CS 463/563 — Cryptography for Cybersecurity (Spring 2026)
Homework 3: Symmetric Cryptography

Sunday Feb 8th 2026

Joshua Bodie
01257160
463
Instructor: J. Takeshita. Institution: Old Dominion University.

This assignment has **3** questions.

Submission instructions: If you don't already have the necessary tools, then install 1) a LaTeX distribution (probably the `texlive-full` package on most Linux distributions) and 2) an IDE or other text editor suitable for LaTeX. For the latter, I suggest the use of the Kile IDE, which is available in most Linux distributions' default packages. You can also just use a web IDE such as **Overleaf**.

Download the source file(s) for this assignment. Edit the file(s) to include your solutions using the text editor or LaTeX IDE of your choice. Type your answers and personal information (e.g., name, section, etc.) into the source file at the appropriate places, and double-check that the variable `solutions` is set to be false. (I should have done this for you, but it's good to be sure.) Make sure to properly format your math! See `https://latex-tutorial.com/tutorials/` if you're not already familiar with LaTeX.

Compile the source to a PDF, either through your IDE or by using the command `pdflatex`. Make sure your compilation has no errors, and address all reasonably actionable warnings (you can ignore small things like badbox warnings, as long as they don't affect the output badly).

Turn in 1) the PDF and 2) any and all `.tex` source files, `.bib` bibliography files, source code, or other artifacts. Submissions that are handwritten or typeset with other methods will not be accepted! If you wrote any software in the course of this work, please include it using the `listing` package's utilities.

Cite any external sources, and include any code you write. The use of unauthorized outside assistance, including any AI/LLM, is strictly prohibited.

A note: While it is possible and allowed to solve these questions by writing your own computer programs, you should be able to also do them by hand, as similar questions may appear on exams.

Questions:

1. Recall that a linearly congruent generator is defined by $s_{i+1} = a \cdot s_i + b \pmod{m}$. Given a seed $s_0 = 1$ and parameters $a = 3$, $b = 4$, $m = 13$, use the linearly congruent generator to find the first ten values generated after s_0, i.e., the values $s_1, \cdots, s_{10}$. (10 points)

