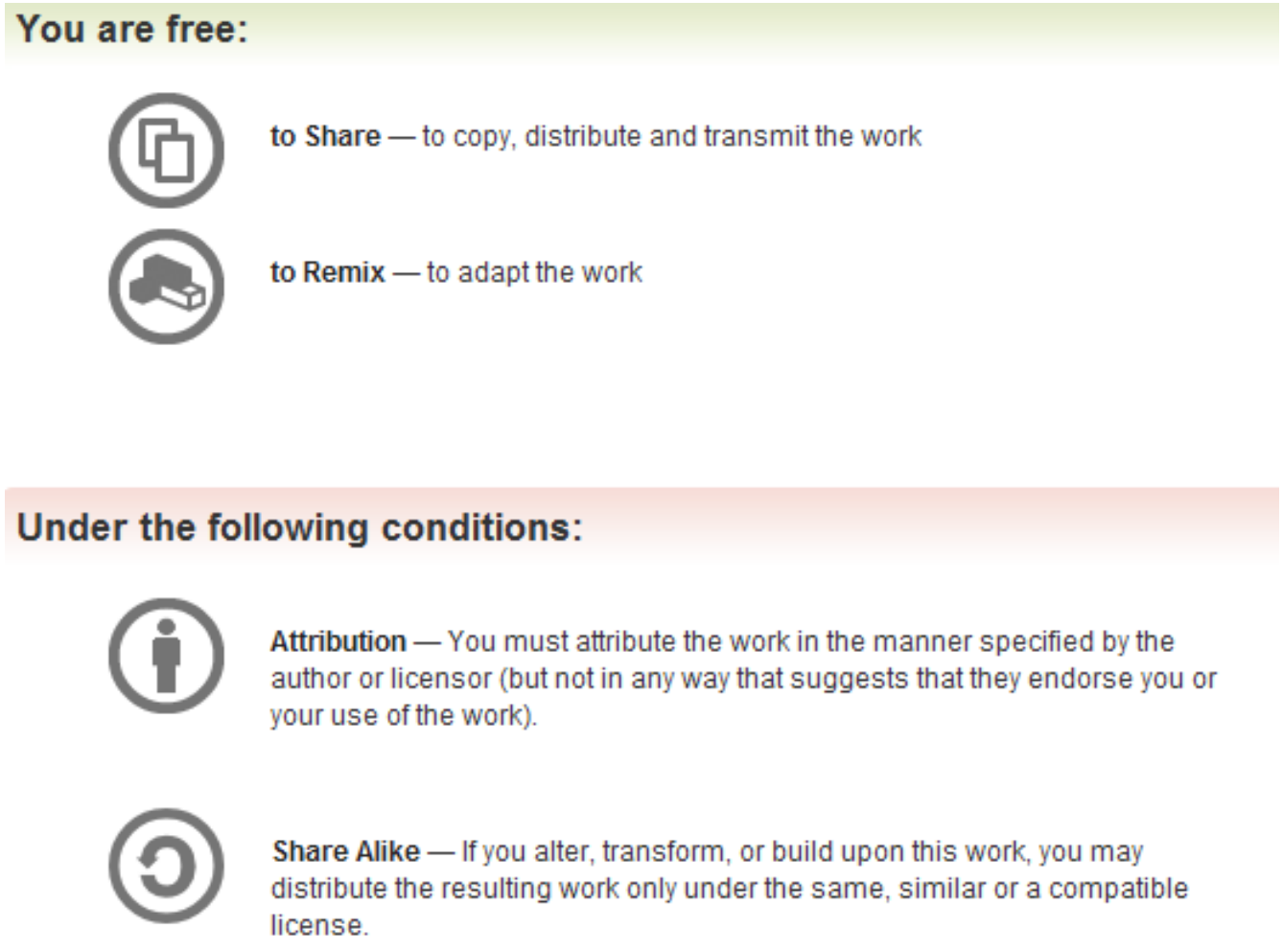
Understanding Cryptology: Cryptanalysis

Instructor name: Dr. Kerry A. McKay

Date of most recent change: 4/23/13

All materials is licensed under a Creative Commons “Share Alike” license.

- http://creativecommons.org/licenses/by-sa/3.0/



Outline

- Intro to this course
- Human-computable crypto
- Number theory and abstract algebra primer
- Factoring attacks
- Attacks on RSA
- Discrete logarithm attacks
- Symmetric system constructions
- Generic attacks
- Linear cryptanalysis
- Differential cryptanalysis
- Integral cryptanalysis on reduced AES
- Conclusions and closing remarks

Goals

- Learn about different techniques that are used to attack modern cryptosystems
  - Focus on deterministic algorithms
  - Implementation agnostic
- At the end of this course, you will
  - Understand foundational mathematics that drive crypto designs and attacks, including topics in number theory, abstract algebra, linear algebra, and probability
  - Be able to identify appropriate methods of analysis based on algorithm class and properties
- This does not mean that you will be able to break everything at the end of this course
  - But you will have a better idea of what you can and cannot do
  - You will have a better understanding what the bad guy can do
- But if I won't be able to break things, why am I here?

The real story

"In theory, theory and practice are the same. In practice, they are not." (thank you Einstein)

- Cryptologists work very hard to analyze systems in theoretical frameworks
  - Working under certain assumptions
- Never underestimate people's ability to do it wrong in practice
  - Poor parameter choices
  - Sacrifice something for better performance
  - Dependency on biased random number generator
  - Reuse of things that should never be reused
  - In general, break the assumptions under which the system was shown to be sufficiently secure
- The "custom" crypto/obfuscation technique
  - Often weaker than their well-studied counterparts

The bottom line

- You may be able to find the key or message in some scenarios
- You will be able to better assess
  - Your needs when considering algorithms
  - Where a system's security is lacking with respect to crypto

What to expect

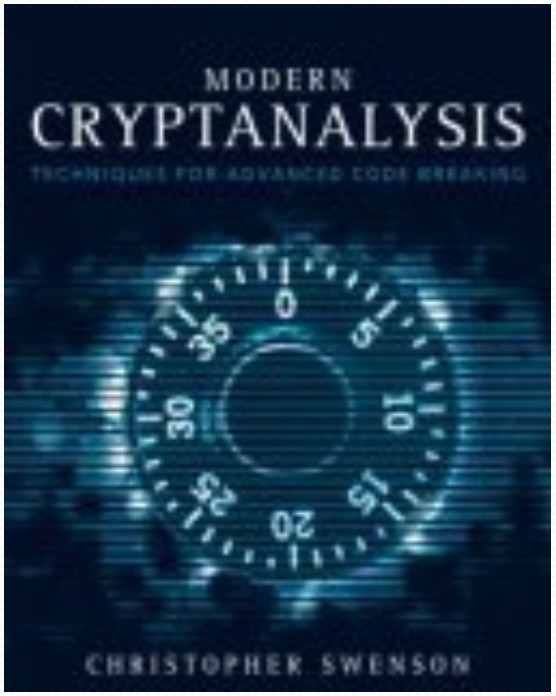
- We'll start with a some human computable ciphers
- Next we'll jump into a brief math background
  - Hours of undergrad math condensed into a few slides
  - If you need your coffee to jump start your brain in the morning, you may want to fill that cup
- We'll be focusing mainly on generic methods of analysis
  - Widest application
- There will be specific attacks as well
  - RSA is widely used and will be for a long time
    - Can keep increasing key length without changing algorithm design
  - Attack on reduced AES
    - This won't work on the full version

When in doubt

- If you have a question, please ask ☺
  - I have a tendency to drive right on if I get no feedback
    - Sometimes I forget that not everyone has the same set of knowledge as myself, and what I say only makes sense to me
    - Everyone knows Fermat's little theorem, right?
      - No?
      - Riiiight… I know that because I've been studying cryptology for years
- Discussion aids the learning process and is encouraged

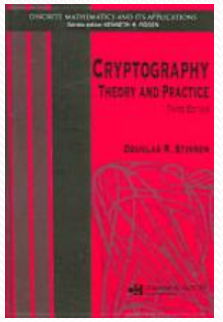
Textbook for this course

- Modern Cryptanalysis: Techniques for Advanced Code Breaking
  - Swenson
- Errata available at http://www.caswenson.com/mcerrata
- This is a good book, and covers most of the material

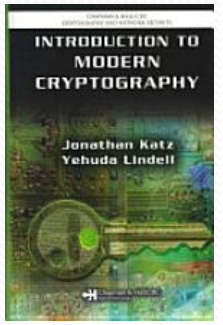


Further reference

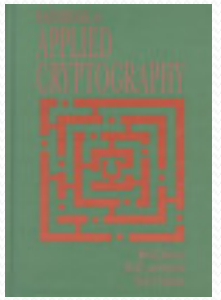
- Cryptography: Theory and Practice
  - Stinson



- Introduction to Modern Cryptography
  - Katz, Lindell

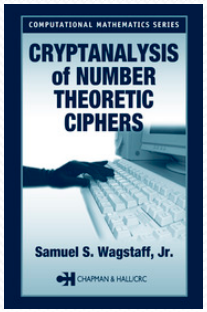


- Handbook of Applied Cryptography
  - Menezes, van Oorschot, Vanstone
  - The latest and greatest (through 1996)
  - Download chapters free online at http://cacr.uwaterloo.ca/hac/ (read the copyright notice)
  - Can fill in gaps of Swenson's book
  - We'll refer to it as "HAC" in this course

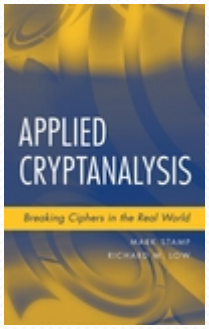


Further reference (continued)

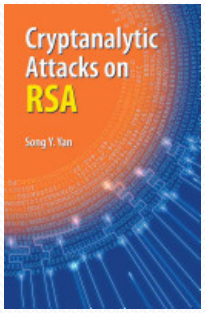
- Cryptanalysis of Number Theoretic Ciphers
  - Wagstaff



- Applied Cryptanalysis
  - Stamp and Low



- Attacks on RSA
  - Yan



Approximate Agenda

- Day 1
  - Human-computable crypto
  - Number theory and abstract algebra primer
  - Factoring attacks
  - Attacks on RSA
- Day 2
  - Discrete logarithm attacks
  - Symmetric system constructions
  - Generic attacks
  - Linear cryptanalysis
- Day 3
  - Differential cryptanalysis
  - Integral cryptanalysis on reduced AES
  - Conclusions and closing remarks

Subject to change based on how quickly folks get through the exercises

The math

- In Core Concepts, we took out as much math as possible to make it more accessible
- We couldn't do that here
- Still tried to keep it accessible
  - Not in theorem and proof format
  - We will not go into excruciating detail*
- Some of the math is essential to understanding
- Some of the math is there to support that a technique has a reason to work
  - Technique can be used without full understanding of why it works

* For some definition of "excruciating detail"

Human-computable crypto

It is always good to start with the basics

Caesar cipher

- Caesar cipher is a monoalphabetic cipher
  - Replace each symbol of the plaintext with a symbol of ciphertext using a single new alphabet
- Suppose each letter is mapped to an integer index
  - A=0, B=1, …., Z=25
  - Encoding becomes a modular addition
  - Rotate indices by a constant integer, *x*
  - Example: D + Y = 3 + 24 = 27 ≡ 1 (mod 26)
    - Result is B
- Decoding is the inverse
  - If encryption key is Y, decryption key is 26-Y
    - Decryption key is 2, or C
    - B + C = 1 + 2 = 3
    - Result is D, as expected

An alternative encoding approach

- Can also just line up a plaintext alphabet and ciphertext alphabet
  - Replace character of plaintext alphabet with ciphertext character at the same position
- Example
  - Plaintext: ABCDEFGHIJKLMNOPQRSTUVWXYZ
  - Ciphertext: FGHIJKLMNOPQRSTUVWXYZABCDE
  - HELLO becomes MJQQT

- Both methods are equivalent
  - One takes more time
    - Negligible
  - The other takes more memory
    - Need to store two alphabet strings
  - This is going to be a running theme

Cryptanalysis of Caesar cipher

- Easily breakable by anyone who knows the cipher algorithm
  - Worked OK back in the day
    - High illiteracy rate
    - Lack of algorithm knowledge
- Attacks
  - Brute force
    - Try every possible key
    - Always an option
    - For a 26-letter alphabet, only 26 possible values for *x*
      - That's doable
  - Frequency analysis

Frequency analysis

- In every language, symbols occur with different probabilities
- Frequency analysis looks at how often each is seen in a sample
- Match frequency in ciphertext to frequency in plaintext
  - Gives a short list of possible mappings

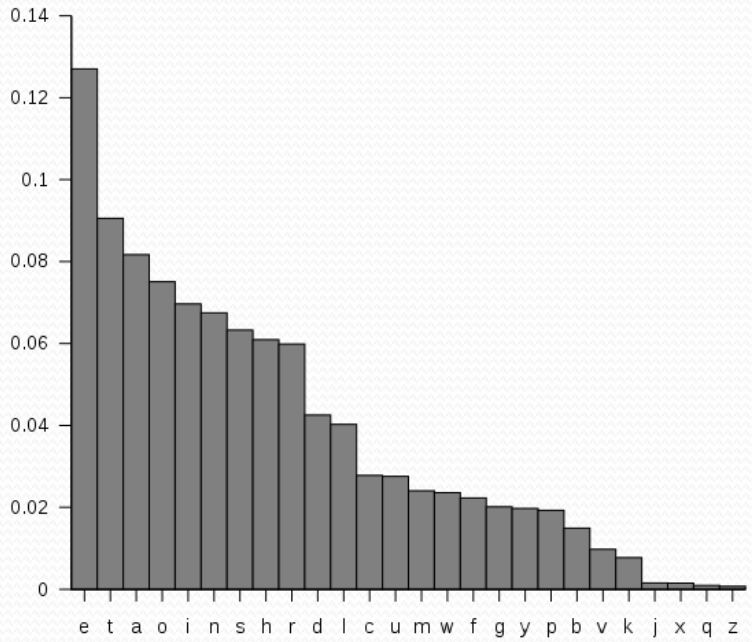


Image from: http://upload.wikimedia.org/wikipedia/commons/b/bo/English_letter_frequency_%28frequency%29.svg

Example

- Suppose you have a message encoded with the Caesar cipher
- You count the number of times each symbol appears, and compute percentage
- What letter goes with what?

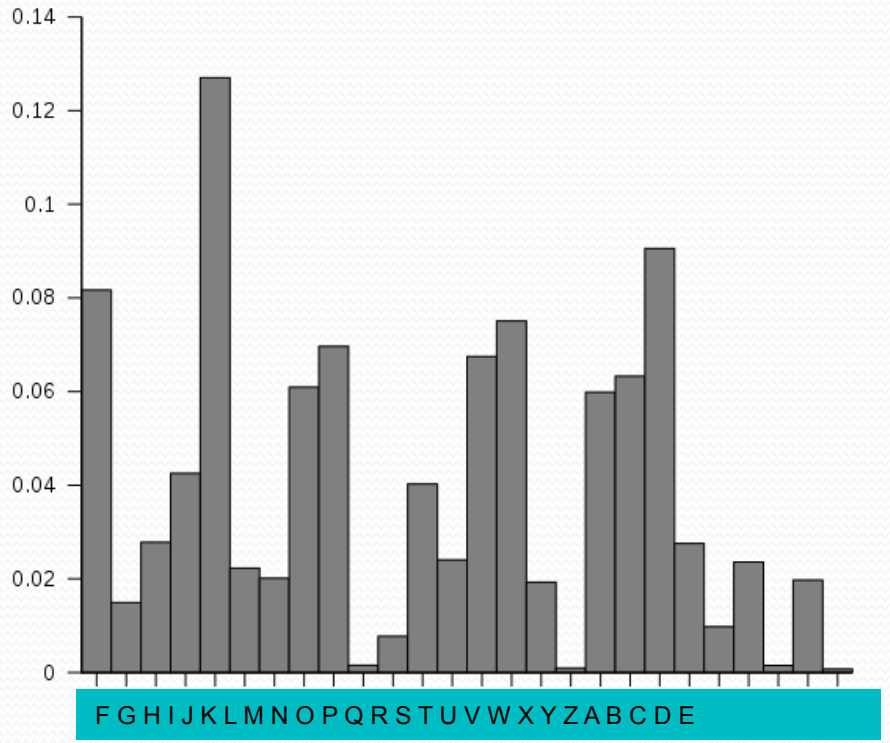


Image from: http://upload.wikimedia.org/wikipedia/commons/d/d5/English_letter_frequency_%28alphabetic%29.svg

Exercise

- Time to get down to some python
- Partial code (and completed) code available in my transfer folder
  - Partial code has it started for you
  - Provided code is not always efficient
    - Goal was understanding, not efficient programming
- Copy the folder “exercises” to your local machine
- Right-click on caesar.py and select “Edit with IDLE” to open

Everything you need to know about python (for this course)

