CTAP Open Assessment

Exam no: Y0076159

20th November 2016

Contents

1 Breaking Stream Ciphers

i. Implementation and Challenge

The stream cipher implementation, found in appendix B (written in Python), can be ran using the Python 2.7.6 compiler with the command `python stream.py`. This will produce the challenge 25 bit output of 3 Lfsrs with initial states 97, 975 and 6420:

1 0 1 0 1 0 1 1 1 0 1 0 1 1 0 0 0 0 0 0 1 0 0 1 1

Code description

The class `Lfsr` implements an LFSR with a tap sequence, size and initial state. It contains 2 methods: `calc_tap`, which calculates the tap value for the current register value, and `shift` which calculates the tap value, shifts the register and outputs the register's value. The 3 LFSR's are combined using `combine_lfsr_outputs`, which uses a lookup table to implement the boolean operation.

ii. Cryptanalysis

Analysis

Linear func.	Walsh transform	Attack No	Attack description
R_3	0		
R_2	0		
$R_2 \oplus R_3$	0		
R_1	-4	1	No dependencies, has a single outlier agreement (see appendix A
$R_1 \oplus R_3$	-4	3	Requires R_1, attack with $R_1 \oplus R_3 \oplus K$
$R_1 \oplus R_2$	4	2	Requires R_1, attack with $R_1 \oplus R_2 \oplus K$
$R_1 \oplus R_2 \oplus R_3$	-4		

Using the Walsh transform displayed in appendix D, we can implement a divide-and-conquer strategy. LFSR1 correlates quite strongly with the output (Walsh value of -4, 2 agreements and 6 disagreements resulting in a probability of 2/8 that it will agree with the function's output, with a bias of 0.25); this is reflected by the outlying value in the graph of sub-keys and their agreements with outputs in appendix A. As such, it can be brute-forced to obtain it's key value by iterating over the space of sub-keys, and finding the one with the highest agreement (sub-key value 27, with an agreement of 0.253).

The next attack targets $R_1 \oplus R_2$, as it has a Walsh value of 4 (6 agreements and 2 disagreements resulting in a probability of 6/8, and a bias of -0.25). It is not feasible to brute-force this LFSR due to the lack of any outlying sub-key agreement. The search space is reduced from $O(n^7 * n^{11}) = O(n^{18})$ to $O(n^7 + n^{11}) = O(n^{11}))$ as we now know the key for LFSR1, allowing us to rapidly brute force the sub key for LFSR2 (value 1357, with agreement 0.247).

Finally LFSR3 can be attacked using $R_1 \oplus R_3$, with a Walsh value of -4 (same agreements, disagreements, probability of correlation and bias as previous). As LFSR1's key is known, the search space is also reduced for this attack from $O(n^7 * n^{13}) = O(n^{20})$ to $O(n^7 + n^{13}) = O(n^{13}))$.

This resulted in the sub-key for LFSR3 (value 7531, with agreement 0.245). This LFSR was also not brute-forcible due to a lack of outlying key agreements, as visible in the box-and-whisker plots in appendix A, where the biggest outlying agreement in LSFR3 is significantly closer to the median than the outlying agreement for LSFR1; this suggests a lower likelihood of the outlying key being correct.

Using this strategy, we have successfully attacked all 3 shift registers in a severely reduced search space $(O(n^{7+11+13}) > (O(n^7 + n^{11} + n^{13}) = O(n^{13}))$ (a reduction of 18 bits), resulting in the set of sub-keys: 27, 1357, and 7531.

Attack Implementation

The Python program which implements this attack loads a key-stream from `stream.txt`, and begins by brute forcing the first register. This is achieved by iterating over all possible keys, recording their agreement with the given data, and returning the key with the highest normalized agreement. The next 2 LFSR's are then broken by iterating over each respective set of possible keys in combination with the output of LFSR1 using the previously known key. This results in the set of initial states: `lfsr1: 27, lfsr2: 1357, lfsr3: 7531`.

iii. Improvement

The current function's key weakness is the correlation of a single shift register (LFSR1) with the output: this allows the key to be brute forced in $O(n^7)$, followed by the other two keys in On^{11} and On^{13}. As such, the current function is not correlation immune to any significant order. By changing the combination function's output to 1 0 0 1 0 1 1 0 (two bit flips on L001 and L101), we can ensure the function is correlation immune to order 2, as the Walsh-Hadamard values for R_1, R_2, R_3, $R_1 \oplus R_2$, $R_1 \oplus R_3$ and $R_1 \oplus R_3$, are now 0, and therefore have no exploitable correlation exist other than all 3 registers combined; this is visible in the table below.

Input	f	w
000	1	0
001	0	0
010	0	0
011	1	0
100	0	0
101	1	0
110	1	0
111	0	-8

This would restrict the attack vector to a combination of LFSR1, LFSR2 and LFSR3 simultaneously, with a Walsh-Hadamard value of -8 for $R_1 \oplus R_2 \oplus R_3$; this is acceptable as exploiting this would require iterating over the complete key-set $O(n^{7+11+13}) = O(n^{31})$, causing the time complexity to be significantly higher than the previous $O(n^{13})$).

