_res, lfstr2_res, polynom_vector, temp,
temp2, pairs, cudaRes, arr;
                vector<vector<unsigned int>> polynoms;
                unsigned int known_mask, unknown_mask, known_z, possible = 0;
                for (int i = 0; i < polynom1->size; i++)
                        polynom_vector.push_back(polynom1->polynom[i]);
                polynoms.push_back(polynom_vector);
                polynom_vector.clear();
                for (int i = 0; i < polynom2->size; i++)
                        polynom_vector.push_back(polynom2->polynom[i]);
                polynoms.push_back(polynom_vector);
                polynom_vector.clear();
                for (int i = 0; i < polynom3->size; i++)
                        polynom_vector.push_back(polynom3->polynom[i]);
                polynoms.push_back(polynom_vector);
                polynom_vector.clear();
                /*
                for (int i = 0; i < lfstr1.size(); i++) {
                        temp = get_lfstr_res(lfstr1[i], polynom1);
                        for (int j = 0; j < temp.size(); j++)
                                lfstr1_res.push_back(temp[j]);
                }
                for (int i = 0; i < lfstr2.size(); i++) {
                        temp = get_lfstr_res(lfstr2[i], polynom2);
                        for (int j = 0; j < temp.size(); j++)
                                lfstr2_res.push_back(temp[j]);
                        kernel_crackGeffe(init_arr, polynom_vector, z,
lfstr1_res, lfstr2_res, &cudaRes);
                        lfstr2_res.clear();
                        cudaRes.clear();
                        cout << i << endl;
                        if (cudaRes[2] != 0)
                                break;
                }

                known_mask = lfstr1_local[0] ^ lfstr2_local[0];

                if (lfstr1_local[0] & ~known_mask != z_local[0] & ~known_mask)
                        return;

                known_z |= (lfstr1_local[0] & known_mask) ^ (~(z_local[0] &
known_mask));
                polynom_vector.clear();
                polynom_vector.shrink_to_fit();
                init_arr->history.clear();
                init_arr->history.shrink_to_fit();
                delete init_arr;
                sort(probable->begin(), probable->end());
                probable->erase(unique(probable->begin(), probable->end()),
probable->end());
}


vector <unsigned int> get_lfstr_res(unsigned int init, pPolynom polynom, int
iterations) {
                unsigned char new_bit, temp_bit, temp_bit2;
                unsigned int curr = init;
                unsigned int bit_length = polynom->polynom[0];
                int vector_size;
                if (iterations % 32 == 0)
                        vector_size = iterations / 32;
                else
                        vector_size = (iterations / 32) + 1;
                vector<unsigned int> res(vector_size);

                for (int iter = 0; iter < iterations; iter++) {
                        new_bit = 0;

                        for (int i = 1; i < polynom->size; i++) {
                                new_bit ^= (curr & (1 << polynom-
>polynom[i])) >> polynom->polynom[i];
                        }

                        temp_bit = (res[vector_size - 1] & (1 << 31)) >> 31;
                        res[vector_size - 1] <<= 1;
                        res[vector_size - 1] |= curr & 1;
                        for (int i = vector_size - 2; i >= 0; i--) {
                                temp_bit2 = (res[i] & (1 << 31)) >> 31;
                                res[i] <<= 1;
                                res[i] |= temp_bit;
                                temp_bit = temp_bit2;
                        }

                        curr >>= 1;
                        curr |= new_bit << (bit_length - 1);

                }
                return res;
}
//                              cout << "Possible" << possible++ << ": "
<< lfstr1[i] << " " << lfstr2[j] << endl;
                                arr.push_back(lfstr1[i]);
                                temp2.push_back(lfstr2[j]);
                                pairs.push_back(i * lfstr2.size() + j);
                                //known_z |= (lfstr1_local[0] &
known_mask) ^ (~(z_local[0] & known_mask));
                        }
                }



                sort(arr.begin(), arr.end());
                arr.erase(unique(arr.begin(), arr.end()), arr.end());
                temp = arr;
                sort(temp2.begin(), temp2.end());
                temp2.erase(unique(temp2.begin(), temp2.end()), temp2.end());