- Right-click on pythonCrashCourse.py and select “Edit with IDLE”
  - Opens file in a python editor
  - Press F5 to run your code
- Python structure is dictated by whitespace
  - No braces
  - Code with same indentation is at the same level
  - Loop/conditional body needs to be indented

- pythonCrashCourse.py contains examples of:
  - For and while loops
  - If statement
  - Print statements
  - List manipulation
  - Random integers

IDLE

- IDLE tips
  - To indent (shift right) a block of code, highlight and press ctrl+]
  - To dedent (shift left) a block of code, highlight and press ctrl+[
  - To comment a block of code, highlight and press alt+3, or got to format menu
  - To uncomment a block of code, highlight and press alt +4, or got to format menu

Lab 1: Caesar cipher

- Objective: Attempt to find the plaintext by
  - Frequency analysis
  - Brute force
- You'll need to fill in the following functions
  - frequency
  - relativeFrequency
  - decrypt
- Finished early?
  - Write an encryption routine as well
  - Encrypt different messages, and see how the frequency analysis changes with the length of the message

Polyalphabetic ciphers

- Monoalphabetic cipher applies the same key to every symbol
- Polyalphabetic cipher switches between a set of keys
  - The next step up from monoalphabetic systems like Caesar's cipher
- We'll look at the Vigenère Tableau
  - Symbols are changed exact same way as Caesar's cipher
  - Difference is that there are multiple key symbols

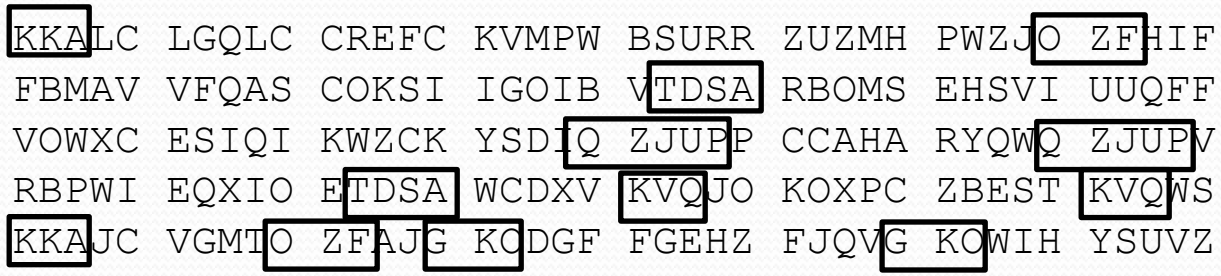
Vigenère Tableau

- There is a table on page 8 of the text
- If symbols are represented numerically, can condense into a simple expression
  - P[i] + (k[i mod len(k)])
    - The second part selects the correct key symbol to use
  - If you do this, watch out for the spaces
    - If they aren't in the alphabet, they should be ignored
- Example:
  - Plaintext: "the quick brown roman fox jumped over the lazy ostrogoth dog"
  - Key: "caesar"
  - Result: "vhi iuzek fjonp rseae hob budreh gvvt tlw lrby sktiqgslh uqg"
- Spaces may preserve plaintext word length, or may occur at fixed intervals to obscure word length

Attacking the Tableau

- Step 1 is to figure out the key length, n
- Look for patterns
- Common words are likely to be encrypted multiple times if the text is long enough
  - "the" is very common in English
  - If there are at least n occurrences of "the" in the plaintext, we can expect at least 2 to have identical ciphertext
- When you find two words of ciphertext that you believe to encrypt the same ciphertext
  - Find the difference in position, d
    - n|d
  - Repeat and narrow in on n by looking for common factors

Example



- Spaces occur at fixed intervals
- Look for any repeated groupings
  - KKA (0,160)
  - OZF (34,169)
  - TDSA (61,131)
  - QZJUP (99,114)
  - KVQ (140,155)
  - GKO (174,189)

Example (continued)

- Find the differences between pairs and factor
  - $160 - 0 = 160 = 2^5 * 5$
  - $169 - 34 = 3^3 * 5$
  - $131 - 61 = 70 = 2 * 5 * 7$
  - $114 - 99 = 155 - 140 = 189 - 174 = 15 = 3 * 5$
- Identify common factors
  - They all have 5 as a factor
  - Since 5 is a prime and the key has an integer length, we know n=5
    - If the only factor is composite, it may be the key length or a multiple of the key length
- Split the ciphertext by key character and perform frequency analysis

Example (continued)

- Let's look at how to find the first letter of the key
  - Conveniently, the groupings have the same length as the key
  - Just take the first letter of each group
- KLCKBZPZ FVCIVZREU VEKYZCRZ REEWKKZK KVZKFFKY
- Find the frequencies and relative frequencies
  - Sort letters by most likely
  - Use a frequency table to create a short candidate list
    - e is the most common letter in English, so we expect it near the top
      - It might be mapped to k, z, e, v, c, f, or r
      - This would mean the first character of the key is g, v, a, r, y, b, or n, respectively
    - Try the same with another top letter, such as t
      - Most likley key letters now are r, g, l, c, j, m, y
  - Both e and t show that g, r, and y as likely for the first key letter

k	0.219512
z	0.170732
e	0.097561
v	0.097561
c	0.073171
f	0.073171
r	0.073171
y	0.04878
b	0.02439
i	0.02439
l	0.02439
p	0.02439
u	0.02439
w	0.02439
a	0
d	0
g	0
h	0
j	0
m	0
n	0
o	0
q	0
s	0
t	0
x	0

Example (continued)

- You can try this with a couple more common letters if you'd like, but this is a good short list
- Repeat this for the other four key characters and try to decrypt with different combinations of your top characters for each key position
- What you'll find is that when you decrypt with "romeo" you get the following message

```
twoho useho ldsbo thali keind ignit yinfa irver
onawh erewe layou rscen efrom ancie ntgru dgebr
eakto newmu tinyw herec ivilb loodm akesc ivilh
andsu nclea nfrom forth thefa tallo insof these
twofo esapa irofs tarcr ossdl overs taket heirl

two households both alike in dignity in fair
verona where we lay our scene from ancient grudge
break to new mutiny where civil blood makes civil
hands unclean from forth the fatal loins of these
two foes a pair of starcrossd lovers take their l
```

Exercise

- Open tableau.py in IDLE
- There is a string in the variable *ciphertext*
  - The spaces have been preserved
- Tasks
  - Implement Vigenère decryption in decrypt
  - Perform the attack just described (you may do some of it by hand)
- Goal: find the plaintext and key

- Hints
  - To handle the spaces, try creating a second string that is the message without spaces
    - Encrypt/decrypt using this list
    - When you output your result, use the original list to put the spaces back in

So I'm at Shmoocon and there's this puzzle…

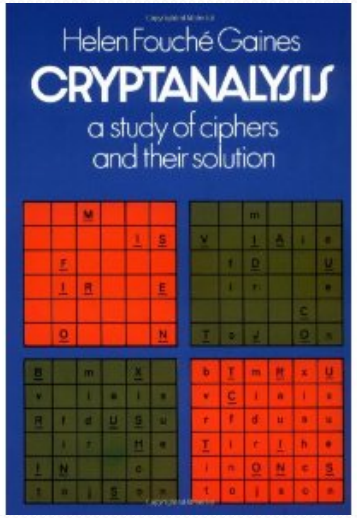
- Monoalphabetic and polyalphabetic ciphers are great for conference challenges
  - But how can you tell which one it is?
- Index of coincidence
- $\mathbf{I} = \sum_{a \text{ in alphabet}} count(a) * [count(a) - 1] / length * (length - 1)$
- In English
  - For each character in the alphabet
    - Multiply the number of times the character appears times that number minus one
    - Divide by the product of ciphertext length and ciphertext length minus 1

Index of coincidence

- Its all about frequencies
- A larger index of coincidence indicates a monoalphabetic cipher
  - Characters are not evenly distributed
  - $I \approx \sum 0 \leq i \leq 25 \uparrow \Pr(i) \uparrow 2 \approx 0.065$
- A smaller index of coincidence indicates a polyalphabetic cipher
  - Characters are evenly distributed
  - $I \approx 26(1/26)\uparrow 2 = 1/26 \approx 0.03846$

And many more!

- Although these are probably the most likely to show up in a text book, they are not the only ones
  - There are a couple more in the course text
- If you are interested in learning more about human-computable ciphers and attacks, check out "Cryptanalysis: a study of ciphers and their solution"



Attack Models and Metrics

Dimensions of attacks

- It can be difficult to compare attacks and definitively say that one is better than another
  - May be comparing apples to oranges
- There are several dimensions to an attack
  - Attack model
  - Data complexity
  - Time complexity
  - Memory (space) complexity

Oracles

- Oracle
  - Something that you query, and it returns a response based on your query
  - Encryption oracle returns the ciphertext of the given plaintext
  - Decryption oracle returns the plaintext of a given ciphertext (or an error)
  - The oracle does not release the key, and the attacker need not know the key to query the oracle
- Examples
  - Smartcards
  - TPM

What is an adversarial model?

- Puts bounds on what the adversary can and cannot do
  - Can she query an encryption or decryption oracle, or only observe?
  - Can she obtain plaintext?

- Known plaintext attack (KPA)
  - Know plaintext and corresponding ciphertext
- Chosen plaintext attack (CPA)
  - Access to encryption oracle
- Chosen ciphertext attack (CCA)
  - Access to decryption oracle
- Ciphertext-only attack (COA)
  - No access to oracle
  - No corresponding plaintext

Ordering

- Intuitively, it may seem that a stronger adversary should be capable of the same attacks that a weaker adversary is
  - Not necessarily true
- Example: CCA
  - Some systems do not return decryption results (plaintext) if an error is detected
  - Attacker can use error messages to find the key, but does not have access to plaintext
    - Cannot mount a known plaintext attack
- "Stronger adversary" means the adversary has more control
  - CPA adversary is stronger than KPA
    - Can query oracle
- Compare by attacker capability

How are attacks compared?

- Adversarial model is only one part
- Three other factors used to compare:
  - Time
    - Number of evaluations of the function required (e.g. # times encrypt(x) is called)
  - Memory
    - How many cipher states and/or candidate keys need to be stored?
  - Data
    - How many ciphertexts or plaintext-ciphertext pairs are needed?

- In cryptanalysis research, may also be a parameter for rounds
  - How much the cipher was reduced to perform analysis
    - Start with significant reduction, progress to full cipher
  - Useful in research, but not as important in practice
    - When attacks get close to full cipher, then becomes important

Which is better?

	Model	Time	Memory	Data
Attack1	CPA	2^{32}	2^{64}	2^{49}
Attack2	CPA	$2^{31.5}$	2^{75}	2^{50}
Attack3	KPA	2^{85}	2^{130}	2^{32}
Attack4	KPA	2^{120}	2^{10}	2^{180}

- Attacks in one model are not always directly comparable to those in another model
  - Adversary has different abilities
- Between attacks in same model, different dimensions are better than others
  - The best attack in a particular situation depends on what you have available to you

Choosing

- You do not look for “the best” attack and try to meet its requirements
- You look at the best you can do given your limits
  - What your capabilities are
    - Choose corresponding model
  - What data you have or can obtain
    - Gives you max data complexity
  - How much memory do you have available
    - Gives you max memory complexity
      - This often not in bytes, but in keys or states (multiple bytes)
        - Be sure to adjust accordingly
  - Then all that is left is time complexity

Number theory & abstract Algebra

A brief introduction

The basics

- Prime and composite
- Divisibility
- Greatest common divisor (GCD)
- Congruence
- Groups, rings, fields

- Unless otherwise specified, we'll be working with integers

Terminology

- Let's begin by answering the following questions to make sure that we're all speaking the same language

- What is a prime?
- What is a composite?
- Is 1 prime, composite, or am I being tricky?

Terminology answers

- A prime is a natural number greater than one that cannot be expressed as a product of any numbers other than one and itself
- A composite is a natural number that is a product of primes, rather than being a prime
- One is neither a prime nor a composite
  - It is a special number called the unit
  - In number theory, and therefore cryptology, it is not

Divisibility

- A/B, B>o
- A = qB + r
  - Integer quotient q
  - Integer remainder r (o ≤ r < B)
  - This is the division theorem
- If r = o, then A = qB
  - B divides A
    - Written B|A

More properties

- Divisibility is transitive
- If a|b and b|c, then a|c
  - $b = q_1 a$, $c = q_2 b$
  - $c = q_2 q_1 a$
  - $c = qa$ (here $q = q_2 q_1$)

- If a|bc, must it also be true that a|b or a|c?

Divisibility of products

- If a|bc, must it also be true that a|b or a|c?
- Let's get a feel for this by examples
- Example 1: a=2, b=2, and c=10
  - It is true that 2|20
  - It is true that 2|2 and that 2|10
  - So far, it looks good
- Example 2: a=4, b=2, and c=10
  - It is true that 4|20
  - It is not true that 4|2 or 4|10
  - Ok, so clearly it isn't always true
- As it turns out, a|bc implies that a|b or a|c only when a is a prime
  - We saw in example 2 that when a is composite, its primes may be split between b and c

GCD

- Greatest common divisor, written gcd(a,b)
  - Largest number g such that g|a and g|b
  - Always positive
  - May be any natural number

- Trick questions
  - gcd(-x, x) = x
  - gcd(-5, -10) = 5
  - gcd(0,0) = ?
    - This one may vary by the rules you are following
      - Undefined or 0

Congruence

- Computers implement finite number systems
  - A byte can only store {0,…,255}
  - A 32-bit word can only store $\{0,\ldots,2^{32}-1\}$
  - BIGNUM is limited to available memory
- What happens to numbers outside those ranges?
  - Mapped to a congruent value
- a and b are congruent mod n iff $n|(a-b)$
  - The distance between a and b on the number line is a multiple of n
  - Write $a \equiv b \pmod{n}$
  - n is called the modulus
- If $a \equiv b \pmod{n}$, then $(a-b) = qn$ by division theorem

Congruence examples

- Let n=3
- Then:
  - {…,-9,-6,-3,**0**,3,6,9,…}
    - 3q+0
  - {…,-8,-5,-2,**1**,4,7,10,…}
    - 3q+1
  - {…,-7,-4,-1,**2**,5,8,11,…}
    - 3q+2
- $-9 \equiv 6 \pmod 3$
  - They are in the same congruency class
  - 6 – (-9) =15 is a multiple of 3
- $8 \equiv 11 \pmod 3$
- Each class is represented by the remainder (or residue): 0, 1, or 2
  - 8 mod 3 = 2

Congruence and operations

- Can use congruent numbers interchangeably in calculations
- Let n=25
  - 94+20 mod 25 = 14
  - 94 ≡ 19 (mod 25), 19+20 mod 25 = 39 mod 25 = 14
  - 94-20 mod 25 = 24
  - 19-20 mod 25 = -1 mod 25 = 24
  - 94*20 mod 25 = 5
  - 19*20 mod 25 = 380 mod 25 = 5
  - 80/20 mod 25 = 4
  - 80 ≡ 5 (mod 25), 5/20 ≠ 4
  - Huh. Why didn't this work?

WARNING: Invalid operation

- People sometimes make the mistake of thinking that if multiplication is available, so is division
  - After all, it is the inverse, right?
- Integers (Z) are not closed under division
  - 4/2 = 2 is in Z
  - 2/4 = 0.5 is not in Z
- In Z_n, the set of integers mod n, results of division are meaningless
  - As we just saw, they are not necessarily correct

- When working in Z or a finite subset of Z, division is not a valid operation
- Let's abstract this a bit

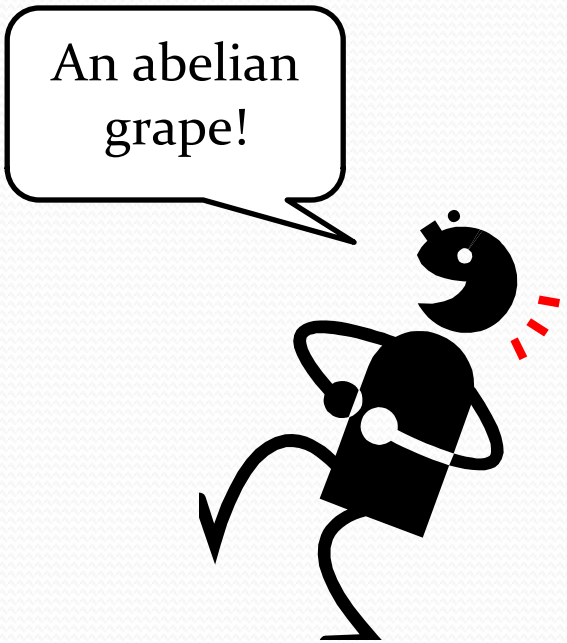
Groups

- Set S with a binary operation ◊
- (S, ◊) is a group iff
  - S is closed under ◊
    - For all x and y in S, x◊y is in S
  - ◊ is associative
    - (x◊y)◊z = x◊(y◊z)
  - S has an identity, e
    - For all x in S, x◊e = e◊x = x
  - All elements have inverses
    - For all x in S, there is a x' x◊x' = x'◊x = e

- If x◊y = y◊x (◊ is commutative) as well, then (S, ◊) is an abelian group

Bad math joke

- What's purple and commutes?