2 Differential Cryptanalysis

i. Implementation and Challenge

The block cipher found in appendix E (written in Python), can be ran using the Python 2.7.6 compiler with the command `python block.py`. This will produce the challenge output of 45858 (1011001100100010 in binary).

ii. Code description

The block cipher is implemented using the `do_4_rounds` method, which controls the execution of substitutions, permutations or combinations of intermediary results with sub-keys dependent on the round. These sub-methods are implemented in `do_substitution(), permute(), and combine_key()` respectively.

iii. Cryptanalysis

Analysis

Following Hey's tutorial, a differential cryptanalysis attack was undertaken on the S-box. The XOR table of differences was generated using `calculate_xor_profiles()` in appendix H. This produced the difference distribution table found in appendix F. This indicated a number of high probability difference pairs:

$\triangle X$	$\triangle Y$	Difference
1	3	8
5	8	8
8	14	8
12	5	6
13	6	6
4	11	6
9	13	6

Using as many of these high-probability difference pairs as possible a set of high probability paths (which go through a minimal amount of S-boxes) can be used to find pairs of plain-text differences and cipher-text differences. These are illustrated in appendix G, which were picked through trial-and-error to find paths with maximal probability which only impacted a single set of 4 bits, so as to avoid iterating over more than 2^4 bits per attack; as such, 4 consecutive $O(2^4)$ attacks occurred to recover the 16 bit sub key. These paths could also have been obtained using Matsui's algorithm [8].

Using these paths, we were able to find a degree of certainty with which we can brute force the keys for 4-bit clusters to find the sub-key which maximally agrees with the provided plain-text/cipher-text pairs. This can be seen in the table below.

Sub-key bits	$\triangle P$	$\triangle U$	Path probability	Sub-key	Agreement with plain-text/cipher-text pairs
1..4	12	12288	0.0234	0xD	0.2083
5..8	13	768	0.0059	0xD	0.1181
9..11	2	48	0.0078	0xD	0.1627
12..16	17	1	0.0039	0x5	0.0855

As such, we were able to fully extract sub-key K_5 with a value of `0xDDD5`.

Implementation

As detailed above, the difference distribution table found in appendix F was generated using `calculate_xor_profiles()`, which simply iterated over every pair of possible 4 bit differences, and evaluated them against the configuration of the S-box. The list of differential pairs is then printed to console in descending probability of occurrence.
Using the manually-found paths through the substitution/permutation network, the sub-key brute-forcing was implemented in the `crack_section_subkey()` method which can be found in appendix H. It takes in $\triangle P$ and $\triangle U$ values, along with a mask and shift to indicate which group of 4 bits is attacked by the iteration. The data is then loaded from block.txt, and each possible 4-bit sub-key is iterated over each plain-text/cipher-text combination. Another plain-text/cipher-text combination is found by xor-ing the current plain-text with the requested difference, resulting in a second cipher-text. Both cipher-texts are then xor'd with the hypothetical sub-key (shifted to match the correct 4 bit cluster), and masked. They are then passed through the S-box in reverse, before being compared to the expected difference.
The list of potential sub-keys is then ordered by highest agreement, and the most promising key is returned.
This attack is repeated 4 times to attack each 4 bit section of the sub-key, as visible in `main()`.

iv. Improvements

The resilience of the cipher could be improved in two approaches. Firstly, the addition of additional rounds would significantly decrease the likelihood of finding high-probability paths through the substitution-permutation network. This could cause brute-forcing algorithms to produce hypothetical sub-key sections which are proportionally less agreeable to plain-text/cipher-text pairs than erroneous keys, and therefore less liable to have a superior agreement over other sub-key sections. This could however be overcome with the use of additional computational time and a larger set of plain-text/cipher-text pairs to improve the significance of the obtained statistics.
S-boxes present the other significant area for resilience; an S-box's strength can be seen as a combination of it's non-linearity (avoid the possibility of finding linear combinations of inputs which agree with linear combinations of subsets of outputs) and minimal autocorrelation (correlations of inputs which satisfy a difference with outputs). Nyberg [9] details a mapping for S-boxes which are 'differentially uniform', which entails that "for every non-zero input difference and any output difference the number of possible inputs has a uniform upper bound". As such, mappings obeying this property would cause further difficulty in finding biased paths through the substitution-permutation network.
The search for methods to develop maximally non-linear S-boxes have led to the adaption of

bent functions (which were discovered prior to their application to cryptography) to the definition of the strict avalanche criterion (SAC) by Forré [3]. This defines a criteria for maximally non-linear functions, whereby a change in a maximally high number of input bits causes each output bit to change with a 50% probability. A method to construct such functions is details in Forré details a method of constructing SAC-fulfilling functions which flattens the difference distribution table. These methods are further explored by Adams and Tavares [1], who developed a quicker method for generating S-boxes fulfilling this criteria.

3 Timing Analysis

i. Example of a real-world timing analysis attack

[7],

ii. Countering timing attacks

Removing data or key dependent branching

The simplest form of timing attack involves estimating the hamming weight in a key based on the execution time of an encryption algorithm. This is possible due to key-dependent branch execution, resulting in a correlation between cycle execution time and key bit value. An example of this issue occurs in RC5 [4] due to variations in the computational time of rotations. Data-dependent computational requirements also occur in modular exponentiation, which is used by the RSA algorithm. This form of attack can be mitigated by ensuring the algorithm does not conditionally branch, causing it to take a longer but consistent time to execute the encryption.

Noise

Timing attacks can also be mitigating by randomizing the execution time of the algorithm Kocher [6] details a method of decreasing the accuracy of timing measurements by adding random delays to the processing time, causing the need for a larger number of cipher-texts to attack the algorithm. This produces a result similar to bucketing, where increasing the time spent waiting reduces performance but increases security, and vice-versa.

Blinding

Blinding is the obfuscation of data to harden RSA (or similar) algorithms to various side-channel attacks. This involves the encoding of data before and after the execution of attackable sections of code using a bijective function, resulting in unusable information if the attacker uses a standard timing attack as the algorithm's state will be significantly less predictable. Kocher [6] has concluded this to be insufficient to fully mitigate timing attacks due to maliciously-designed modular exponentiation causing timing spikes corresponding to exponent bits, revealing the hamming weight of the exponent.