                for (int i = 0; i < pairs.size(); i++) {
                        int index_l1 = pairs[i] / lfstr2.size();
                        int index_l2 = pairs[i] % lfstr2.size();
                        for (int j = 0; j < temp.size(); j++) {
                                if (lfstr1[index_l1] == temp[j]) {
                                        index_l1 = j;
                                        break;
                                }
                        }
                        for (int j = 0; j < temp2.size(); j++) {
                                if (lfstr2[index_l2] == temp2[j]) {
                                        index_l2 = j;
                                        break;
                                }
                        }
                        pairs[i] = index_l1 * temp2.size() + index_l2;
                }
                lfstr1 = temp;
                lfstr2 = temp2;
                temp2.clear();
                temp2.shrink_to_fit();

                for (int i = 0; i < lfstr1.size(); i++) {
                        temp = get_lfstr_res(lfstr1[i], polynom1);
                        for (int j = 0; j < temp.size(); j++)
                                lfstr1_res.push_back(temp[j]);
                }

                for (int i = 0; i < lfstr2.size(); i++) {
```

```
		for (int i = 0; i < lfstr1.size(); i++) {
			vector<unsigned int>t1 = get_lfstr_res(lfstr1[i],
polynom1, 2048);
			for (int j = 0; j < lfstr2.size(); j++) {
				vector<unsigned int>t2 =
get_lfstr_res(lfstr2[i], polynom2, 2048);
				bool tr = true;
				for (int k = 0; k < 2048; k++) {
					known_mask = t1[i] ^ t2[j];

					if ((t1[i] & ~known_mask) !=
(z[0] & ~known_mask)) {
						tr = false;
						break;
					}
				}
				if (tr == true) {
					cout << "Possible" <<
possible++ << ": " << lfstr1[i] << " " << lfstr2[j] << endl;
					temp1.push_back(lfstr1[i]);
					temp2.push_back(lfstr2[i]);
					pairs.push_back(i *
lfstr2.size() + j);
				}
				//known_z |= (lfstr1_local[0] &
known_mask) ^ (~(z_local[0] & known_mask));
			}
		}

		*/

		unsigned int z_rev;
		//temp1.empty();
		z_rev = reverseBits(z[0]);//get_lfstr_res(z[0], polynom1, 32);
		unsigned int end_mask = pow(2, min(polynoms[0][0],
polynoms[1][0], polynoms[2][0])) - 1;

		for (int i = 0; i < lfstr1.size(); i++) {
			for (int j = 0; j < lfstr2.size(); j++) {
				known_mask = lfstr1[i] ^ lfstr2[j];
				unknown_mask = ~known_mask;
				unknown_mask &= end_mask;

				if ((lfstr1[i] & unknown_mask) != (z_rev &
unknown_mask))
					continue;
```

```
				temp = get_lfstr_res(lfstr2[i], polynom2);
				for (int j = 0; j < temp.size(); j++)
					lfstr2_res.push_back(temp[j]);
		}
		temp.clear();
		temp.shrink_to_fit();
		cout << "Number of probable pairs: "<< pairs.size() << endl;
		unsigned int* cudaPointer = kernel_crackGeffe(polynoms[2],
min(polynoms[0][0], polynoms[1][0], polynoms[2][0]), z, lfstr1_res, lfstr2_res,
pairs);
		cudaRes.assign(cudaPointer, cudaPointer + 3);
		free(cudaPointer);
		polynom_vector.clear();
		polynom_vector.shrink_to_fit();
		result->push_back(lfstr1[cudaRes[0]]);
		result->push_back(lfstr2[cudaRes[1]]);
		result->push_back(cudaRes[2]);
}


unsigned int reverseBits(unsigned int num)
{
		unsigned int count = sizeof(num) * 8 - 1;
		unsigned int reverse_num = num;

		num >>= 1;
		while (num)
		{
			reverse_num <<= 1;
			reverse_num |= num & 1;
			num >>= 1;
			count--;
		}
		reverse_num <<= count;
		return reverse_num;
}
```

Висновки:

Під час данного комп’ютерного практикуму, ми ознайомились з деякими принципами побудови криптосистем на лінійних регістрах зсуву та з методом кореляційного аналізу криптосистем на прикладі генератора Джиффі.