Group examples

- $\Diamond$ = addition mod n, S = Z_n = {0,…,n-1}
  - Addition is associative
  - S is closed under addition mod n
  - 0 is the identity
  - The inverse of x is n-x
    - $x+n-x = n \equiv 0 \pmod{n}$

Multiplicative groups

- $\lozenge$ = multiplication mod n, $S = Z^*_n = \{1,\ldots,n-1\}$
  - Multiplication is associative
  - S is closed under multiplication mod n
  - a is the identity
  - The inverse of x is x^{-1}
    - $x \lozenge x^{-1} \bmod n = 1$
    - Exists if and only if gcd(n,x)=1
- For $S = Z^*_n$, if n is not prime, then there will be elements with no inverse
- If S only contains elements that are relatively prime to n, maybe it is a group
  - Check for closure and the rest

Rings

- A ring consists of
  - A set G and two operations ◊ and ○
  - (G, ◊) is an abelian group
  - (G, ○) isn't quite a group
    - ○ is associative
    - Closure is satisfied
    - The identity property is satisfied
    - All elements do *not* need inverses
  - ○ distributes over ◊
    - (a ◊ b) ○ c = (a ○ c) ◊ (b ○ c)
    - c ○ (a ◊ b) = (c ○ a) ◊ (c ○ b)
- Think of ◊ as addition and ○ as multiplication

Field

- A field satisfies all the properties of a ring, plus more
  - Only the identity under $\lozenge$, $e_{\lozenge}$, does not have an inverse under $\circ$
  - $(G\backslash\{e_{\lozenge}\}, \circ)$ is an abelian group
- Z_3 with operations modular addition and modular multiplication is a field
  - $(Z_3,+)$ is a group
  - $(Z_3\backslash\{0\}, x)$ is a group
    - This is Z_3 without the 0
    - It is the set Z^*_3

Ready?

- Wasn't that fun?
- Now let's put it to good use ☺

FACTORING

Asymmetric systems

Asymmetric construction

- Asymmetric algorithms use different keys for encryption and decryption
  - Algorithms are based on hard problems
- Factoring is one such problem
  - Given a very large integer n with large factors, it is difficult to find the factors
- The fundamental theorem of arithmetic states that all integers can be written as a unique product of primes
  - N = $\prod i\uparrow\blacksquare p\downarrow i\uparrow\downarrow\uparrow k$, where $p<N$, $p\downarrow i$ is a distinct prime, and $k>0$
  - (this is why 1 can't be a prime in number theory)

- Difficulty depends on what the factors are
  - 2 is pretty easy to find
  - So is 10

Naïve method (brute force)

- If N is composite, then it must be the product of at least two primes
  - If p is the smallest factor of N, then $N > p^2$
- Trial division by at most $\sqrt{N}$ integers

- Works great for small N, but what about N with 1024 bits?

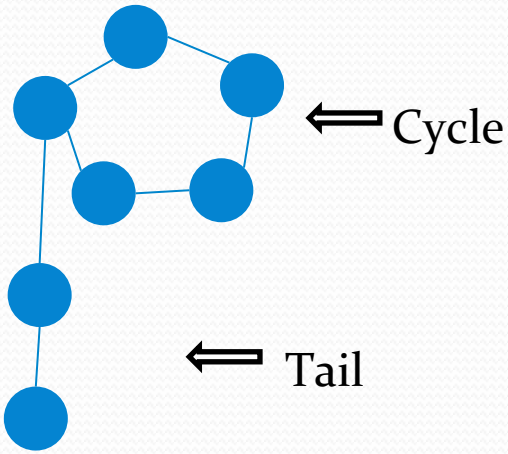
Another approach

- N is composite, let p be the smallest prime factor of N
- If $x \neq x'$ and $x \equiv x' \pmod{p}$, then $p \leq \gcd(x - x', N) < N$
- Can find p by finding collision
  - $x \equiv x' \pmod{N}$

- Why should this work?
  - p|N
  - p|$(x - x')$
  - If $x \equiv x' \pmod{N}$, then $(x - x')|N$
  - $\gcd(x - x', N)$ is a factor of N
    - 1,N, and N's prime factors
- So if $1 < \gcd(x - x', N) < N$, then $\gcd(x - x', N)$ is a prime factor of N

Pollard's Rho

- Algorithm for finding cycles in number patterns
- Two variables moving at different speeds
  - A = f(A)
  - B = f(f(B))
- Graph looks like the symbol rho



Example

- N = 1517
- Let $f(x) = (x^2 + 1) \bmod N$
- Start at A=134
- Sequence A = f(A) is

134	1270	330	1194	841	360
656	1026	1396	989	1174	841
360	656	1026	1396	989	1174

Sequence repeats at 841

Example

- N = 1517
- Let $f(x) = (x^2 + 1) \bmod N$
- Start at A=134

134	1270	330	1194	841	360
656	1026	1396	989	1174	841
360	656	1026	1396	989	1174

Example

- N = 1517
- Let $f(x) = (x^2 + 1) \bmod N$
- Start at A=134

Tail length: 4
Cycle length: 7

134 1270 330 1194 841

360

1174

989

656

1396

1026

Pollard's Rho

Input: composite N,

external function $f(x) = (x^2 + a) \bmod N$, where a is a small integer

Output: prime factor of N, or fail

```
A = rand(0,N-1)
B = A
g=1
while g=1 do
        A = f(A)
        B = f(f(B))
        g = gcd(A – B, N)
if g < N
        return g
else
        fail
```

Example

Suppose $f(x) = (x^2 + 3) \bmod N$

Step	A	B	GCD(A-B, N)
0	2	2	1
1	7	52	1
2	52	742	1
3	1190	200	1
4	742	705	37
5	1413	1458	1
6	200	742	1
7	561	200	1
8	705	705	1517
9	969	1458	1
10	1458	742	1

tail

cycle

Exercise

- Open factoring.py
- You'll see a stub for 'pollardRho(N)'
  - N is the number you are trying to factor
- An f(x) function has already been created for you
  - It takes two parameters, x and composite N
- Objective
  - Implement Pollard's Rho
  - Factor 1234567
- Finished early?
  - Try factoring larger integers
  - Don't stop when you've got an answer. Let it run for a while and observe the cycles

Use gcd function in the cryptoUtils module provided in the same folder.

To import:
import cryptoUtils as cu

To use:
cu.gcd(X, N)

Analysis of Pollard's rho

- $O(n^{1/4})$
  - At most $\sqrt{p}$ iterations
  - $p < \sqrt{N}$

- What if it fails to find a prime?
- Two options:
  - Try a different initial value
  - Try a different f(x)
    - Can define as $(x^2 + b) \bmod N$, where b=rand(1,N-3)

Pollard's p-1

- Pollard has another solution to factoring
- Fermat's little theorem
  - Given prime p and any integer b, $b^{p-1} \equiv 1 \pmod{p}$
- Let p be a prime factor of N
- If $x = q(p-1)$, then $p \mid \gcd(b^x - 1, N)$

- Let integer B be an upper bound
- Going to use $(p-1) \mid B!$
  - May or may not work depending on values of N and B
- Compute a such that $a \equiv 2^{B!} \pmod{n}$ and $a \equiv 2^{B!} \pmod{p}$
  - $a \equiv 1 \pmod{p}$
  - $a-1 = kp$
  - $p \mid (a-1)$
- $p \mid n$ and $p \mid \gcd(a-1, n)$

Pollard's p-1

Input: composite N, upper bound B
Output: prime factor of N, or failure

a=2
for i=2 to B
    a = a^i mod N
d = gcd(a-1,N)
if 1 < d < N
    return d
else
    fail

Note
The algorithm on page 75 of the text is slightly different. It requires a list of primes, whereas this version does not.

Exercise

- Open factoring.py
- You'll see a stub for 'pollardP1(N,B)'
  - N is the number you are trying to factor
  - B is the threshold
- Objective
  - Implement Pollard's p-1 method
  - Factor 15770708441, with bound 180
- Time estimate
  - 15-30 minutes
- Finished early?
  - Try decreasing and increasing the bound. How large does B need to be for this composite N?
  - What about for N=12345678910111213?

Use modExp function in the cryptoUtils module to compute a^i mod N.

To import:
import cryptoUtils as cu

To use:
cu.modExp(a, i, N)

Analysis of p-1

- Complexity: $O(B \log B (\log n)^2 + (\log n)^3)$
  - Great when B is very small compared to $\sqrt{N}$
  - Not so great when B is near $\sqrt{N}$
    - Approaches brute force
- Only works when p-1 has "small" prime factors
  - Easy to prevent attack
  - Suppose N=pq
    - Large primes p and x such that p = 2x+1
    - Large primes q and y such that q = 2y+1
    - Factors too large for p-1 to work

General number field sieve

- The fastest known method factoring method
  - Subexponential complexity
- Very complicated
  - There's a good reason only half a page is devoted to it in the text
  - Plenty of fun with polynomials, ring homomorphisms, and modular roots
- There are tools available that can handle smaller numbers
  - GGNFS, Msieve
  - 512 bits is doable
  - Might help you out in a game of capture the flag
- Record: 768-bit modulus
  - Years of effort on distributed system
  - GNFS experts

Prime selection

- Techniques we discussed can find prime factors
  - It is hard to find large primes
- So the big question is: when the key was chosen, where did the primes come from?
- Options
  - Have a list of large primes to use

Bad idea

    - What could go wrong?
  - Generate a random number of appropriate length and determine whether it is prime

Good idea

    - But if factoring is hard, how do we know it's a prime?

Miller-Rabin

- The standard method of finding large primes is to
  1. Select a random number of appropriate size
  2. Trial division of primes up to a threshold
     - If division succeeds, go back to 1
  3. Use Miller-Rabin primality test to decide if it is prime
     - If composite, go back to 1
- Miller-Rabin test finds probable primes
  - Monte Carlo algorithm for identifying composites
  - If it is not definitely a composite, then it is probably a prime

Miller-Rabin primality test

Input: odd integer $n \geq 3$, number of trials $t \geq 1$
Output: "prime" or "composite"

write $n-1 = 2^s r$, where r is odd
choose random a such that $2 \leq a \leq n-2$
for i from 0 to t
    $y = a^r \bmod n$
    if $y \neq 1$ and $y \neq n-1$
        j = 1
        while $j \leq s-1$ and $y \neq n-1$
            $y = y^2 \bmod n$
            if y = 1 return "composite"
            j = j+1
        if $y \neq n-1$ return "composite"
return "prime"

Miller-Rabin test and t

- The probability of an odd composite being labeled as a prime is less than $\mathbf{0.25}^{t}$
- If you need a n-bit prime with k bits of security
  - Choose t such that $p_{n,t} \leq (\mathbf{1/2})^{k}$

Exploiting prime selection

- If a key distributor uses a prime list, then there's an easy way to factor
  - If the list is short, just do trial divisions
  - Collect many moduli, $\{n_1, n_2, \ldots, n_m\}$
    - If n_i and n_j share a prime factor, p, then $\gcd(n_i, n_j) = p$
    - For each modulus, try computing a pairwise GCD until a prime is found
- Even without a use of prime list, this attack will work with some probability on a large collection
  - The GCD computation may be costly though
- Miller-Rabin relies on random number generation
- If the RNG is bad (i.e. predictable), then the adversary can use that to narrow the possible inputs to the prime generation algorithm

Recent findings regarding primes

- There have been two recent studies that looked at RSA prime selection in the wild
  - They both found problems
  - Here's a few
- Using /dev/urandom instead of /dev/random
  - /dev/random should be used for long-lived key generation
    - Blocks when entropy sources not available
      - Some interpreted this as a "usability issue"
- Boot-time entropy hole causes prime to be generated from predictable state
  - Little entropy => the same primes for everyone!
- IBM remote management cards that use a list of 9 primes
  - Wow! 9 choose 2 = 36 different moduli!
  - That's right! You too can break RSA in at most 8 division operations

See:
"Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices", Heninger et al.
"Ron was wrong, Whit is right", Lenstra et al.

Factoring: summary

- Factoring is a hard problem that is used as a foundation for asymmetric cryptosystems
  - The secret depends on prime factors
  - If the modulus can be factored, the system can be broken
- There are several algorithms for factoring, but they are too computational complex to be used on large enough numbers
- Prime selection is at least as important as the key length
  - The key length is supposed to make the primes hard to find!

- Now let's move on to specific attacks on a factoring-based cryptosystem

RSA

Attacking the algorithm

RSA (Rivest, Shamir, Adleman)

- Famous factoring-based cryptosystem
- Large primes p and q
- $N = pq$
- $\Phi(N) = (p-1)(q-1)$
  - This function gives the number of elements $0 < x < N$ such that $\gcd(N,x)=1$
    - Number of elements ***relatively prime*** to N
- The public (encryption) exponent e that is relatively prime to $\Phi(N)$
  - 65537 is a popular choice
- The private (decryption) exponent is computed as follows
  - $de \equiv 1 \pmod{\Phi(N)}$
    - This means d is the multiplicative inverse of e in $Z^*_{\Phi(N)}$
      - $Z^*_{\Phi(N)}$ is the set $\{0,1,\ldots, \Phi(N)-1\}$

RSA operations

- Encryption
  - M in Z_N
    - $Z_N = \{0,\ldots,N-1\}$
  - $C = M^e \bmod N$
- Decryption
  - $M = C^d \bmod N$
- Why does this work?
  - Euler's theorem states that $x^{\Phi(N)} \equiv 1 \pmod N$ if N and $\Phi(N)$ are relatively prime
  - A corollary to Euler's theorem allows us to show:
    - $C^d \bmod N \equiv M^{ed} \bmod N \equiv M^{k\Phi(N)+1} \bmod N \equiv M \bmod N$
      - k is integer ≥ 0
    - In other words, $M^{k\Phi(N)+1} \bmod N \equiv (M^{\Phi(N)})^k M \bmod N \equiv 1^k M \bmod N \equiv M \bmod N$

RSA example

- Key generation
  - Let p=11, q = 13, e=17
  - N = pq = 143
  - $\Phi(N) = (p-1)(q-1) = 10*12 = 120$
  - d=113
    - $17*113 = 1921 = 16*120 + 1 \equiv 1 \pmod{120}$
- Encryption
  - Let m=5
  - $5^{17} = 762939453125 \equiv 135 \pmod{143}$
- Decryption
  - $135^{113} \equiv 5 \pmod{143}$

RSA exercise

- Open the file RSA.py
- You'll see that all of the essentials are provided
  - Class privKey contains private key information
    - Two functions, set and display
  - Class pubKey contains public key information
    - Two functions, set and display
  - Function genD
    - Creates a private exponent
    - Uses inverse function from cryptoUtils.py
  - Encrypt and decrypt functions
    - Use modExp function from cryptoUtils.py
- To create a public key structure
  - Pub = pubKey()
  - Pub.set(p, q, e) #set the key information
  - Pub.display() #print the key information

RSA exercise

- Objective
  - Create keys
  - Encrypt a message
  - Decrypt a message
- Short list of primes
  - 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113
- Try it with different parameters
  - Is a sufficient d always found?
    - What if e=2?
  - When does private exponent generation fail?

Attacking RSA

- The private exponent can be found if N is factored
  - p, q, and e are all that is needed to find d
- There are other attacks as well that exploit parameter properties
  - Public exponent e
  - Leaked exponents
  - Known Φ(N)

Using Φ(N)

- Computing Φ(N) is as hard as factoring
- Some side channel may allow its discovery
  - i.e. stored in memory
- Can be recognized by similarity to N in most significant half

  $\Phi(N) = (p-1)(q-1)$

  $= pq - p - q + 1$

  $= N - p - q + 1$

  - Since p and q are about half the length of N, the top half of bits of Φ(N) will be very similar to those in N, especially as the length increases
  - Example:
    - Let p and q be 16 bits each, N is 32 bits
    - Max value (not necessarily prime) of p and q is $2^{16} - 1$
    - Min value (not necessarily prime) of p and q is 2^{15}
    - Value of p+q is at most $2(2^{16} - 1) = 2^{17} - 2$ and at least 2^{16}
    - $N - 2^{17} - 1 < \Phi(N) < N - 2^{16} + 1$

Using Φ(N) (continued)

- Φ(N) can be used to solve for p and q
- Write q = N/p
  - Know this is an integer because q|N
- Then substitute

$$N = pN/p = \Phi(N) + p + N/p - 1$$
$$pN = p(\Phi(N) + p - 1) + N$$
$$0 = p(\Phi(N) + p - 1) + N - pN$$
$$= p(\Phi(N) + p - 1 - N) + N$$
$$= p^2 - p(N - \Phi(N) + 1) + N$$

  - Only one unknown
- Solve equation for p

This was a little tricky. We moved into the reals for division, but all of our values are integers.