Balancing

Balancing is a timing attack mitigation implemented in a program by executing every operation on the complement of the data as well as the data. This diminishes the correlation to single data bits [5], resulting in a lack of correlation between hamming weight and timing data. This can be applied to a large number of operations, such as fixed offset shifts, bitwise operations and arithmetic operations.

Bucketing

Bucketing is a method of trading off computational performance while limiting the benefit of timing attack measurements. This was first achieved by Köpf and Dürmuth [7], and consists of a discretization of possible execution times. Bucketing is implemented by partitioning the system's execution times into intervals (called buckets) of variable length, where computations wait until the end of the current bucket's time before returning results of a computation. The effectiveness of this method can be varied to either minimise computational penalty (by reducing bucket size to minimise waiting) or force the attacker to increase the sampling rate used to conduct the timing attack. Köpf and Dürmuth provide an algorithm to find the optimal bucket size and frequency, allowing the program to maximise both it's performance and security. One should note that bucketing would not improve resilience to power-monitoring attacks, as a long period of low power usage would indicate a wait.

Cache timing attacks

Bernstein [2] details an attack vector based on cache hits and misses: this is applicable to data-dependent lookups, for example in the AES algorithm. On the assumption that the attacker can monitor the time taken by the victim to encrypt each character in the input, a split in the location of a stored array (where part is stored on faster cache, and part is stored on slower ram) will reveal which section of the array was requested. In the case of AES, this could indicate which part of the S-box was requested, and therefore which value was imputed to the S-box. This attack vector can be mitigated by removing S-box lookups and instead implementing them using constant time bit operations [2]; this results in a timing attack immune software which is unfortunately much slower than using lookups. Bernstein also notes that maintaining an S-box in cache is not reliably feasible due to lines of the S-box being kicked out of cache by computation other than AES. Similarly, cache-levels can also cause variations in access times for S-boxes which are entirely stored in cache, mitigating the measurable time difference but leaving some information to be attacked.

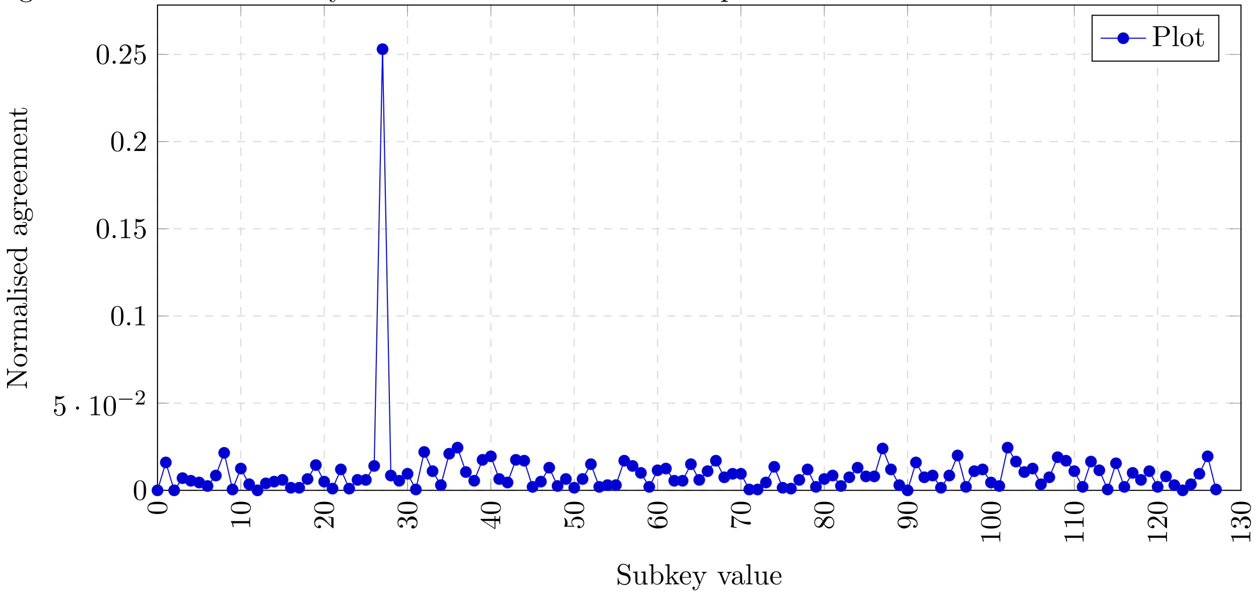
4 Open cryptography

Should the public be allowed to use strong cryptographic algorithms?