Answer: a=3, b=4, m=13

$S_1 = a \cdot S_0 + b \pmod{m}$

$S_1 = 3 \cdot 1 + 4 \pmod{13}$

$S_1 = 7 \pmod{13}$

$S_1 = 7$

$S_2 = a \cdot S_1 + b \pmod{m}$

$S_2 = 3 \cdot 7 + 4 \pmod{13}$

$S_2 = 25 \pmod{13}$

$S_2 = 12$

$S_3 = a \cdot S_2 + b \pmod{m}$

$S_3 = 3 \cdot 12 + 4 \pmod{13}$

$S_3 = 36 + 4 \pmod{13}$

$S_3 = 40 \pmod{13}$

$S_3 = 1$

$S_4 = a \cdot S_3 + b \pmod{m}$

$S_4 = 3 \cdot 1 + 4 \pmod{13}$

$S_4 = 7 \pmod{13}$

$S_4 = 7$

$S_5 = a \cdot S_4 + b \pmod{m}$

$S_5 = 3 \cdot 7 + 4 \pmod{13}$

$S_5 = 25 \pmod{13}$

$S_5 = 12$

$S_6 = a \cdot S_5 + b \pmod{m}$

$S_6 = 3 \cdot 12 + 4 \pmod{13}$

$S_6 = 36 + 4 \pmod{13}$

$S_6 = 40 \pmod{13}$

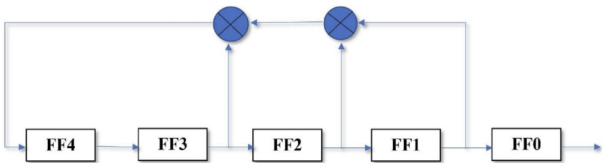

Figure 1: Showing lsfr

$S_6 = 1$
$S_7 = a \cdot S_6 + b \pmod m$
$S_7 = 3 \cdot 1 + 4 \pmod{13}$
$S_7 = 7 \pmod{13}$
$S_7 = 7$
$S_8 = a \cdot S_7 + b \pmod m$
$S_8 = 3 \cdot 7 + 4 \pmod{13}$
$S_8 = 25 \pmod{13}$
$S_8 = 12$
$S_9 = a \cdot S_8 + b \pmod m$
$S_9 = 3 \cdot 12 + 4 \pmod{13}$
$S_9 = 36 + 4 \pmod{13}$
$S_9 = 40 \pmod{13}$
$S_9 = 1$
$S_{10} = a \cdot S_9 + b \pmod m$
$S_{10} = 3 \cdot 1 + 4 \pmod{13}$
$S_{10} = 7 \pmod{13}$
$S_{10} = 7$

2. Consider the LSFR with $m = 5$ shown in **??**. (Note that this figure uses $\otimes$ to represent XOR gates; the usual notation for an XOR gate or operation is $\oplus$.) Suppose the LSFR is initialized with the values $FF4 = 0$, $FF3 = 0$, $FF2 = 1$, $FF1 = 1$, $FF0 = 1$. Give the first 30 bits generated from this LSFR, and determine the length of its period. (10 points)

Answer: First 30 bits= 111000010100110111000010100110 1 Length of period =14 bits.

CLK	FF4	FF3⊗	FF2	FF1 ⊗	FF0⊗
0	0	0	1	1	1
1	0	0	0	1	1
2	0	0	0	0	1
3	1	0	0	0	0
4	0	1	0	0	0
5	1	0	1	0	0
6	0	1	0	1	0
7	0	0	1	0	1
8	1	0	0	1	0
9	1	1	0	0	1
10	0	1	1	0	0
11	1	0	1	1	0
12	1	1	0	1	1
13	1	1	1	0	1
14	0	1	1	1	0
15	0	0	1	1	1
16	0	0	0	1	1
17	0	0	0	0	1
18	1	0	0	0	0
19	0	1	0	0	0
20	1	0	1	0	0
21	0	1	0	1	0
22	0	0	1	0	1
23	1	0	0	1	0
24	1	1	0	0	1
25	0	1	1	0	0
26	1	0	1	1	0
27	1	1	0	1	1
28	1	1	1	0	1
29	0	1	1	1	0
30	0	0	1	1	1

Table 1: 5-bit Counter State Table (0–30)

3. Consider the DES f-function. For a 32-bit input to this function `0x860b2de2` and a 48-bit subkey `0xad8e4d3dff36`,

determine the 32-bit output and give it in hexadecimal. (20 points)

Answer:

DES f-function Calculation

Step 1 Expand R from 32 bits to 48-bits using DEs expansion

Binary conversions from hexadecimal to binary for R and K (Grouped in 4 Bits)

Input function in Hex= 0x860b2de2 Binary= 1000 0110 0000 1011 0010 1101 1110 0010

Now convert 32 bit to 48 bit by using right most bit of left neighbor and left most bit of right neighbor of each 4 bit chunk. This gives us 010000 000110 000001 010110 100101 011011 111100 000101

48 -bit sub key: 0xad8e4d3dff36 in binary 101011 011000 111001 001101 001111 011111 111100 110110

Now perform XOR On 48 bit input function and 48 bit sub key.

010000 000110 000001 010110 100101 011011 111100 000101

$\otimes$

101011 011000 111001 001101 001111 011111 111100 110110

Result =111011011110111000011011101010000100000000110011

Now compute S1- s8 substitution for each 6 bit block S1- S8

DES S-Box Substitution (S1–S8)

48-bit Input (After XOR)