Leaked exponent

- Suppose Bob accidentally leaks his private exponent, d
- Bob panics and changes his public exponent and calculates a new private exponent
- His new private exponent is safe from prying eyes

- Is Bob's key secure?
  - Probably not
- He changed the secret exponent, so why isn't it secure?

Leaked exponent

- There is a Las Vegas algorithm that can factor N given d
  - He can change d, but if he doesn't change N we can still calculate p and q
    - We can derive his new private exponent
  - Unlike games in the real Las Vegas, the odds are very good
    - More than ½

Factor with private exponent

Input: modulus N, private exponent d, public exponent e
Output: prime factor of N

Let $ed-1 = 2^s r$, r is odd
w = rand(1,N-1)
x = gcd(w,N)
If $1 < x < N$
    return x
$v = w^r \bmod N$
If $v \equiv 1 \pmod N$
    fail
While $v \not\equiv 1 \pmod N$
    $v_o = v$
    $v = v^2 \bmod N$
if $v_o \equiv -1 \pmod N$
    fail
Else
    $x = \gcd(v_o + 1, N)$
    return x

Stinson, "Cryptography: Theory and Practice", 3rd Ed., page 204

Factor with private exponent (continued)

- If it fails, try again
  - Different values of w yield different results
  - The chance the algorithm succeeds with a selected w is a little over 50%
- The lesson?
  - If your private exponent is compromised, change your entire key
    - This means the primes too!

Exercise

- Open factoring.py
- You'll see a stub for 'VegasFactor(N, d, e)'
  - N is the number you are trying to factor
  - d is the private exponent
  - e is the public exponent
- Objective
  - Implement factoring with private exponent
  - Factor the moduli of the following (N, d, e) tuples
    - (437, 317, 5)
    - (38419, 26269, 13)
    - (11021, 3139, 31)

Use modExp function in the cryptoUtils module to compute a**i mod N.

-1 ≡ N-1 (mod N)
2**x= 1<<x = 2<<(x-1)

Low exponent attack

- Suppose Alice sends a message, M, to Bob, Charley, and Dana
  - All of their keys have different moduli
    - $\gcd(N_B, N_C) = \gcd(N_B, N_D) = \gcd(N_D, N_C) = 1$
  - All of their keys have the same small public exponent, e=3
- Eve knows the following congruence relations
  - $C_B \equiv M^3 \pmod{N_B}$
  - $C_C \equiv M^3 \pmod{N_C}$
  - $C_D \equiv M^3 \pmod{N_D}$
- She can efficiently find M using the Chinese Remainder Theorem

Chinese Remainder Theorem (CRT)

- Another quick math lesson before we proceed
  - This will play a role in how we think about operations
- Z_N is isomorphic to product group Z_p x Z_q
  - Structurally the same
- Z^*_N is isomorphic to Z^*_p x Z^*_q

- Computing x^b mod N is expensive
- Computing x^b mod p and x^b mod q, where N=pq is cheaper
  - Often used to speed up exponentiation

CRT example

- $15 = 3 * 5$
- $Z^*_{15} = \{1,2,4,7,8,11,13,14\}$
- $1 \leftrightarrow (1,1)$
  - $1 = 1 \pmod 3$
  - $1 = 1 \pmod 5$
- $2 \leftrightarrow (2,2)$
  - $2 = 2 \pmod 3$
  - $2 = 2 \pmod 5$
- $4 \leftrightarrow (1,4)$
  - $4 = 1 \pmod 3$
  - $4 = 4 \pmod 5$
- $7 \leftrightarrow (1,2)$
  - $7 = 1 \pmod 3$
  - $7 = 2 \pmod 5$
- $8 \leftrightarrow (2,3)$
  - $8 = 2 \pmod 3$
  - $8 = 3 \pmod 5$
- $11 \leftrightarrow (2,1)$
  - $11 = 2 \pmod 3$
  - $11 = 1 \pmod 5$
- $13 \leftrightarrow (1,3)$
  - $13 = 1 \pmod 3$
  - $13 = 3 \pmod 5$
- $14 \leftrightarrow (2,4)$
  - $14 = 2 \pmod 3$
  - $14 = 4 \pmod 5$

CRT example (continued)

- Suppose we wanted to compute 14*13 mod 15
  - 14 → (2,4)
  - 13 → (1,3)
- 14 * 13
  - 2*1 = 2 (mod 3)
  - 4*3 = 2 (mod 5)
- (2,2) → 2
  - Indeed, 14*13 = 182 ≡ 2 (mod 15)

Conversion back

- Use extended Euclidean algorithm to find s and t
  - $s*3 + t*5 = 1 \pmod{15}$
  - s=2, t=2
  - $x = 3s = 6$
  - $y = 5t = 10$
- $(1,4) \rightarrow 1*y + 4*x = 1*10 + 4*6 = 34 = 4 \pmod{15}$
- $(2,2) \rightarrow 2*10 + 2*6 = 32 = 2 \pmod{15}$
- And so on

Low exponent attack (continued)

$$M^3 \equiv C_B \pmod{N_B}$$

$$M^3 \equiv C_C \pmod{N_C}$$

$$M^3 \equiv C_D \pmod{N_D}$$

$$M^3 \equiv C_B*(N_C*N_D)*((N_C*N_D)^{-1} \bmod N_B) +$$
$$C_C*(N_B*N_D)*((N_B*N_D)^{-1} \bmod N_C) +$$
$$C_D*(N_B*N_C)*((N_B*N_C)^{-1} \bmod N_D) \bmod (N_B * N_C * N_D)$$

- Let $M' = M^3$
- Compute the cubic root of M'
  - Normal cubic root over the integers
  - Not the *modular* cubic root, which is difficult to compute

Low exponent attack (continued)

- The number of ciphertexts needed is equivalent to the value of the public exponent
- This does not mean that e=3 is necessarily insecure
  - The same message is essential to this attack
  - If only one congruence is known, this attack will not work
- If e = 65537, then Alice would have to send the same message to 65537 parties with different moduli and same exponent for attack to succeed
  - When is the last time you sent the same message to 65537 of your closest friends using RSA?
  - Just another reason that this is a popular exponent choice
    - Small enough to facilitate fast encryption
    - Large enough to make small exponent attacks difficult

How is this useful?

- Bad protocols
  - Good protocols use nonces to prevent messages from being predictably identical
- Same message sent repeatedly
  - Same every time
- You need to capture e ciphertext-modulus pairs
  - If e=3, only need 3 pairs
  - If e=65537, need 65537 pairs
    - Also need to take the 65537^{th} root instead of cubic root
- Take away
  - When you design protocols, add random nonces
  - When you select a public exponent, choose one greater than 3
    - Still want to keep them small for efficiency

Exercise

- Suppose Alice sends M to three parties with the following public keys
  - $N_1 = 11413$, $e_1 = 3$
  - $N_2 = 18511$, $e_2 = 3$
  - $N_3 = 3799$, $e_3 = 3$
- You capture the following ciphertexts
  - $C_1 = 6249$
  - $C_2 = 6032$
  - $C_3 = 2260$
- Objective: find the plaintext, M

Exercise

- Open RSA.py
- Complete function lowExponent(C1,N1,C2,N2,C3,N3,e)
- Verify that your solution for M is correct using the public keys
  - Make sure the encryption creates the correct ciphertext

- Hints
  - x**a is python for x^a
  - 1/3 = 0, but 1.0/3 = 0.3333…
  - int(x) casts a float to an int, but it might give an incorrect answer
    - Try int(math.ceil(x))
    - Yay, rounding!

A note on RSA

- We've only been talking about "plain" RSA
  - Deterministic
  - Same input and key always results in same output
- RSA is more often used with padding
  - Parts of the message are random
    - This is a good thing!
  - Makes it more difficult to get two of the same or known plaintext-ciphertext pair

DISCRETE LOG

Asymmetric systems

Generators

- We need a little bit more algebra for this section
- Suppose ◊ = modular multiplication
- The set generated by a (denoted <a>) is all elements that can be written as $a^0, a^1, a^2, \ldots$
  - In Z^*_N, this would be all elements that can be written as $a^i \bmod N$
- The order of a set is the number of elements in the set
- The order of <a> is the smallest positive x such that $a^x = 1$
  - If ◊ = addition, then x such that ax = 0
- Example
  - Let a=2, N=7
    - $2^1 \bmod 7 = 2$
    - $2^2 \bmod 7 = 4$
    - $2^3 \bmod 7 = 8 \bmod 7 = 1$
    - The set generated by 2 (in Z^*_7) is <2> = {1,2,4}
    - <2> has order 3 in Z^*_7

Generators (continued)

- An element a generates a set S if all elements in S can be expressed as a power of a
- In previous example, 2 only generates a subset of Z^*_7
- What about <3>?
  - $3^1 \bmod 7 = 3$
  - $3^2 \bmod 7 = 2$
  - $3^3 \bmod 7 = 6$
  - $3^4 \bmod 7 = 4$
  - $3^5 \bmod 7 = 5$
  - $3^6 \bmod 7 = 1$
  - The set generated by 3 (in Z^*_7) is <3> = {1,2,3,4,5,6} = Z^*_7
- 3 generates Z^*_7

The discrete log problem

- Given a, b, and p, what is the value of x if $\alpha^x \bmod p = \beta$?
- This is hard (in terms of complexity) on classical computers
  - i.e. ***your*** computer
- There may be multiple solutions for x
- Why?

Why are multiple solutions possible?

- Let n be the order of α mod p
  - Then $\alpha^n \equiv 1 \pmod p$
- Can expand this
  - $\alpha^{2n} \equiv (\alpha^n)^2 \equiv 1^2 \equiv 1 \pmod p$
  - More generally, $\alpha^{kn} \equiv (\alpha^n)^k \equiv 1^k \equiv 1 \pmod p$
- Can write x as a function of n
  - i.e. x = kn+j

- We saw that the order of <2> mod 7 is only 3 (as opposed to 6)
- Suppose $2^x \bmod 7 = 4$
  - $x = 0*3 + 2$
    - $2^2 \bmod 7 = 4 \bmod 7 = 4$
  - $x = 1*3 + 2 = 5$
    - $2^5 \bmod 7 = 32 \bmod 7 = 4$
  - And that is all that are in Z^*_7, but there are an infinite number of solutions in Z
- There is only one solution in Z^*_p if α generates Z^*_p

How do you find a generator of a set?

- Two options
  - Randomized algorithm
  - Choose a standard modulus and generator
- The algorithm
  - Choose random element α in Z^*_p

  For all i, where p_i is a factor of (p-1)

  Compute $b = \alpha^{((p-1)/p_i)}$,

  If b = 1, go back to step 1

  Return α
- Diffie-Hellman key exchange is a discrete log crypto algorithm
  - Used in SSH and IKE
  - Standardized groups ☺
    - There's several
  - Parties agree on modulus size and group before exchange
- Let's take a look at some, shall we?

Diffie-Hellman group 1

- RFC 2409
- Different modulus sizes specified
  - 768-bit, 1024-bit
  - Elliptic curve groups as well, but we haven't done those yet
- Not the most secure option these days
- Example: 768-bit modulus
  - Generator is 2
  - 768-bit prime is (in hex)

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234
C4C6628B 80DC1CD1 29024E08 8A67CC74
020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437
4FE1356D 6D51C245 E485B576 625E7EC6
F44C42E9 A63A3620 FFFFFFFF FFFFFFFF
```

Diffie-Hellman group 14

- RFC 3526
- Different modulus sizes specified
  - 1536-bit, 2048-bit, 3072-bit, 4096-bit, 6144-bit, and 8192-bit
- Example: 1536-bit modulus
  - Generator is 2
  - 1536-bit prime is (in hex)