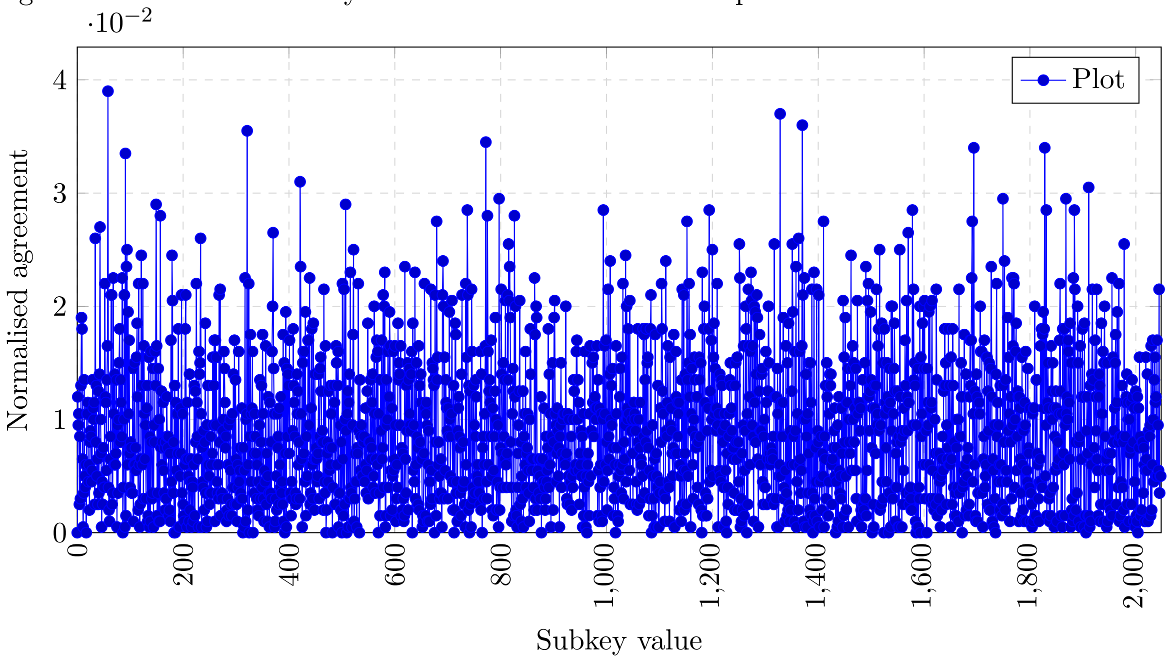
http://web.cs.ucdavis.edu/ rogaway/papers/moral-fn.pdf https://www.technologyreview.com/s/519281/crypto
have-an-ethics-problem/ https://www.cs.kent.ac.uk/people/staff/eab2/talks/Cryptographylawethics.pdf

A LFSR agreements

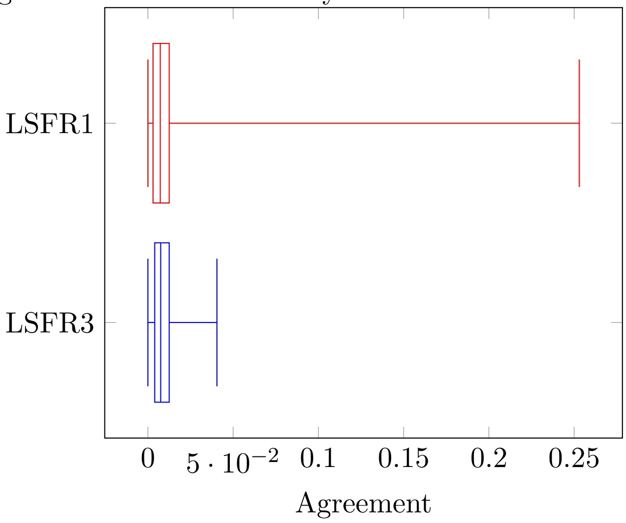
Agreements of 7-bit subkeys in LSFR1 with function output



Agreements of 11-bit subkeys in LSFR2 with function output



Agreements of 13-bit subkeys in LSFR3 with function output



B Stream.py

```
class Lfsr:
    # Initialisation method for a LFSR
    def __init__(self, length, taps, initial_val):
        self.taps = []
        self.length = length
        for tap in taps:
            self.taps.append(tap)
        self.reg_val = initial_val

    # Calculates the tap value for the current LFSR's state
    def calc_tap(self):
        val = self.reg_val[self.taps[0]]
        for x in self.taps[1::]:
            val = (val ^ self.reg_val[x])
        return val

    # Calculate the tap value, shift the register and return the
    # value outputted by the register
    def shift(self):
        new_val = self.calc_tap()
        self.reg_val = [new_val] + self.reg_val
        return self.reg_val.pop()


# Implements the boolean function combining LFSR outputs
def combine_lfsr_outputs(out_1, out_2, out_3):
    val = str(out_1) + str(out_2) + str(out_3)
    val = int(val, 2)
    sub_arr = [1, 1, 0, 1, 0, 0, 1, 0]
    return str(sub_arr[val])


# Static instantiation of every LFSR
arr0 = [1, 1, 0, 0, 0, 0, 1]
l0 = Lfsr(7, [5, 6], arr0)

arr1 = [0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1]
l1 = Lfsr(11, [8, 10], arr1)

arr2 = [1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0]
```

```
l2 = Lfsr(13, [7, 10, 11, 12], arr2)


def main():
    # Shift and combine output of LFSR's for 25 bits
    out = ""
    for x in range(25):
        out_0 = l0.shift()
        out_1 = l1.shift()
        out_2 = l2.shift()
        out += combine_lfsr_outputs(out_0, out_1, out_2)
    print "\nChallenge output: " + str(out)


if __name__ == "__main__":
    # execute only if run as the entry point into the program
    main()
```

C Walsh transform of combining function

Output	L000	L001	L010	L011	L100	L101	L110	L111
000	1	1	1	1	1	1	1	1
001	1	-1	1	-1	1	-1	1	-1
010	1	1	-1	-1	1	1	-1	-1
011	1	-1	-1	1	1	-1	-1	1
100	1	1	1	1	-1	-1	-1	-1
101	1	-1	1	-1	-1	1	-1	1
110	1	1	-1	-1	-1	-1	1	1
111	1	-1	-1	1	-1	1	1	-1
Walsh-Hadamard values:	0	0	0	0	-4	4	-4	-4