111011 011110 111000 011011 101010 000100 000000 110011

S-Box Computations

S1: 111011

Row $= 11_2 = 3$
Column $= 1101_2 = 13$

$$S1[3][13] = 0 \Rightarrow 0000$$

S2: 011110

Row $= 00_2 = 0$
Column $= 1111_2 = 15$

$$S2[0][15] = 10 \Rightarrow 1010$$

S3: 111000

Row $= 10_2 = 2$
Column $= 1100_2 = 12$

$$S3[2][12] = 5 \Rightarrow 0101$$

S4: 011011

Row $= 01_2 = 1$
Column $= 1101_2 = 13$

$$S4[1][13] = 6 \Rightarrow 0110$$

S5: 101010

Row $= 10_2 = 2$
Column $= 0101_2 = 5$

$$S5[2][5] = 4 \Rightarrow 0100$$

S6: 000100

Row = $00_2 = 0$
Column = $0010_2 = 2$

$$S6[0][2] = 10 \Rightarrow 1010$$

S7: 000000

Row = $00_2 = 0$
Column = $0000_2 = 0$

$$S7[0][0] = 4 \Rightarrow 0100$$

S8: 110011

Row = $11_2 = 3$
Column = $1001_2 = 9$

$$S8[3][9] = 12 \Rightarrow 1100$$

Combine 4-bit Outputs

0000 1010 0101 0110 0100 1010 0100 1100

Final 32-bit Output

$$\boxed{00001010010101100100101001001100}$$

Hex Representation

$$\boxed{0x0A564A4C}$$

DES P-Permutation Step

Input to P (Output of S-Boxes)

$$S = 00001010010101100100101001001100$$

$$S = \texttt{0x0A564A4C}$$

P-Permutation Table

$$P = [16, 7, 20, 21,\ 29, 12, 28, 17,\ 1, 15, 23, 26,\ 5, 18, 31, 10,\ 2, 8, 24, 14,$$

32,27,3,9, 19,13,30,6, 22,11,4,25]

Output After P-Permutation

$$P = 01011100011111010001000000100000$$

$$\boxed{P} = \texttt{0x5C7D1020}$$

4. Answer the following questions on pseudorandom functions (PRFs) and pseudorandom generators (PRGs). (15 points)
   (a) Why are PRGs frequently used, instead of sources of "true" randomness (e.g., reading bytes from `/dev/random`)?
   (b) Why are PRFs frequently used instead of truly random functions?
   (c) Consider the function $F_k(m) = k \cdot m \pmod{p}$, for a publicly known parameter p. Is this function a PRF? Justify your answer[1].

[1]This phrase usually means "provide a proof or counterexample", but a well-reasoned argument will

Answer:

(a) Pseudorandom generators (PRGs), especially cryptographically secure ones (CSPRNGs), are used instead of true randomness sources like /dev/random because they are faster, more practical, and still secure. True randomness is slow to collect and limited in quantity, and sources like /dev/random can block when the system runs out of entropy. This can cause delays or even application failures.

In contrast, a PRG is seeded once with a small amount of true entropy and can then generate large amounts of random-looking data very efficiently. As long as the seed remains secret, the output of a secure PRG is computationally indistinguishable from true randomness.

For this reason, modern systems use true randomness mainly to seed a PRG, and rely on the PRG for ongoing cryptographic operations such as key generation, nonces, and session tokens.

(b) Pseudorandom functions (PRFs) are used instead of truly random functions because truly random functions are not practical to implement. A truly random function would require storing a completely random output for every possible input, which is infeasible due to the enormous input space (e.g., 2^{12} possible inputs). This would require an impossible amount of memory and coordination.

In contrast, a PRF is an efficient, deterministic algorithm that uses a secret key to produce outputs that are computationally indistinguishable from those of a truly random function. As long as the key remains secret, an attacker cannot predict outputs or distinguish the PRF from a random function within feasible computational limits.

PRFs are practical, efficient, and secure for real-world cryptographic applications such as message authentication codes (MACs), key derivation functions, and TLS protocols. They provide the security benefits of randomness without the impractical storage and implementation requirements of a true random function.

(c) **Is $F_k(m) = k \cdot m \pmod{p}$ a PRF?**

No, $F_k(m) = k \cdot m \pmod{p}$ is not a pseudorandom function (PRF).

suffice in lieu of a full proof in the affirmative case – proofs of pseudorandomness are quite difficult.