```
FFFFFFFF FFFFFFFF C90FDAA2 2168C234
C4C6628B 80DC1CD1 29024E08 8A67CC74
020BBEA6 3B139B22 514A0879 8E3404DD
EF9519B3 CD3A431B 302B0A6D F25F1437
4FE1356D 6D51C245 E485B576 625E7EC6
F44C42E9 A637ED6B 0BFF5CB6 F406B7ED
EE386BFB 5A899FA5 AE9F2411 7C4B1FE6
49286651 ECE45B3D C2007CB8 A163BF05
98DA4836 1C55D39A 69163FA8 FD24CF5F
83655D23 DCA3AD96 1C62F356 208552BB
9ED52907 7096966D 670C354E 4ABC9804
F1746C08 CA237327 FFFFFFFF FFFFFFFF
```

Standardized generators

- These examples both have 2 has the generator
- This is not a coincidence
- 2 has good properties
  - Exponentiation is expensive
    - Multiplication is expensive
  - Left shifts are cheap
  - 2^x can be written as 1<<x
  - Efficient!
- But as we saw with p=7, 2 won't generate every group

- $2^0 = 1 = 1 << 0$
- $2^1 = 2 = 1 << 1$
- $2^2 = 4 = 1 << 2$
- $2^3 = 8 = 1 << 3$
- And so on

Baby-step giant-step method

- Use lookup tables to save time
  - Tradeoff with space
- Precompute first L powers of α

Baby-step giant-step algorithm

Input: α, β,p where $\alpha^x \equiv \beta \pmod p$

Output:

$L = \text{floor}(\sqrt{p})$

Compute lut[1]= $\alpha \bmod p$, lut[2]= $\alpha^2 \bmod p$,..., lut[L]= $\alpha^L \bmod p$

$h=(\alpha^{-1})^L \bmod p$

t = β #starting point

for j=0 to L

    if there is a value i such that lut[i]=t

        return i+j*L

    t = t * h mod p

BSGS Example

- Given $3^x \bmod 113 = 37$, find x
- $L=\text{floor}(\sqrt{113})=10$
- Precompute table
  - $\text{Lut}[0] = 3^0 \bmod 113$
  - $\text{Lut}[1] = 3^1 \bmod 113$
  - …
  - $\text{Lut}[10] = 3^{10} \bmod 113$
- $h=(3^{-1})^{10} \bmod 113 = 61$

- Iterate through j loop
  - j: 0, t: 37 ← $t = \beta$
  - j: 1, t: 110
  - j: 2, t: 43
  - j: 3, t: 24
  - j: 4, t: 108
  - j: 5, t: 34
  - j: 6, t: 40

If value not found, set next t: $t = t * h \bmod p$

- Result: 10*6 + 7 = 67
  - $3^7 \bmod 113 = t = 40$
  - j = 6
  - 40 is at index 7, which is why it is added to 10*6

BSGS analysis

- Memory: $\sqrt{p}$
  - For the powers of α
- Time: $\sqrt{p}$ log(p)

Exercise

- Open discreteLog.py
- Complete babyStepGiantStep(a,b,p)
  - a is generator, b is result, p is prime modulus
- Objective: try to solve and verify the following
  - $89^x \bmod 809 = 618$
  - $16^x \bmod 809 = 46$
  - $16^x \bmod 809 = 324$
  - $2^x \bmod 2777 = 512$

Adaptations

- Factoring ↔ discrete log
- Algorithms can be adapted

- Pollard's rho
  - Partition {0, …, p-1} into three sets of approximately equal size
    - Many partitions possible
      - {0,…, (p-1)/3}, {(p-1)/3+1,…, 2(p-1)/3}, {2(p-1)/3,…,p-1}
      - $x \equiv 0 \pmod 3$, $x \equiv 1 \pmod 3$, $x \equiv 2 \pmod 3$
  - Compare triples (x_i, a_i, b_i) and (x_{2i}, a_{2i}, b_{2i})
  - Look for collision $x_{2i} = x_i$

Pollard's rho for discrete logs (Swenson)

Input: α,β,p
Output: solution to $\alpha^x \bmod p = \beta$
$a_0 = b_0 = 0$
$x_0 = 1$
$i = 0$
While $x_i \neq x_{2i}$ do
    $i = i+1$
    $x_i = f(x_{i-1})$
    $a_i = g(x_{i-1}, a_{i-1})$
    $b_i = h(x_{i-1}, b_{i-1})$
    $x_{2i} = f(f(x_{2i-2}))$
    $a_{2i} = g(f(x_{2i-2}), g(x_{2i-2}, a_{2i-2}))$
    $b_{2i} = h(f(x_{2i-2}), h(x_{2i-2}, b_{2i-2}))$
If $b_i = b_{2i}$
    fail
$m = a_i - a_{2i} \bmod (p-1)$
$n = b_{2i} - b_i \bmod (p-1)$
Solve $mx \equiv n \pmod{p-1}$ for x

- If x in partition 1
  - $f(x) = \beta x \bmod p$
  - $g(x,n) = (n+1) \bmod (p-1)$
  - $h(x,n) = n \bmod (p-1)$
- If x in partition 2
  - $f(x) = x^2 \bmod p$
  - $g(x,n) = 2n \bmod (p-1)$
  - $h(x,n) = 2n \bmod (p-1)$
- If x in partition 3
  - $f(x) = \alpha x \bmod p$
  - $g(x,n) = n \bmod (p-1)$
  - $h(x,n) = (n+1) \bmod (p-1)$

Solving for x

- $mx \bmod (p-1) = n$
  - $x = n/m \bmod (p-1)$
  - Oh division, how you trip us up
  - The inverse of multiplication is multiplication
- $x = m^{-1}n \bmod (p-1)$
  - Use inverse(m,p-1) in cryptoUtils

Caveats and fine print

- This version is not the most general
  - Assumes that α has order p-1
    - May not be the case
    - While $\alpha^{p-1} \equiv 1 \bmod p$, there may be an $x < p-1$ such that $\alpha^x \equiv 1 \bmod p$
  - We can say this for the version of BSGS we looked at as well, but it doesn't matter as much there
- Produces multiple solutions

- Let's look at another version from Stinson's book
  - Less memory!
  - Generalized!
  - Assumes you know the order of α

Pollard's rho for discrete logs (Stinson)

Input: α,β,p,n
Output: solution to $\alpha^x \bmod p = \beta$, where α has order n

$(x,a,b) = f(1,0,0)$
$(x',a',b') = f(x,a,b)$
While $x \neq x'$ do
    $(x,a,b) = f(x,a,b)$
    $(x',a',b') = f(x',a',b')$
    $(x',a',b') = f(x',a',b')$

If $\gcd(b'-b, n) \neq 1$
    fail
Else
    return $((a - a')(b' - b)^{-1} \bmod n)$

f:

x in partition	x	a	b
1	$\beta x \bmod p$	$a \bmod n$	$(b+1) \bmod n$
2	$x^2 \bmod p$	$2a \bmod n$	$2b \bmod n$
3	$\alpha x \bmod p$	$(a+1) \bmod n$	$b \bmod n$

Exercise

- Open discreteLog.py
- Complete findOrder(a, n), pollardRho(alpha,beta,p,n), and f(x,a,b,alpha,beta,p,n)
  - Follow Stinson's version
- Objective: try to solve and verify the following
  - $89^x \bmod 809 = 618$
  - $16^x \bmod 809 = 46$
  - $16^x \bmod 809 = 324$
  - $2^x \bmod 2777 = 512$

This exercise is from Stinson, page 239

Exercise hints

- Use remainder mod 3 for partitions
  - 1: $x \equiv 1 \pmod 3$
  - 2: $x \equiv 0 \pmod 3$
  - 3: $x \equiv 2 \pmod 3$
- You can set three variables at once
  - [x,a,b] = f(x,a,b,alpha,beta,p,n)
    - Just need to have f return a list
      - return [result1, result2, result3]
  - If you don't like this, you can break it into three functions like in Swenson's algorithm

Index calculus

- Does not involve derivatives or integrals
  - We're saving that for later ☺
- Most powerful discrete log attack
  - Analog to GNFS
- Not really described in the text
  - Not in Stinson either
  - If you want to read more, see the HAC

- Here we go!

Index calculus algorithm

1. Choose a factor base (set of primes)
2. Obtain set of congruence relations mod p
   - Represent with factor base
3. Create system of equations (mod order of α)
4. Solve the system
5. Profit

Example

- From HAC, page 110
- Problem: $6^x \equiv 13 \pmod{229}$
  - 6 has order 228
- Factor base = {2,3,5,7,11}
- To obtain congruence relations, raise 6 to power mod 229
  - If result can be represented as a product of factor base, keep the relation
  - Otherwise, discard it

Example (continued)

- Relations:
  - $6^{100} \bmod 229 = 180 = 2^2 * 3^2 * 5$
  - $6^{18} \bmod 229 = 176 = 2^4 * 11$
  - $6^{12} \bmod 229 = 165 = 3 * 5 * 11$
  - $6^{62} \bmod 229 = 154 = 2 * 7 * 11$
  - $6^{143} \bmod 229 = 198 = 2 * 3^2 * 11$
  - $6^{206} \bmod 229 = 210 = 2 * 3 * 5 * 7$
- Relations give the following log equations
  - $100 \equiv 2 \log_6 2 + 2 \log_6 3 + \log_6 5 \pmod{228}$
  - $18 \equiv 4 \log_6 2 + \log_6 11 \pmod{228}$
  - $12 \equiv \log_6 3 + \log_6 5 + \log_6 11 \pmod{228}$
  - $62 \equiv \log_6 2 + \log_6 7 + \log_6 11 \pmod{228}$
  - $143 \equiv \log_6 2 + 2 \log_6 3 + \log_6 11 \pmod{228}$
  - $206 \equiv \log_6 2 + \log_6 3 + \log_6 5 + \log_6 7 \pmod{228}$

Note that we're using p-1 here

Example (continued)

- Equations (from previous slide)
  - $100 \equiv 2\log_6 2 + 2\log_6 3 + \log_6 5 \pmod{228}$
  - $18 \equiv 4\log_6 2 + \log_6 11 \pmod{228}$
  - $12 \equiv \log_6 3 + \log_6 5 + \log_6 11 \pmod{228}$
  - $62 \equiv \log_6 2 + \log_6 7 + \log_6 11 \pmod{228}$
  - $143 \equiv \log_6 2 + 2\log_6 3 + \log_6 11 \pmod{228}$
  - $206 \equiv \log_6 2 + \log_6 3 + \log_6 5 + \log_6 7 \pmod{228}$
- Rewrite in a more familiar style

$$
\begin{aligned}
2a + 2b + c &= 100 \\
4a + e &= 18 \\
b + c + e &= 12 \\
a + d + e &= 62 \\
a + 2b + e &= 143 \\
a + b + c + d &= 206
\end{aligned}
$$

Key:
$a = \log_6 2$
$b = \log_6 3$
$c = \log_6 5$
$d = \log_6 7$
$e = \log_6 11$

6 equations and 5 unknowns? If there is a unique solution, we can solve that!

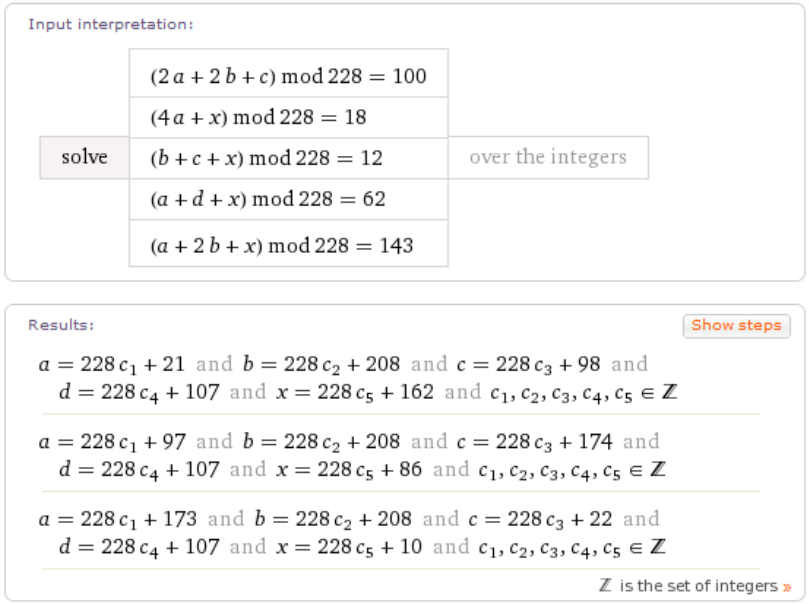
Solving the system of equations

- Many options
  - Take a trip down memory lane to high school algebra
  - Use Gaussian elimination, if you know it
  - Use tools such as Matlab, Mathematica, or Wolfram Alpha
- Least license-free effort: use Wolfram Alpha
  - Let's take a look at how to do this

Solving the system with Wolfram Alpha

- Go to www.wolframalpha.com
- Enter the following in the equation line
  - integer solutions ((2a+2b+c) mod 228) = 100, ((4a+x) mod 228) = 18, ((b+c+x) mod 228) =12, ((a+d+x) mod 228) = 62, ((a+2b+x) mod 228) = 143
- Workarounds
  - Notice that we took out ((a+b+c+d) mod 228) = 206
    - Not sure why, but it did not work with all six equations
  - "e" is interpretted as the transcendental number e, so we replace it with some other character, like "x"
- Because we took one equation out, we couldn't find a unique solution
  - Calculate again, replacing one of the equations with the one we took out

Solving the system with Wolfram Alpha (continued)



Input interpretation:

solve

$(2a + 2b + c) \bmod 228 = 100$
$(4a + x) \bmod 228 = 18$
$(b + c + x) \bmod 228 = 12$
$(a + d + x) \bmod 228 = 62$
$(a + 2b + x) \bmod 228 = 143$

over the integers

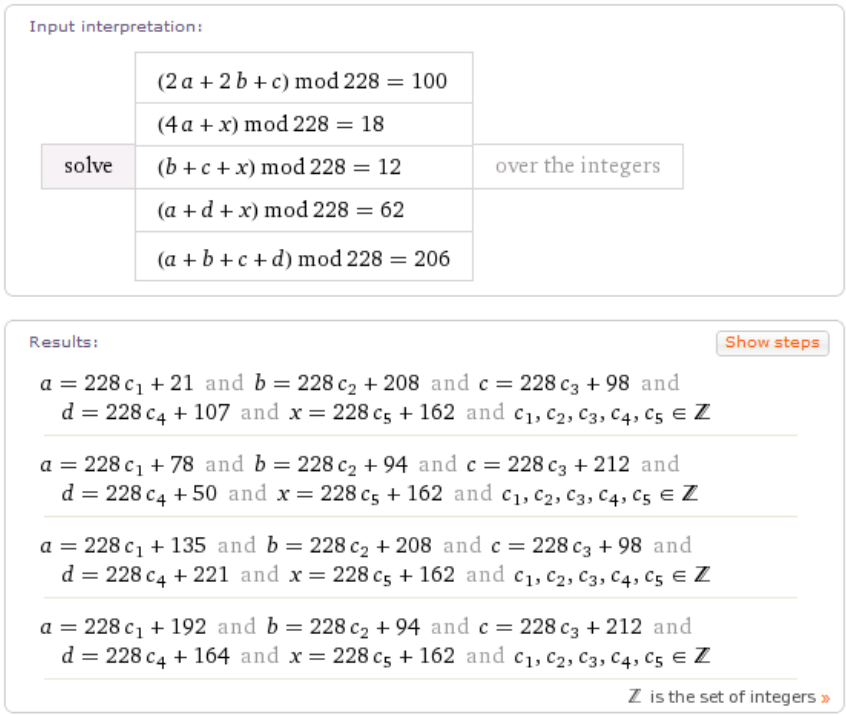
Results: Show steps

$a = 228c_1 + 21$ and $b = 228c_2 + 208$ and $c = 228c_3 + 98$ and $d = 228c_4 + 107$ and $x = 228c_5 + 162$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$a = 228c_1 + 97$ and $b = 228c_2 + 208$ and $c = 228c_3 + 174$ and $d = 228c_4 + 107$ and $x = 228c_5 + 86$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$a = 228c_1 + 173$ and $b = 228c_2 + 208$ and $c = 228c_3 + 22$ and $d = 228c_4 + 107$ and $x = 228c_5 + 10$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$\mathbb{Z}$ is the set of integers »



Input interpretation:

solve

$(2a + 2b + c) \bmod 228 = 100$
$(4a + x) \bmod 228 = 18$
$(b + c + x) \bmod 228 = 12$
$(a + d + x) \bmod 228 = 62$
$(a + b + c + d) \bmod 228 = 206$

over the integers

Results: Show steps

$a = 228c_1 + 21$ and $b = 228c_2 + 208$ and $c = 228c_3 + 98$ and $d = 228c_4 + 107$ and $x = 228c_5 + 162$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$a = 228c_1 + 78$ and $b = 228c_2 + 94$ and $c = 228c_3 + 212$ and $d = 228c_4 + 50$ and $x = 228c_5 + 162$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$a = 228c_1 + 135$ and $b = 228c_2 + 208$ and $c = 228c_3 + 98$ and $d = 228c_4 + 221$ and $x = 228c_5 + 162$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$a = 228c_1 + 192$ and $b = 228c_2 + 94$ and $c = 228c_3 + 212$ and $d = 228c_4 + 164$ and $x = 228c_5 + 162$ and $c_1, c_2, c_3, c_4, c_5 \in \mathbb{Z}$