D Streamattack.py

```
class Lfsr:

    def __init__(self, length, taps, initial_val):
        self.taps = []
        self.length = length
        for tap in taps:
            self.taps.append(tap)
        self.reg_val = initial_val

    # Calculate the tap output for the current register value
    def calc_tap(self):
        val = self.reg_val[self.taps[0]]
        for x in self.taps[1::]:
            val = (val ^ self.reg_val[x])
        return val

    # Calculate the tap, push the value to the front of the register, and pop
        the last value
    def shift(self):
        new_val = self.calc_tap()
        self.reg_val = [new_val] + self.reg_val
        return self.reg_val.pop()

    def set_reg(self, new_reg_val):
        self.reg_val = new_reg_val

    def get_taps(self):
        return self.taps

    def get_length(self):
        return self.length


# Implements the given boolean function
def combine_lfsr_outputs(out_1, out_2, out_3):
    if out_1 == 0 and out_2 == 0 and out_3 == 0:
        return "1"
    elif out_1 == 0 and out_2 == 0 and out_3 == 1:
        return "1"
    elif out_1 == 0 and out_2 == 1 and out_3 == 0:
        return "0"
    elif out_1 == 0 and out_2 == 1 and out_3 == 1:
        return "1"
    elif out_1 == 1 and out_2 == 0 and out_3 == 0:
        return "0"
    elif out_1 == 1 and out_2 == 0 and out_3 == 1:
        return "0"
    elif out_1 == 1 and out_2 == 1 and out_3 == 0:
        return "1"
    elif out_1 == 1 and out_2 == 1 and out_3 == 1:
        return "0"
    else:
        return ""


# Execute the shift registers and return their combined output for the
    requested number of bits
def test_combined(lfsr1, lfsr2, lfsr3, num_vals):
    out = ""
    for x in range(num_vals):
```

```
        out_1 = lfsr1.shift()
        out_2 = lfsr2.shift()
        out_3 = lfsr3.shift()
        out += combine_lfsr_outputs(out_1, out_2, out_3)
    return out


# Convert a decimal value to a binary array of given length
def dec2bin(length, key_val):
    val = [int(c) for c in '{0:b}'.format(key_val)]
    while len(val) < length:
        val.insert(0, 0)
    return val


# Brute force a single LFSR by finding a subkey with maximal agreement
def crack(lfsr, data):
    length = lfsr.get_length()
    lfsr_max_key = (2 ** length)
    data_len = len(data)
    max_agreement_val = -1.0
    max_agreement_ind = -1
    for x in range(0, lfsr_max_key):
        # Reset LFSR with hypothetical subkey
        lfsr.reg_val = dec2bin(lfsr.length, x)
        output = []
        for y in range(0, data_len):
            output.append(lfsr.shift())
        # Calculate normalised agreement
        agreement = abs(0.5 - calc_agreement(data, output))
        if agreement > max_agreement_val:
            # Store best subkey so far
            max_agreement_val = agreement
            max_agreement_ind = x
    # Return best subkeys
    return max_agreement_ind, max_agreement_val


# Crack an LFSR using a known LFSR (lfsr2) and it's subkey (key_2)
def crack_with(lfsr, lfsr2, key_2, data):
    length = lfsr.get_length()
    lfsr_max_key = (2 ** length)
    data_len = len(data)
    max_agreement_val = -1.0
    max_agreement_ind = -1
    for x in range(1, lfsr_max_key):
        key_val = dec2bin(length, x)
        lfsr.reg_val = key_val[0:length]
        lfsr2.reg_val = key_2
        output = []
        for y in range(0, data_len):
            # XOR output of cracking LFSR with output from second LFSR (which
                has an independently checked key)
            output.append(lfsr.shift() ^ lfsr2.shift())
        agreement = abs(0.5 - calc_agreement(data, output))
        if agreement > max_agreement_val:
            # Store best subkey so far
            max_agreement_val = agreement
            max_agreement_ind = x
    # Return best subkeys
    return max_agreement_ind, max_agreement_val
```

```
# Calculate agreement of output from a single or multiple LFSR (xor'd together)
    with an expected output
def calc_agreement(data1, data2):
    length = len(data1)
    agreements = 0
    for x in range(0, length):
        if data1[x] == data2[x]:
            agreements += 1
    return float(agreements) / float(length)


# Convert a binary array into a decimal value
def bin2dec(arr):
    return int("".join([str(x) for x in arr]), 2)


arr = [0, 1, 0, 1, 1, 0, 0]
l = Lfsr(7, [5, 6], arr)
arr1 = [0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1]
l1 = Lfsr(11, [8, 10], arr1)
arr2 = [0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0]
l2 = Lfsr(13, [7, 10, 11, 12], arr2)

lsrs = [l, l1, l2]


def main():
    # Read ciphertext from file and convert to binary array
    with open('stream.txt', 'r') as myfile:
        data_raw = myfile.read().replace('\n', '').replace(' ', '')
        data = [int(c) for c in data_raw]

        key_l = crack(l, data)[0]
        print "L0: " + str(key_l)

        key_l1 = crack_with(l1, l, dec2bin(l.length, key_l), data)[0]
        print "L1: " + str(key_l1)

        print crack_with(l2, l, dec2bin(l.length, key_l), data)
        key_l2 = crack_with(l2, l, dec2bin(l.length, key_l), data)[0]
        print "L2: " + str(key_l2)


if __name__ == "__main__":
    # execute only if run as the entry point into the program
    main()
```