A PRF must be computationally indistinguishable from a truly random function. However, this function has clear algebraic structure. For example, if an attacker queries $m = 0$, the output is always:

$$F_k(0) = k \cdot 0 \equiv 0 \pmod{p}.$$

A truly random function would output 0 only with probability $1/p$, so this behavior easily distinguishes F_k from a random function.

Additionally, the function is linear:

$$F_k(m_1) + F_k(m_2) \equiv F_k(m_1 + m_2) \pmod{p}.$$

A truly random function would not consistently satisfy such a property.

Because it is predictable and highly structured, $F_k(m)$ is easily distinguishable from a random function and therefore is not a PRF.

5. Answer the following questions. (5 points)
   (a) For a linearly congruent generator parameterized by s_0, a, b, m, how many outputs can be generated before a cycle occurs?
   (b) For a LSFR parameterized by m, a set of up to m taps, and an initial state of m bits, how many outputs can be generated before a cycle occurs?
   (c) What is the common weakness in linearly congruent generators and LSFRs? How can this become a cryptographic vulnerability when one of these generators is used for encryption?

Answer:

(a) The maximum number of distinct outputs before the sequence repeats (i.e., before a cycle occurs) is m.

(b) An m-bit LFSR can generate at most $2^m - 1$ outputs before repeating.

(c) The common weakness of LCGs and LFSRs is that they are linear and therefore predictable. If used for encryption, an attacker who observes enough output can recover the internal state and predict future keystream values, allowing them to decrypt ciphertext.

6. Give a new question on the topic of this week's material and its solution, suitable for inclusion in future versions of this homework assignment. Your question should

be different from the other questions here – don't just change some values around! You can use textbooks, scholarly papers, or other reputable sources of information for inspiration, but you should not directly copy someone else's work. Your question should show me that you understand the content well enough to ask and answer an interesting question about it. Include any source code used to generate/solve the problem.

Some free online textbooks include *Understanding Cryptography* by Paar et al., *The Joy of Cryptography* by Rosulek, or *A Graduate Course in Applied Cryptography* by Boneh and Shoup.

Possible topics include the one-time pad, stream ciphers, LSFRs, or DES.

Answer:

New Homework Question: LFSR vs. Nonlinear Combiner

Question

Consider the following two keystream generators:

1. A single 5-bit LFSR with primitive feedback polynomial.
2. A nonlinear combiner generator that XORs the outputs of three independent 5-bit LFSRs and then applies the Boolean function

$$f(x_1, x_2, x_3) = (x_1 \wedge x_2) \oplus x_3.$$

1. What is the maximum possible period of the single 5-bit LFSR?
2. What is the maximum possible period of the nonlinear combiner generator (assuming the three LFSRs are maximal and pairwise relatively prime)?
3. Explain why the nonlinear combiner is more resistant to attacks such as Berlekamp–Massey than a single LFSR.

Solution

1. A 5-bit LFSR has at most $2^5 - 1 = 31$ states. Therefore, the maximum period is:
$$2^5 - 1 = 31.$$

2. If each of the three 5-bit LFSRs is maximal (period 31) and their periods are relatively prime, the combined period is the least common multiple:
$$\text{lcm}(31, 31, 31) = 31.$$
However, since their internal states are independent, the total state space is:
$$(2^5 - 1)^3 = 31^3 = 29791.$$
Thus the maximum possible period of the nonlinear combiner generator is 29791.

3. A single LFSR is linear over $\mathbb{F}_2$. Its output sequence can be recovered efficiently using the Berlekamp–Massey algorithm after observing approximately $2m$ bits.
In contrast, the nonlinear combiner applies a nonlinear Boolean function to multiple LFSR outputs. Because the output is no longer a linear function of the internal states, linear algebra techniques such as Berlekamp–Massey cannot directly recover the combined generator. This increases resistance to linear cryptanalysis and makes state recovery more difficult.