$\mathbb{Z}$ is the set of integers »

- Both solution sets contain one match
  - a=21, b=208, c=98, d=107, x=162
  - This is the solution
- $\log_6 2 = 21$, $\log_6 3 = 208$, $\log_6 5 = 98$, $\log_6 7 = 107$, and $\log_6 11 = 162$

We have a solution. Now what?

- Recall we started with $6^x \equiv 13 \pmod{229}$
- Pick a random k between 0 and N-1, inclusive
- Calculate $\beta * \alpha^k = 13 * 6^k$ and represent with logs we just found
- Example:
  - Suppose k=77
  - $13 * 6^{77} \bmod 229 = 147$
  - $147 = 3 * 7^2$
  - $\log_6 13 = (\log_6 3 + 2 \log_6 7 - 77) \bmod 228 = 117$
  - So x = 117
    - $6^{117} \equiv 13 \pmod{229}$

Exercise

- Suppose p = 10007, and $\alpha = 5$
- Let {2,3,5,7} be the factor base

- Objective: find $\log_5 9451 \pmod{10007}$

- Hints
  - $\log_5 5$ is an easy congruence
  - Factoring will be trial and error, but the factor base is so small that it will be easy

Adapted from Stinson example 6.5

Discrete log: summary

- Solve for x if $\alpha^x \bmod p = \beta$
- Factoring techniques may be adapted
- Pollard's rho
- Baby-Step Giant-Step
- Index calculus

Implementation notes for factoring and discrete log

- We've done small examples with python here
- If you ever want to do this with realistic size moduli, you'll need something that allows larger integers
  - BIGNUM data structures
  - Only limited by available memory
- Luckily, there are handy libraries you can use
  - OpenSSL BIGNUM library
    - http://www.openssl.org/docs/crypto/bn.html
  - GNU MP
    - http://gmplib.org/

Symmetric Systems

An introduction

Symmetric cryptosystems

- Saw that asymmetric algorithms based on hard problems
  - Solving the problem finds the key
- Symmetric systems are based on principles of confusion and diffusion
  - Confusion: obscures the relationship between plaintext and ciphertext
  - Diffusion: spreads plaintext statistics through the ciphertext
    - A bit of the output is influenced by many bits of the input
      - Both forward and backward diffusion are important!
  - Finding the key is… different

- Niels Ferguson once told me that cryptanalysis was half mathematics and half black magic
  - Everyone here likes magic, right?
- Let's learn some tricks

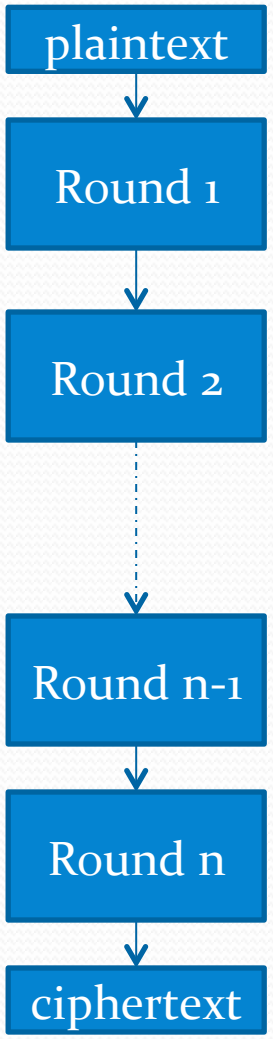
Little bit of math, little bit of magic

- First, we'll look at a couple kinds of constructions
  - Not an exhaustive list, but you're likely to see these in practice
- Then we'll move onto the math and "magic" we need to analyze them

- The math
  - Probability
    - A dash of statistics
  - Linear algebra
- The magic
  - Slide attacks
  - Linear cryptanalysis
  - Differential cryptanalysis
  - Integral cryptanalysis

Common Construction

- Product cipher
  - Combines two or more transformations
  - Resulting cipher more secure than components
  - Simple function f
- Algorithm = f ○ f ○ f ○ f ○ f ○ … ○ f ○ f
  - ○ is composition
  - f(f(f(f(f(….f(f(x))) …)))))
  - f is called a round function



Round function

- Provides confusion and diffusion
- Key combined with state (confusion)
  - Often called "key mixing"
- Non-linear transform
  - Often implemented as a substitution
    - S-box
- Data mixing/permutations (diffusion)
  - Each bit of output depends on many bits of input
    - In the best case, all

- One round doesn't have to achieve high confusion and diffusion
- High confusion and diffusion achieved by repeated application of round function
  - There may be multiple unique round functions

Round function (continued)

- Composition of simple function has benefits over single complicated function
  - Several optimization options available
    - Pick the best implementation strategy for the target
      - 32-bit architecture
      - 8-bit architecture
      - Limited memory
    - Loop unrolling
    - Smaller circuit size

- We'll talk about two round function constructions
  - Substitution permutation network
  - Feistel

Substitution permutation network

- Substitution layer comprised of S-boxes
  - Takes in a small number of bits (i.e. a byte)
  - Outputs a small number of bits
  - Relation between input and output is complex
    - High-degree polynomial
    - **Not linear**
- Permutation layer
  - May also be called a P-box
  - Shuffles all the bits of the state
    - May do so in chunks

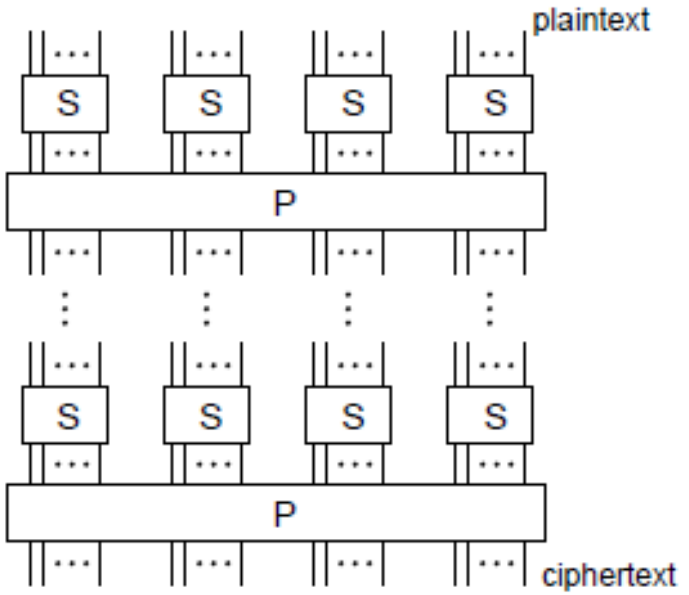


Figure 7.7: *Substitution-permutation (SP) network.*

Image is from the HAC

A closer look at permutations

- Let's look at three different permutations
  - Suppose each "chunk" is the input/output of an S-box on previous slide
- Which provides most diffusion? The least?
  - Draw it out to see

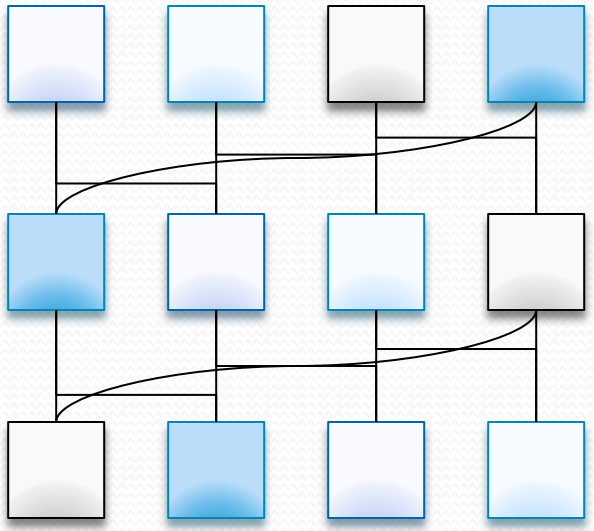
x	0	1	2	3
P(x)	1	0	3	2

x	0	1	2	3
P(x)	1	2	0	3

x	0	1	2	3
P(x)	1	2	3	0

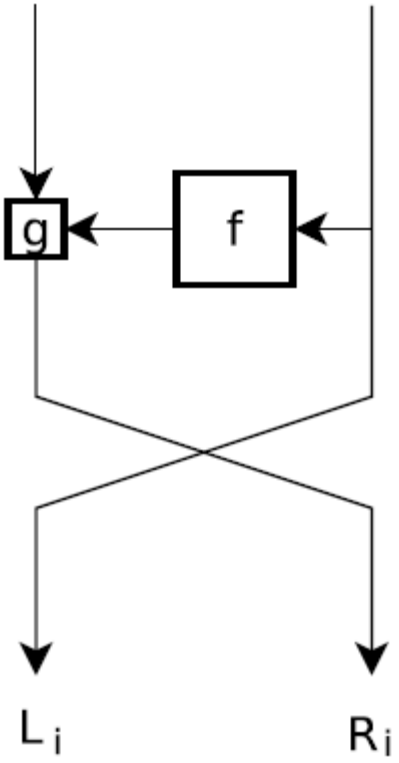
A closer look at permutations (continued)

- Ok, that was a trick
  - They were all equally terrible
- Since the same groupings just go from S-box to S-box, all of them have a simple form
- To provide diffusion, the permutation has to mix data between S-boxes
- A cipher with a permutation layer like this is just begging to be attacked
  - More on this later



Feistel

- Main features
  - State is split in two parts
    - Non-linear transform performed on one part
    - Result combined with other part
    - Parts swap
  - Encryption and decryption use the same round function logic
    - Less code
    - Fewer gates
- Each part is usually half
  - Called a balanced Feistel function



Key schedule

- We've talked about types of round functions, but need one more piece
- Key isn't usually applied as-is each round
- It goes through a function to produce a *key schedule*
  - Set of keys
    - Can be generated as needed, or before first round is applied, depending on algorithm
  - Each round uses one of these keys
  - When decrypting, they are applied in reverse order from encryption
    - True in Feistel ciphers as well

EASY1

- Toy cipher in the textbook
  - Diagram on page 101
- Features
  - SPN construction
  - 36-bit blocks
  - 18-bit key
    - Creates 36-bit key where upper and lower 18-bits are identical
- Most of the Python code is on pages 103-106
  - Missing from this section: apbox and asbox (part of decryption)
    - They show up in later pages
  - EASY1.py has all the code you need to run EASY1
    - Object-oriented version of what is in the book

EASY1.py

- To use:
  1. Create an EASY1 object
     - cipher = EASY1 ()
  2. Encrypt
     - Arguments are plaintext, key, and number of rounds
     - C = cipher.encrypt(123L, 456, 1)
  3. Decrypt
     - Arguments are ciphertext, key, and number of rounds
     - Mp = cipher.decrypt(C, 456, 1)
       - Mp should give you 123 back
- In this implementation, you can give encrypt/decrypt either an 18-bit key or 36-bit key
  - Will create 36-bit key from 18-bit
  - Checks form of 36-bit key
- Alright – let's try it!

Exercise

- Objective: use the EASY1 cipher in EASY1.py

- Open a new file and write a script that does the following
  - Encrypts the message 123456 with key 9876
  - Decrypts the message
  - Reports an error if the decrypted message does not match the original

Ready?

- Now that we've got that down, let's get down to some analysis!

Generic attacks

For symmetric systems

The extremes

- Assume Eve discovers a plaintext-ciphertext pair
- Two extreme methods for finding the key

- Brute force
  - Exhaustive key search
  - Try each possible key on single plaintext/ciphertext
    - Message not known ahead of time
  - Takes O(1) memory and $O(2^k)$ operations, where the key has k bits
- Massive pre-computation
  - Pre-compute all possible ciphertexts for P
  - Takes $O(2^k)$ memory
  - After pair observed, perform a lookup
    - Assuming lookup is constant, takes O(1) operations

Hellman's time-memory trade-off

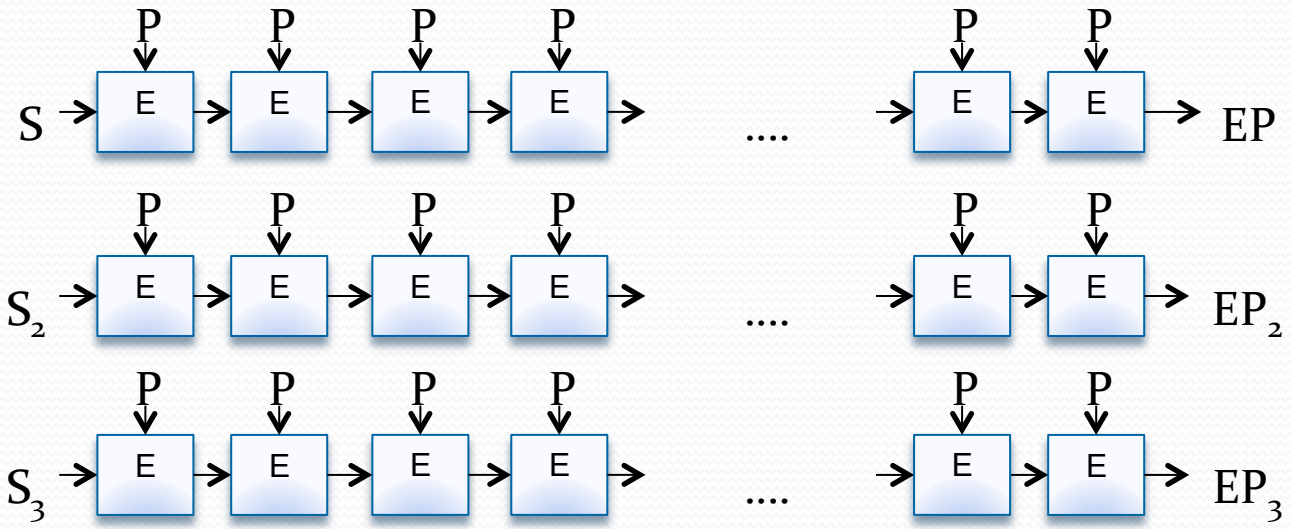
- Also called time-space trade-off
  - Memory is what is meant by space
- Middle ground between brute force and massive pre-computation
- Main idea: create chains of encryptions, and only store the start and end of each chain

- This is best described visually

TMTO

1. Choose a starting point, S
2. Choose a plaintext, P
3. C = E(P,S)
   - The result becomes the key for the next encryption in the chain
4. Repeat until endpoint, EP, reached
5. Go back to step 1

If C has more bits than the key, then a reduction has to be performed before the next encryption



TMTO (continued)

- The attacker stores each (S, EP) pair
  - May do this for several different plaintexts, but need to keep track of which chains associated with which P
  - Let's say there are t pairs
- Now we move from the pre-computation to the attack

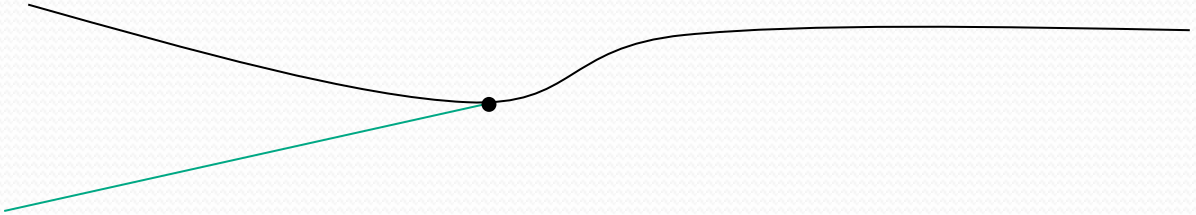
1. Obtain a plaintext-ciphertext pair, where the plaintext is P
2. Compute an encryption chain starting with the ciphertext
   - Keep track of the number of encryptions performed, i
3. When an endpoint is reached, know which chain you're on
4. Calculate forward from the starting point until C_{t-i-1} reached

Convergence, cycles, etc.

- Chains do not always run in parallel
- Sometimes they converge
  - Multiple starting points terminate in the same endpoint
- Sometimes they form cycles
- In both cases, work is performed that does not provide any new information
  - Waste of time
- Workaround: add some randomness
  - Choose a permutation function, F, for each chain
  - Calculate $C_i = F(E(P, C_{i-1}))$

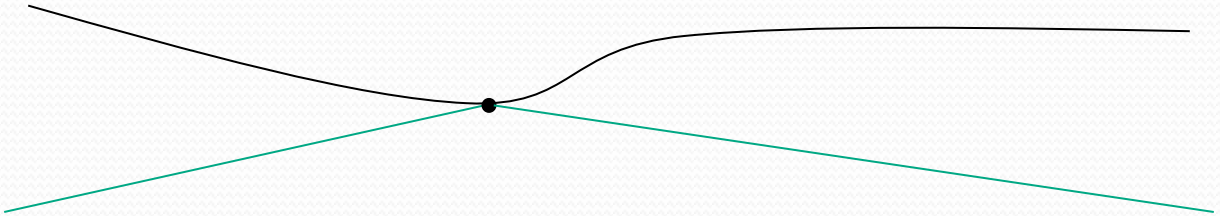
The effect of F

- Without F functions



  - From here, the second chain duplicates the first

- With F functions



  - Because F is different in each chain, they diverge again

F and reality

- Using a different F for each chain brings up a problem
  - Have to now store each of these functions
- Alternative: choose r different functions
- For each function, construct m chains
  - Each chain should start at a random point
- This produces r tables with m chains of length t

FindChains algorithm

for i=0 to r-1

choose random function F_i

for j=0 to m-1

$SP_{ij} = rand(2^k)$

$C_0 = SP_{ij}$

for L = 1 to t-1

$C_L = F_i(E(P, C_{L-1}))$

$EP_{ij} = C_{t-1}$

These algorithms adapted from "Applied Cryptanalysis" by Stamp and Low

findEP algorithm

Input: C and (SP,EP) pairs
Output: corresponding EP or failure