E Block.py

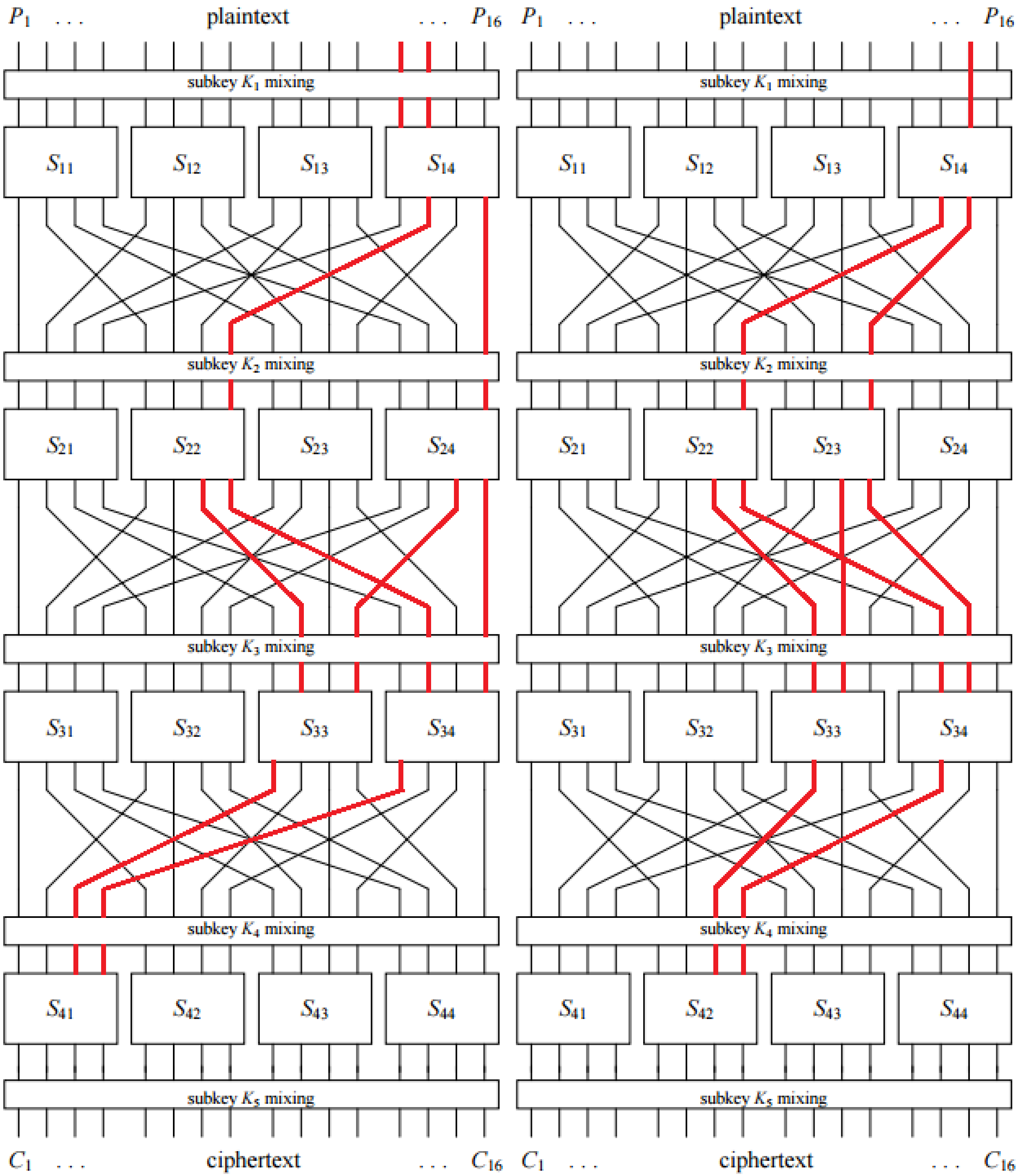
```
# Convert a decimal number into a chunked 16 bit binary number, outputted in an
     array of chunks of 4 bits
def dec2bin_chunks(key_val):
    val = "{0:b}".format(key_val)
    while len(val) < 16:
        val = "0" + val
    n = 4
    return [val[i:i + n] for i in range(0, len(val), n)]


# Convert a decimal to a binary string of minimum length
def dec2bin(val, length):
    out = "{0:0b}".format(val)
    while len(out) < length:
        out = "0" + out
    return out


# Convert a binary value to a decimal
def bin2dec(val):
    return int(val, 2)


# Execute an s_box substitution on a 4 bit input value
def substitute(in_val):
    arr = [0x4, 0x0, 0xC, 0x3, 0x8, 0xB, 0xA, 0x9, 0xD, 0xE, 0x2, 0x7, 0x6, 0x5
       , 0xF, 0x1]
    return arr[int(in_val)]


# Execute an s_box submission for all 16 bits
def do_substitution(val):
    bin_vals = dec2bin_chunks(val)
    sub_vals = [dec2bin(substitute(bin2dec(x)), 4) for x in bin_vals]
    sub_vals = "".join(sub_vals)
    return bin2dec(sub_vals)


# Executes a permutation of the 16 bit value according to the provided network
    structure
def permute(val):
    a = list(dec2bin(val, 16))
    permuted_val = [a[0], a[4], a[8], a[12], a[1], a[5], a[9], a[13], a[2], a
        [6], a[10], a[14], a[3], a[7], a[11],
                    a[15]]
    return bin2dec("".join(permuted_val))


# xor a value with a subkey
def apply_subkey(val, subkey):
    return val ^ subkey


# Execute a single round of subkey mixing, substitution and optionally
    permutation
def do_round(val, subkey, last):
    val = apply_subkey(val, subkey)
    val = do_substitution(val)
    if not last:
        val = permute(val)
    return val
```

```
# Executes all 4 rounds of the algorithm
def do_4_rounds(val, subkeys):
    for x in range(0, 4):
        print "␣"
        val = do_round(val, subkeys[x], x > 2)
        print "Round:␣{0}␣Val:␣{2}␣Subkey:␣{1}␣Last:␣{3}".format(x, subkeys[x],
             val, x > 2)

    val = apply_subkey(val, subkeys[4])
    return val


def main():
    subkeys = [4132, 8165, 14287, 54321, 53124]
    print do_4_rounds(13571, subkeys)


if __name__ == "__main__":
    main()
```