```
findEP(C)
        for i=0 to r-1
                Y=F_i(C)
                for j=1 to t
                        for L=0 to m-1
                                if Y == EP_iL
                                        found = findKey(i,j,L)
                                        if found
                                                return found
                                Y=F_i(E(P,Y))
        return failure
```

findKey

Inputs: i (table number) ,L (chain number), j (position in chain), ciphertext C

Output: key or failure

```
findKey(i,L,j)
        Y = SP_iL
        for q=1 to t-j-1
                Y=F_i(E(P,Y))
        K=Y
        if C == E(P,K)
                return K
        else fail
```

How many chains?

- That depends on how certain you want to be that you'll get the result
- Longer chains and more chains will cover more of the key space
  - More chains means more memory
  - Longer chains means more computation in both pre-computation and attack phases

- Hellman suggested the following for a k-bit key
  - $2^{k/3}$ tables
  - $2^{k/3}$ chains
  - $2^{k/3}$ iterations per chain
- Probability of success = $1 - e^{-mtr/2^k}$
  - With above, this is 0.63
  - Assumes not cyclical and not converging
    - Bad chains reduce success probability

Exercise

- Implement TMTO on EASY1
- Open TMTO.py
  - Finish the following functions
    - F
    - findChains
    - findKey
    - findEP
- Objectives
  - Analyze the chains of EASY1
    - Derive the chains, but also print out some of the intermediate points
    - This will help you debug your code!
  - Use TMTO to find the key

Exercise: tips and tricks

- You can create a “random” function like this
  - Fi = "(({0}*3+{1}*{2}) % 2**25) & 0x3FFFFL".format(x,i,r)
  - res = eval(Fi)
  - This is an example, feel free to use your own!
- You may want to try some things to ensure your code is functioning properly
- Some suggestions:
  - Make sure the key is inside a chain by fixing SP[i][j] to the key
    - Tests when the key is an SP
  - Print out where in a chain it is
    - Tests when the key is inside a chain
      - i.e. not an SP

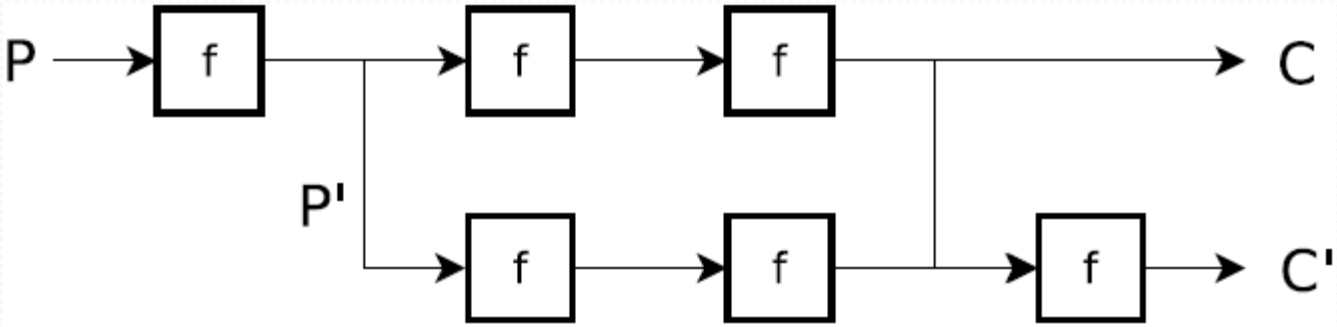
Myth: more rounds equals more secure

- Fact: adding more rounds can increase security, but it is not inherently given

- Let's see why

Slide attacks

- Suppose all round functions are completely identical
  - All have same round key as well (key schedule is just repeated applications of the key)
- Then m round functions can be reduced to one in analysis
  - Need to find ***slid pairs***
  - Plaintexts P and P' that produce ciphertexts C and C' (respectively) such that P' = f(P) and C' = f(C)



Slide attacks (continued)

- The trick here is realizing when you have a slid pair
- May be difficult with an SPN

- Feistel constructions are simpler
  - Suppose right side is transformed during round
  - Then left side is unchanged, and swapped with the right
  - Old left side is new right side

  - Aren't there false positives?
    - Of course
    - The probability that this happens in both the (P,P') pair and (C,C') pair is low
      - Much higher if you are only considering one

I get it now!

- The purpose of the key schedule is to ensure that each round is not 100% identical
- The round transformation might be the same, but applying a different key will yield different results
- The round key should be unique for each round
  - If there are two round keys alternate, then the slide attack can be done by looking at two rounds instead of one
  - It is the repetition that causes vulnerability

Exercise (part 1)

- All rounds of EASY1 are identical
  - Let's try a slide!
  - We'll do this in two parts

- Open slide.py
  - Assume you're in a chosen plaintext model
  - Complete Slide.CreateSlidPair()
- Objectives
  - Identify how to determine when you have found a slid pair
  - Write code that finds the key, given a slid pair
- Tips
  - You can request a single round encryption using self.key as key argument
  - The key will change each time you run the code, so look at what is always the same

Exercise (part 2)

- Let's change models
- Now assume that you are collecting plaintext-ciphertext pairs, but cannot make requests
  - i.e. you have to generate both plaintexts randomly
- Complete Slide.findSlidPair()
  - Now you need to recognize a pair instead of creating one
  - Generate random plaintext and compute ciphertext

- Objective
  - Reduce a 20-round cipher to one round and find the key
    - This means no more 1-round requests with the key – 20 rounds only!
- Tips
  - You can still make one round requests with a fixed key of your choosing, just not the randomly selected one

More math

Just a little

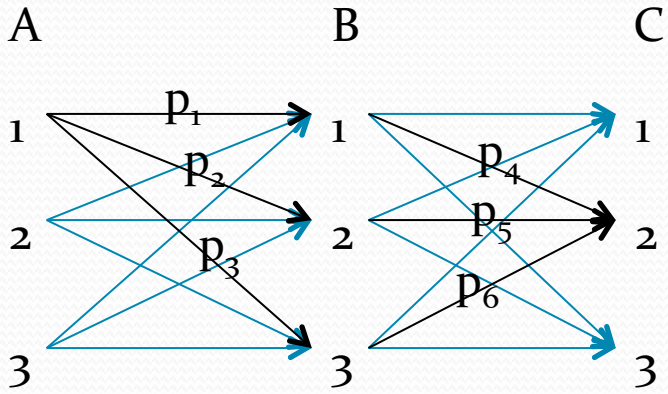
Product groups

- We've talked about groups
  - If they commute, they are abelian groups
- A product group is constructed from several groups
  - Suppose (G,+) is an abelian group
  - $G^n = G \times G \times \cdots \times G$
  - Recall when we did this with CRT
- Example:
  - Consider G = {0,1}
  - $G^2 = G \times G$ (two bits)
  - Each element of is a G^2 2d vector
    - (0,0), (0,1), (1,0), (1,1)
    - u + v = w means $u_i + v_i = w_i$
      - (1,0) + (0,1) = (1,1)
      - Exclusive-or, in this case

Linearization

- Attacks often reason in a linearized version of a cipher
- An n-bit integer $\rightarrow Z_2^n$
- Working with vectors of bits

Probability



- Random variables take on values according to a probability distribution
  - Probability that a letter in English is "e"
  - Probability that a letter in English is "z"
  - If X is the random variable, write Pr(X=e) and Pr(X=z)
- In image above, what is Pr(C=2) if we know that A=1?
  - $Pr(B=1|A=1) = p_1$, $Pr(B=2|A=1) = p_2$, $Pr(B=3|A=1) = p_3$
  - $Pr(C=2|B=1) = p_4$, $Pr(C=2|B=2) = p_5$, $Pr(C=2|B=3) = p_6$
  - $Pr(C=2|A=1) = Pr(C=2|B=1)\ Pr(B=1|A=1) + Pr(C=2|B=2)\ Pr(B=2|A=1) + Pr(C=2|B=3)\ Pr(B=3|A=1) = p_1p_4 + p_2p_5 + p_3p_6$

Statistics

- The only distribution we care about right now is the uniform distribution
  - Distribution of a perfect cipher
- Histograms are the only tool we need
  - No Z-tables or t-tables for us!

- We will only be concerned with
  - Values that occur far from expected in a uniform distribution
  - Values that occur an expected number of times

Matrix addition

- Two matrices, A and B, with same dimensions
- C = A + B is computed by component-wise addition
  - $C\downarrow i,j = A\downarrow i,j + B\downarrow i,j$
  - $[\blacksquare 1\&2@3\&4\ ]+ [\blacksquare 5\&6@7\&8\ ]= [\blacksquare 1+5\&2+6@3+7\&4+8\ ]= [\blacksquare 6\&8@10\&12\ ]$

- In Wolfram Alpha (www.wolframalpha.com) the syntax to compute this is
  - {{1,2},{3,4}}+{{5,6},{7,8}}

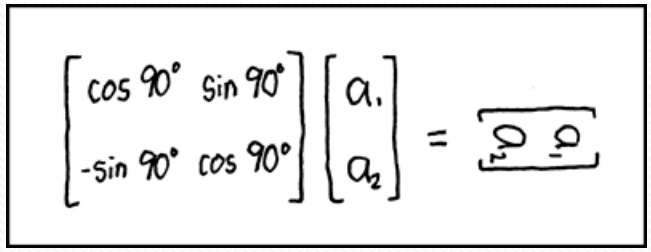
Matrix multiplication

- A has dimension m x n and B has dimension n x p
  - The number of columns in the left matrix must match the number of rows in the right matrix
  - Note that this operation does not commute, so you can't just swap left and right
    - May need to transpose
- A x B = C
  - $C_{i,j} = \sum_{k=1}^{n} A_{i,k} B_{k,j}$
  - $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1*5+2*6 \\ 3*5+4*6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$
- In Wolfram Alpha, the syntax to compute this is
  - {{1,2},{3,4}}*{{5},{6}}

Caveats

- It is pretty straightforward with regular addition and multiplication
- Remember that in our abstract algebra world, + and x might mean something else
  - + might be exclusive-or or modular addition
  - x might be a different kind of multiplication
    - In AES, it is multiplication in a Galois Field
      - Polynomial fun

Some xkcd math humor



http://xkcd.com/184/

Linear & differential cryptanalysis

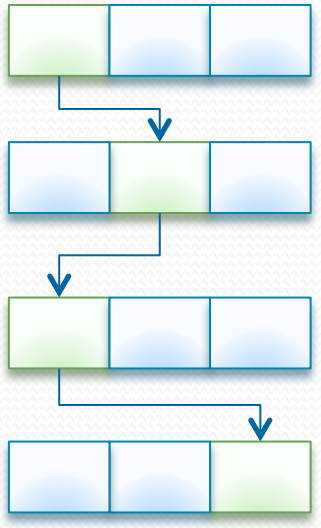
Symmetric systems

Attacking symmetric systems

1. Find a simple way to express a relationship between input bits and output bits
2. Use math-fu to find key data
3. Profit

Partitioning is the key concept

- When you're targeting a particular algorithm, you can do more than in generic attacks
- A significant difference between asymmetric and symmetric systems is the ability to partition the state and still retain information
- Reduce your work
  - Isolate parts of the state/key as much as possible
  - If the part you're interested doesn't use a set of key bits, then they can be ignored
  - $2^{n/3} + 2^{n/3} + 2^{n/3}$ is a lot smaller than 2^n
    - Example: n=9
      - 8+8+8 = 24 vs. 512
    - This is why diffusion is important
      - Prevents adversary from isolating sections

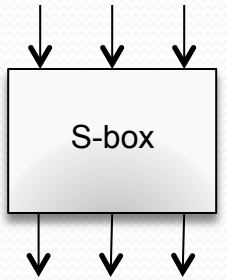


Linear cryptanalysis

- Working with vectors of bits
  - 1011 → (1,0,1,1)
- Operation is exclusive-or
- Suppose cipher has n-bit input and output, m-bit key
  - Write the plaintext as $P = (p_0, \ldots, p_i, \ldots, p_{n-1})$
  - Write ciphertext as $C = (c_0, \ldots, c_i, \ldots, c_{n-1})$
  - Write key as $K = (k_0, \ldots, k_i, \ldots, k_{m-1})$
- Main idea: approximate part of a non-linear function using a linear expression
  - $p_0 \oplus p_{20} \oplus c_9 \oplus c_{24} \oplus c_1 \oplus k_0 \oplus k_1 \oplus k_7 = 0$
  - Tells us the parity of 3 bits of key
    - $p_0 \oplus p_{20} \oplus c_9 \oplus c_{24} \oplus c_1 = k_0 \oplus k_1 \oplus k_7$
- Note that if 1 appears in the equation, it is affine
  - Won't work here

How do you approximate a function?

- Start with the non-linear elements
  - Linear elements are easy to write as linear expressions
    - No approximation needed
- In an SPN, this will be the S-boxes
- Try to approximate the output in terms of the input
- Let's consider the following 3-bit S-box
  - 3-bits in
  - 3-bits out



input	0	1	2	3	4	5	6	7
output	3	7	2	4	1	5	0	6

Linear S-box approximation

input	0	1	2	3	4	5	6	7
output	3	7	2	4	1	5	0	6

- First, note that there are $2^3 * 2^3 = 2^{3+3} = 2^6 = 64$ different expressions
  - You can see how this grows with larger S-boxes
- Use a mask to determine if a bit is part of the expression
  - A mask of 101 means that the most significant and least significant bits are part of the expression
  - Input: 101, output 110 translates to $x_0 \oplus x_2 = y_1 \oplus y_2$
    - Equivalently, $x_0 \oplus x_2 \oplus y_1 \oplus y_2 = 0$
- Need to try all possible input output pairs
  - $2^6 * 2^3 = 512$ operations
- Count how many times the equation is true

Linear expression bias

- The usefulness of a linear approximation is based on its bias
  - The expectation is that the probability an approximation holds is 0.5
  - If an approximation holds significantly more or less often than this, it can be used to find the key

- Let T be the number of times an expression holds (is true)
- Let N be the number of trials (distinct plaintext-ciphertext pairs)
- Let ϵ represent bias
- Then: $\boldsymbol{T/N} = 1/2 + \epsilon$
- Which means: $\boldsymbol{T/N} - 1/2 = \epsilon$

Two notations for bias

- Bias should range between -0.5 and 0.5
  - 0.5 - 0.5 = 0 (approximation never holds)
  - 0.5 + 0.5 = 1 (expression always holds)

- People often don't like dealing fractions
  - Easier to write code for integers
- Alternate definition
  - Bias = N($\boldsymbol{T/N} - 1/2 = \epsilon$) = T – N/2
  - This one is not normalized, but can be useful, as we'll see

Linear round approximation

- Once the non-linear components are approximated, the rest is usually much simpler
- Trace the bits of your approximations through the rest of the round
  - Input and output masks for the round
- Once you have input and output masks for one round, extend to another
  - And so on, as far as you can go with a reasonable bias

What really happens

- Don't usually use ciphertext bits in approximation
- If you're looking to break R rounds, you need an expression for R-1 rounds
  - The "ciphertext" of the approximation is the output of round R-1
  - The ciphertext is the output of R rounds
- From the ciphertext, backup to the output of round R-1
  - This requires guessing some bits of the last round key
  - These guesses are going to be where you get your power
    - Otherwise, you'd just get 0 or 1

Matsui's algorithm 1

- Given an expression with probability p of the form

  $P_{i_1} \oplus P_{i_2} \oplus \ldots \oplus C_{j_1} \oplus C_{j_2} \oplus \ldots = K_{k_2} \oplus \ldots \oplus K_{k_m}$

1. Collect N plaintext-ciphertext pairs where encryption was performed under the same key
2. For each pair, calculate the left side of the equation