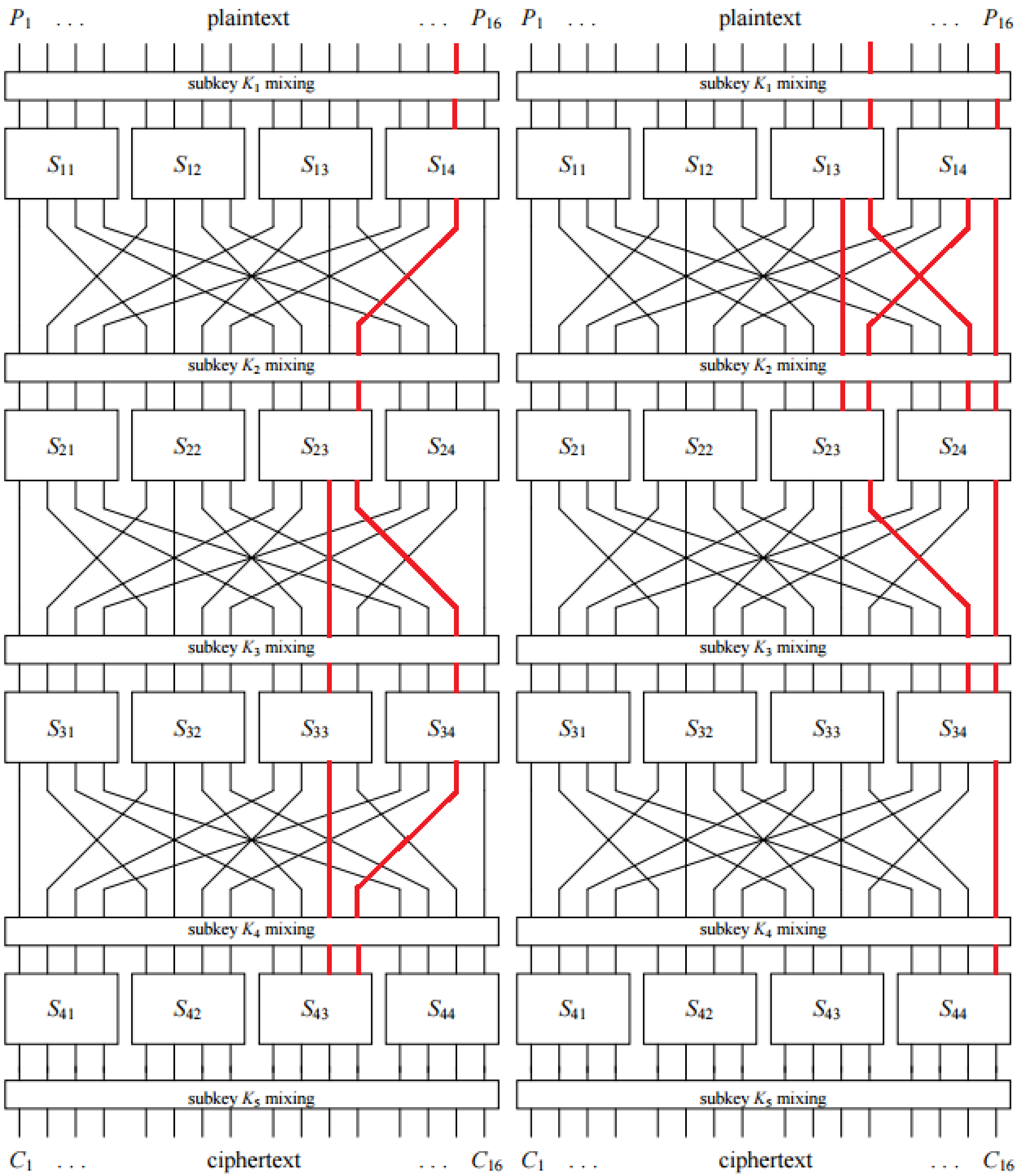
F Difference distribution table

Output Difference	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	8	2	2	0	0	0	0	0	0	0	0	2	2
2	0	0	4	2	2	0	0	0	2	4	0	0	0	0	0	2
3	0	4	0	0	0	0	0	4	0	0	4	0	4	0	0	0
4	0	0	0	0	0	0	4	0	0	0	2	6	2	2	0	0
5	0	0	0	2	0	2	0	0	8	2	0	0	0	0	0	2
6	0	0	4	0	4	0	0	0	2	2	0	0	0	0	2	2
7	0	4	0	0	0	0	0	4	0	0	2	2	2	2	0	0
8	0	0	0	0	2	2	0	0	2	2	0	0	0	0	8	0
9	0	2	0	0	0	0	2	0	0	0	2	4	0	6	0	0
10	0	2	0	0	0	0	2	4	0	0	2	0	4	2	0	0
11	0	0	4	2	2	0	0	0	0	2	0	0	0	0	2	4
12	0	0	4	2	0	6	0	0	2	0	0	0	0	0	2	0
13	0	2	0	0	0	0	6	0	0	0	0	2	2	4	0	0
14	0	2	0	0	0	0	2	4	0	0	4	2	2	0	0	0
15	0	0	0	0	4	4	0	0	0	4	0	0	0	0	0	4

G Paths through substitution-permutation network



The diagrams above demonstrate the path through the substitution-permutation network undertaken to crack the first 8 bits of the subkey, using the differential pairs `(12, 12288)` and `(13, 768)` respectively.



The diagrams above demonstrate the path through the substitution-permutation network undertaken to crack the last 8 bits of the subkey, using the differential pairs `(2, 48)` and `(17, 1)` respectively.

H Blockattack.py

```
# S-box
s_box = [0x4, 0x0, 0xC, 0x3, 0x8, 0xB, 0xA, 0x9, 0xD, 0xE, 0x2, 0x7, 0x6, 0x5,
    0xF, 0x1]

# Generate the reverse s-box to avoid searching through entire array for every
    substitution
reverse_s_box = []
for x in range(16):
    reverse_s_box.append(s_box.index(x))


# Convert a decimal value to an array of 4-bit binary chunks
def dec2bin_chunks(key_val):
    val = "{0:b}".format(key_val)
    while len(val) < 16:
        val = "0" + val
    n = 4
    return [val[i:i + n] for i in range(0, len(val), n)]


# Convert a decimal value to a binary value padded to length
def dec2bin(val, length):
    out = "{0:0b}".format(val)
    while len(out) < length:
        out = "0" + out
    return out


# Convert a binary value to decimal
def bin2dec(val):
    return int(val, 2)


# Execute a reverse s_box substitution for a single 4 bit value
def reverse_substitute(in_val):
    return reverse_s_box[in_val]


# Execute a reverse s_box substitution for the full 16 bit value
def do_reverse_substitution(val):
    bin_vals = dec2bin_chunks(val)
    sub_vals = [dec2bin(reverse_substitute(bin2dec(x)), 4) for x in bin_vals]
    sub_vals = "".join(sub_vals)
    return bin2dec(sub_vals)


# Method used to calculate XOR profile for all differential pairs
def calculate_xor_profiles():
    maxs = 0
    differential_pairs = []
    for x in range(0, 15):
        for y in range(0, 15):
            s = 0
            for z in range(0, 15):
                if (s_box[z] ^ s_box[z ^ x]) == y:
                    s += 1
            if not (x == 0 and y == 0):
                if maxs < s:
                    maxs = s
            differential_pairs.append((x, y, s))
    differential_pairs.sort(key=lambda x: x[2], reverse=True)
```

```
    print differential_pairs


# Method to find most agreeable subkey for a given s-box, plaintext difference
    and ciphertext difference pair
def crack_section_subkey(plain_diff, cipher_diff, shift):
    mask = 0xF << shift
    vals = {}
    subkeys = []
    with open('block.txt', 'r') as myfile:
        for line in myfile:
            (key, val) = line.split()
            vals[int(key)] = int(val)
        for subkey in range(0, 15):
            prob = 0
            for plaintext in vals:
                cipher1 = vals[plaintext]
                cipher2 = vals[plaintext ^ plain_diff]
                # Shift subkey to correct index, xor to cipher text and mask to
                     find agreement over only those 4 bits
                cipher1 = (cipher1 ^ (subkey << shift)) & mask
                cipher2 = (cipher2 ^ (subkey << shift)) & mask

                cipher1 = do_reverse_substitution(cipher1)
                cipher2 = do_reverse_substitution(cipher2)
                if (cipher1 ^ cipher2) == (cipher_diff & mask):
                    prob += 1
            prob = float(prob) / len(vals)
            subkeys.append((subkey, prob))
    subkeys.sort(key=lambda x: x[1], reverse=True)
    return subkeys[0]


def main():
    crack = crack_section_subkey(12, 12288, 12)
    print "Bits␣1..4:␣value:␣␣␣␣{0},␣␣␣␣prob:␣␣{1}".format(crack[0], crack[1])
    crack = crack_section_subkey(13, 768, 8)
    print "Bits␣5..8:␣value:␣␣␣␣{0},␣␣␣␣prob:␣␣{1}".format(crack[0], crack[1])
    crack = crack_section_subkey(2, 48, 4)
    print "Bits␣9..12:␣value:␣␣␣{0},␣␣␣␣prob:␣␣{1}".format(crack[0], crack[1])
    crack = crack_section_subkey(17, 1, 0)
    print "Bits␣13..16:␣value:␣␣{0},␣␣␣␣prob:␣␣{1}".format(crack[0], crack[1])


if __name__ == "__main__":
    main()
```

Bibliography

[1] C. Adams and S. Tavares, "The structured design of cryptographically good s-boxes," *J. Cryptology*, vol. 3, no. 1, pp. 27–41, 1 Jan. 1990.

[2] D. J. Bernstein, "Cache-timing attacks on aes," The University of Illinois at Chicago, Tech. Rep., 2005.

[3] R. Forré, "The strict avalanche criterion: Spectral properties of boolean functions and an extended definition," in *Proceedings on Advances in Cryptology*, ser. CRYPTO '88. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 450–468.

[4] H. Handschuh and H. M. Heys, *A Timing Attack on RC5.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 306–318. [Online]. Available: http://dx.doi.org/10.1007/3-540-48892-8_24

[5] V. R. Joan Daemen, "Resistance against implementation attacks, a comparative study of the aes proposals," Tech. Rep., 1999.

[6] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology CRYPTO 96.* Springer-Verlag, 1996, pp. 104–113.

[7] B. Köpf and M. Dürmuth, "A provably secure and efficient countermeasure against timing attacks," in *2009 22nd IEEE Computer Security Foundations Symposium*, jul 2009, pp. 324–335.

[8] M. Matsui, "On correlation between the order of s-boxes and the strength of DES," in *Lecture Notes in Computer Science no. 950.* Springer-Verlag, 1995, pp. 366–375.

[9] K. Nyberg, "Differentially uniform mappings for cryptography," pp. 55–64.