   - Let T be the number of times the left side is zero
3. If $T > \boldsymbol{N/2}$ and $p > \boldsymbol{1/2}$, guess that the right side is zero
   - This means that the parity (or exclusive-or sum) of the selected key bits is zero
4. If $T < \boldsymbol{N/2}$ and $p < \boldsymbol{1/2}$, guess that the right side is zero
5. Otherwise, guess that the right side is one

Matsui's algorithm 1 (continued)

- Pros
  - Low implementation cost
    - Nothing but xor operations
- Cons
  - Must know probability, p
  - This algorithm gives only the parity

- He gave a second, more useful algorithm

Matsui's algorithm 2

- Given an expression of the form

  $P_{i_1} \oplus P_{i_2} \oplus \ldots \oplus C_{j_1} \oplus C_{j_2} \oplus \ldots = K_{k_2} \oplus \ldots \oplus K_{k_m}$

1. Collect N plaintext-ciphertext pairs
2. For each candidate set of key bits, calculate
   - T = # times true
   - $\epsilon = |T - N/2|$
3. Select key candidates that have the highest bias, ϵ
   - These are more likely to be correct, by principle of maximum likelihood

Matsui's algorithm 2 (continued)

- Pros
  - Gives [partial] key candidate rankings in addition to parity
  - Don't need to know the probability
- Cons
  - Requires N calls to the cipher for each candidate key

- Comparison
  - 1 takes less work, but yields less information
    - You'll have to make up the work later if you want to find the key
  - It is assumed that the attacker knows everything but the key, so access to the encryption function is assumed
  - Key bits with the same parity may not be equally good. Algorithm 2 will rank these, but algorithm 1 will not

What does bias look like?

- 2-dimensional table
  - Row: input mask
  - Column: output mask
- Bias and masks often written in hex
- (0,0) is always equal to N/2, so ignore it
  - Involves not input or output bits
  - More for completeness than anything else
- Example
  - A 2-bit S-box may have the following table

	0	1	2	3
0	2	0	0	0
1	0	-1	1	1
2	0	1	0	0
3	0	0	1	-1

Exercise in three parts

- We're going to try linear cryptanalysis on two rounds of the EASY1 cipher
- Break this down into three parts
  - Find biases of linear expressions for the s-box
  - Extend approximation to two full rounds
  - Use expressions to find key bits

Exercise: part 1

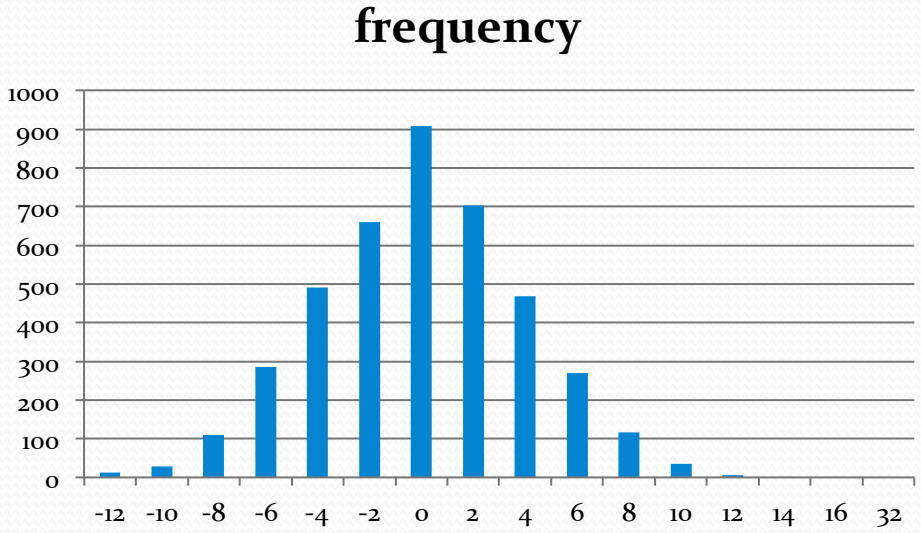
- In this part, we only care about the S-box
- We're going to use the bias notation T-N/2, since it is easier to code

- Objectives:
  - Calculate S-box bias for all possible expressions
    - Fill in function findBias
  - Create a frequency table for each bias, and display
    - Fill in function findFrequency
  - Identify high-bias expressions
    - At least the top 4
    - Use findMasks

- Hints
  - Setting a bit to zero has the same effect as not including it in the expression, so use AND operations to zero out any unneeded bits
  - You can use decimal or hex, just be consistent
  - Don't look at next slide yet

Exercise: part 1 discussion

- The highest bias is 16
  - Remember, the bias in cell (0,0) doesn't count!
- 14, 12, and -12 are next highest
  - 12 and -12 are equally strong
- Frequency isn't actually part of the attack
  - Observe how rare high-bias expressions are compared to low-bias
- What matters is strong biases involving a small number of bits

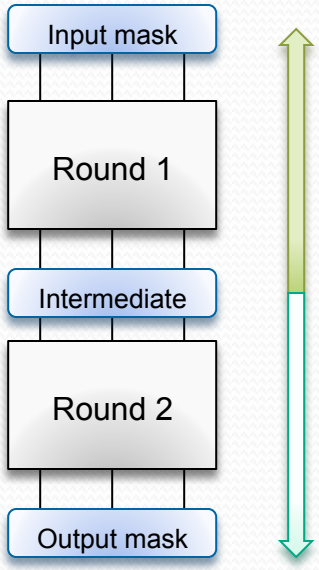


Exercise: part 2

- Look at table 6-4 of the text (page 177)
  - You see some expressions have smaller bias, but also fewer addends
- Choose 2 of these expressions to try in this part
- Objectives:
  - Extend these expressions to find input/output masks for one full round
  - Extend one round to two rounds
- Hints
  - When extending to one round, follow bits of S-box output mask forward
  - When going from one round to two, follow the first round input bits back (find dependencies)
  - This is a pen & paper exercise
  - You can use the image in the book to trace the paths
  - It may take you a couple tries and different S-box approximations for each round
  - Don't worry about the key yet – we'll get there

Exercise: part 2 (more hints)

- There are really three masks here
  - Input to two rounds
  - Output of two rounds
  - Between rounds
- A lot of bits in the middle may mean more in the input or output
  - Set the middle to either the left or right half of each equation
  - Look carefully equations and how they trace in both directions
- Minimizing bits in key bits used will save time in key recovery step
  - You don't need to find the minimum, but try your best
- Mind the permutations!
  - Bit x of the output in one S-box in not necessarily bit x of input to the next
- This may take a while, so don't get discouraged if it takes a few tries



Sidebar: what is the bias now?

- We still have one more part to the exercise, but now we have new expressions
  - What is the bias?
- Let's first go back to the normalized notation of bias
  - Divide each of the biases we were just using by 64
    - 32 → ½, -12 → - 0.1875, etc.
- Let $\epsilon 1$ be the bias of the first round expression, let $\epsilon 2$ be the bias of the second round expression, and let r be the number of expressions
- The new combined bias is $\mathbf{2\uparrow r-1}\ (\epsilon 1 * \epsilon 2)$
  - This is due to Matsui's piling-up lemma

Exercise: part 3

- Now it is time to find those key bits!
- Use the two-round approximation from part 2 to perform a 3-round attack
- Backup from ciphertext to third round input using guessed key bits
  - You only need to guess the key bits that are in the expression and required to backup
- Objectives
  - Fill in attack()
    - Create equations in format for Matsui's algorithm 2
    - Implement algorithm 2
    - Find bias for each candidate, and sort
    - Which key bits are most likely?

Exercise: part 3 (continued)

- Hints
  - If you need an expression, try using the S-box expressions on page 179
  - Use grab(x, y) to select bit y from state x
  - If you get stuck, complete code starts on page 189
    - Please try to do as much as you can on your own

Differential cryptanalysis

- Linear cryptanalysis deals with approximating values
- Differential cryptanalysis deals with differences
  - It doesn't matter what the values are, as long as the correct difference is present
    - Now need pairs of plaintexts and pairs of ciphertexts
      - Instead of looking at (p,c), looking at (p,p') and (c,c')
- Requires a stronger adversary
  - Need input, output, and ability to request encryption/ decryption operations
    - Chosen plaintext attack (if encryption access)
    - Chosen ciphertext attack (if decryption access)
  - If a protocol is really poor, adversary may be able to perform attack without requests
    - Shouldn't happen

Differences

- Suppose we're using an encryption oracle
  - This means that we ask something for the encryption of a message, and it gives us the result
  - We don't know the key, the oracle does
- Have two plaintexts, p and p', such that $p \oplus p' = \Delta_p$
  - Δ_p is input difference
- Encrypt plaintexts under unknown key by asking oracle
  - c = encrypt(p)
  - c' = encrypt(p')
- Calculate output difference
  - $\Delta_c = c \oplus c'$

Differences (continued)

- A particular input difference leads to particular output difference with some probability
  - Written $\Pr[\Delta_p \rightarrow \Delta_c]$
  - $\Delta_p \rightarrow \Delta_c$ is called a characteristic
  - The probability of a characteristic is determined through analysis
    - Unlike linear approximations, these differences are exact
- The key that produces the expected output difference with closest to the expected probability wins
- Like linear cryptanalysis, we begin with the non-linear components

S-box characteristics

- Try all possible input difference and output difference pairs
- Count how many times each pair occurs
  - The probability of the pair (characteristic) is the count divided by the number of combinations
  - For EASY1, divide by 64
- Influence of the key needs to be addressed
  - In EASY1, key addition step is
    - $(X \oplus K) \oplus (X' \oplus K) = X \oplus X' \oplus K \oplus K = X \oplus X'$
    - Key does not change analysis
      - Not necessarily true in all ciphers!
      - It depends how the key is mixed in

Extending difference characteristics

- Chain characteristics together
  - Particularly easy if input difference equals output difference
  - If you want to add a round and have output difference x, choose a high probability with input characteristic x
- As in linear cryptanalysis, we want a differential that does not go to all R rounds
  - We want to make key guesses and backup, then check difference
- Probabilities multiply
  - Probability of a → b → c is Pr[a → b]*Pr[b → c]
  - If there are multiple ways to get to c from a, chain the probabilities like we did before

- Food for thought
  - Higher probability characteristics require fewer requests for attack to work well
  - Need high overall probability for multiple rounds
    - A high probability followed by a low one may be worse than two weaker ones
    - 0.9*0.1 = 0.09
    - 0.5*0.5 = 0.25

Exercise (part 1)

- Open differential.py
  - Complete genMatrix()

- Objectives:
  - Find probability for all possible characteristics over S-box
  - Identify high-probability characteristics
    - At least the top 2
    - More if you're up to it

Exercise (part 2)

- Implement the attack we described
  - createPairs()
  - differentialAnalysis(…)

- Tips
  - Try duplicating the example in the book
    - Key ox555555555
    - Random number generator is already seeded for you

Summary of linear and differential cryptanalysis

- Linear
  - Collect plaintext-ciphertext pairs
  - Approximate cipher using linear equations
- Differential
  - Make encryption/decryption requests
  - Calculate difference probabilities

Integral cryptanalysis

Symmetric systems

Integral

- We've looked at linear equations
- We wouldn't want calculus to feel left out, now would we?
- Relax, you don't need to remember how to find the integral of a continuous function
  - We're all about discrete math here

AES

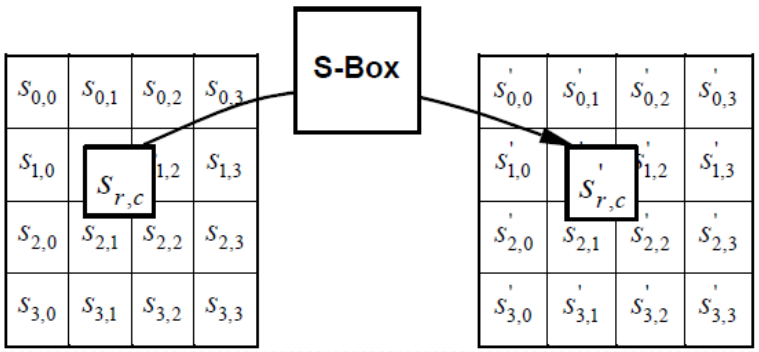
- We will look at this attack in terms of 4-round AES-128
  - The full cipher is 10 rounds

- Anatomy of the Advanced Encryption Standard
  - Round consists of nonlinear step and linear mixing
  - Round structure has 4 parts
    - SubBytes
      - S-box
    - ShiftRows, MixColumns
      - Linear mixing
    - AddRoundKey
      - Combine key with state via exclusive-or
  - 16-byte state is represented as a 2d array
    - 4 by 4 bytes

For full spec, AES-192, and AES-256, see FIPS 197

AES: SubBytes

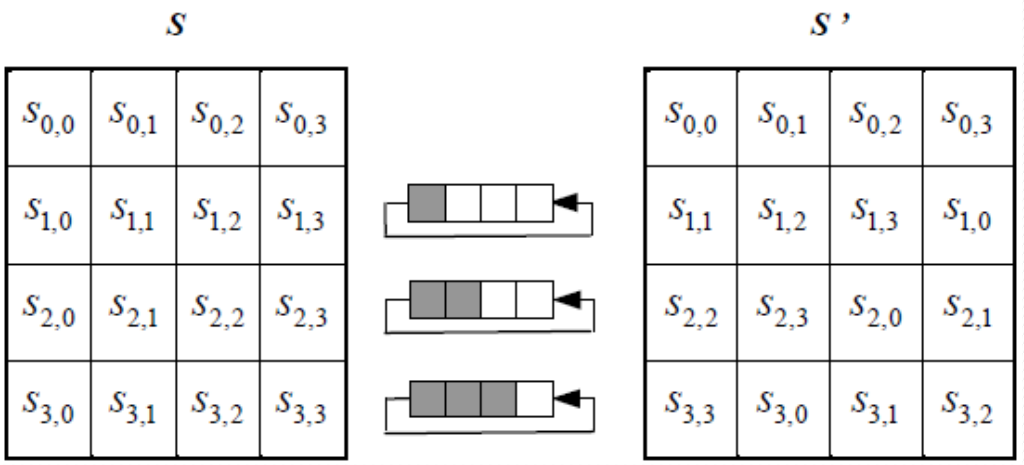
- Byte substitution
  - Usually a 16 by 16 table
    - Split byte value into high and low nibble
      - Byte xy replaced by column x, row y
      - For example, the value “ao” is replaced by the value in row “a” column “o”
    - Efficient in software
  - Can also be calculated on the fly
    - Good for low-memory hardware



x \ y	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

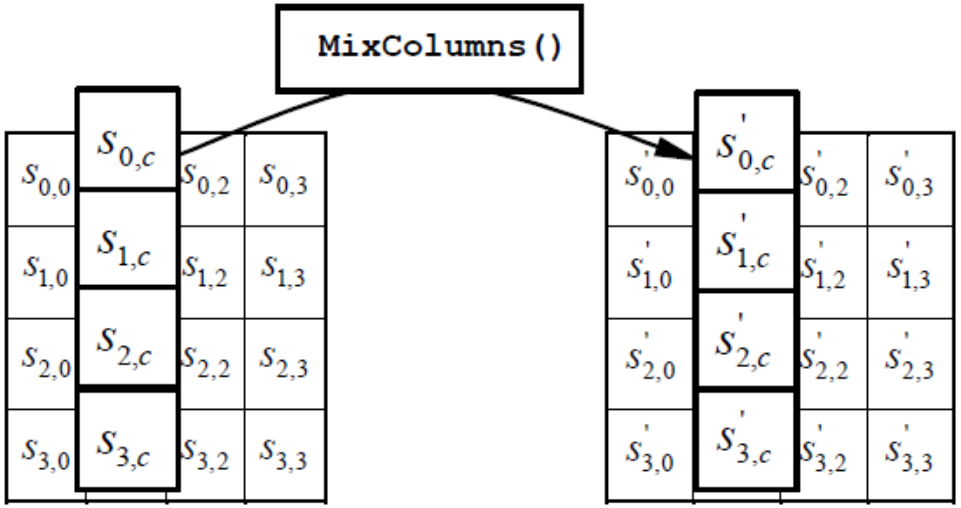
AES: ShiftRows

- ShiftRows is about diffusion moving bytes horizontally
- Each column is broken up



AES: MixColumns

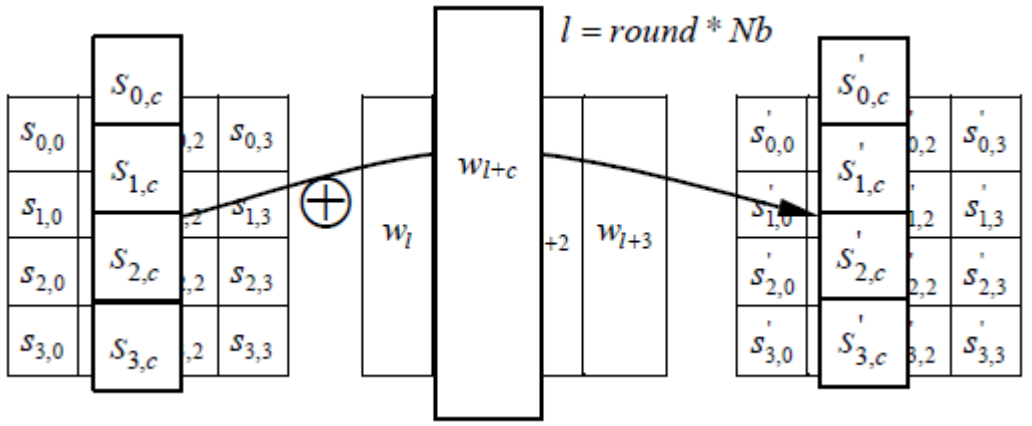
- MixColumns is about column diffusion
  - This is why columns broken up in ShiftRows
    - Can't isolate a single column throughout the round
  - Spread the influence of the bytes
  - Can be expressed as matrix multiplication
- MixColumns is present in all rounds *except* the last one



$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

AES: AddRoundKey

- Byte by byte xor of key and state
- Key is stored in a 2d array as well
- Simply treat each column as a matrix, and add the appropriate column from the key schedule

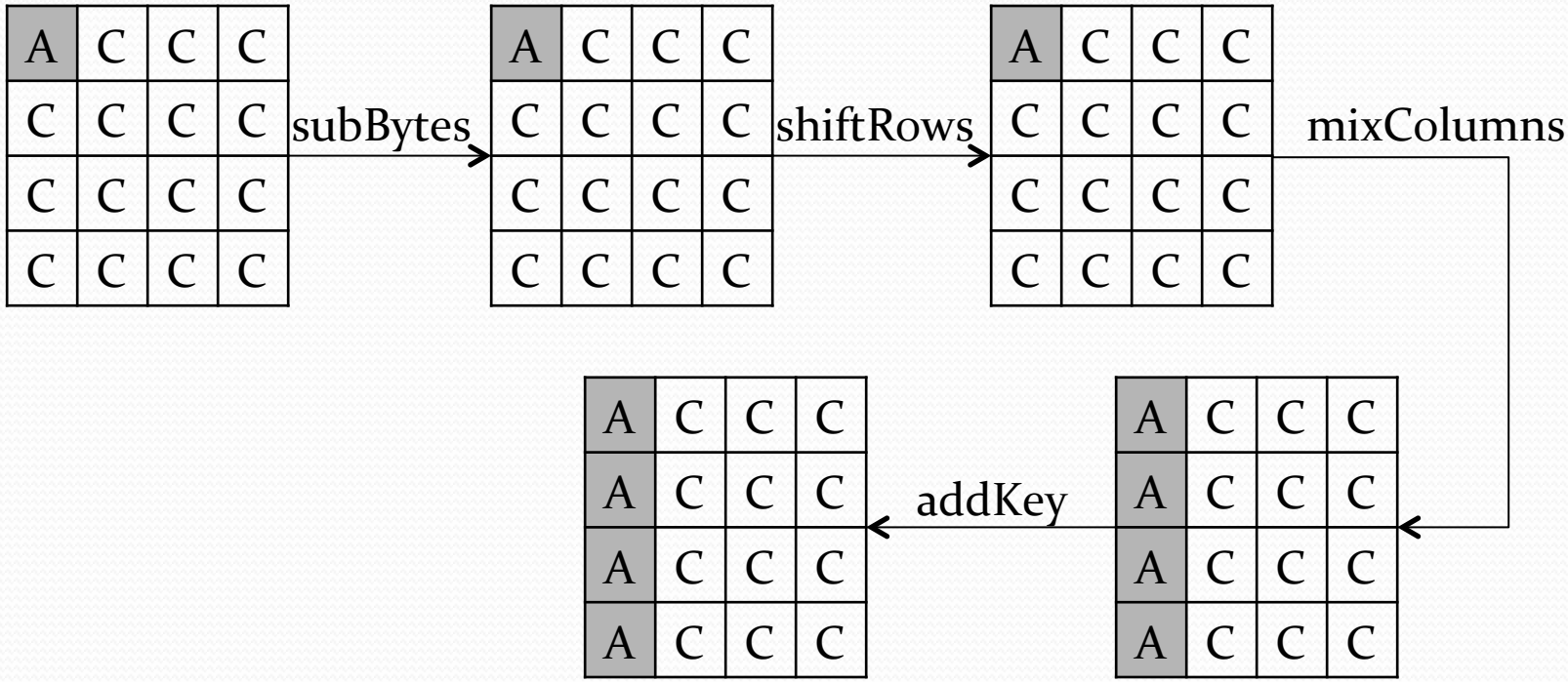


Integral attack on 4 rounds of AES

- Set all but 1 byte of the plaintext to an identical fixed value
- Create a set of 256 plaintexts such that the variable byte takes on all 256 values
- Request the encryptions of all 256 plaintexts, and obtain a set of 256 ciphertexts
- Guess one byte of the last round key, and set all other bits to 0
- Decrypt the last round of all ciphertexts using the candidate round key
- For each byte of state, calculate the xor sum of each byte across the 256 states
  - If the sum is not 0, discard it
  - Otherwise, it is a still a candidate

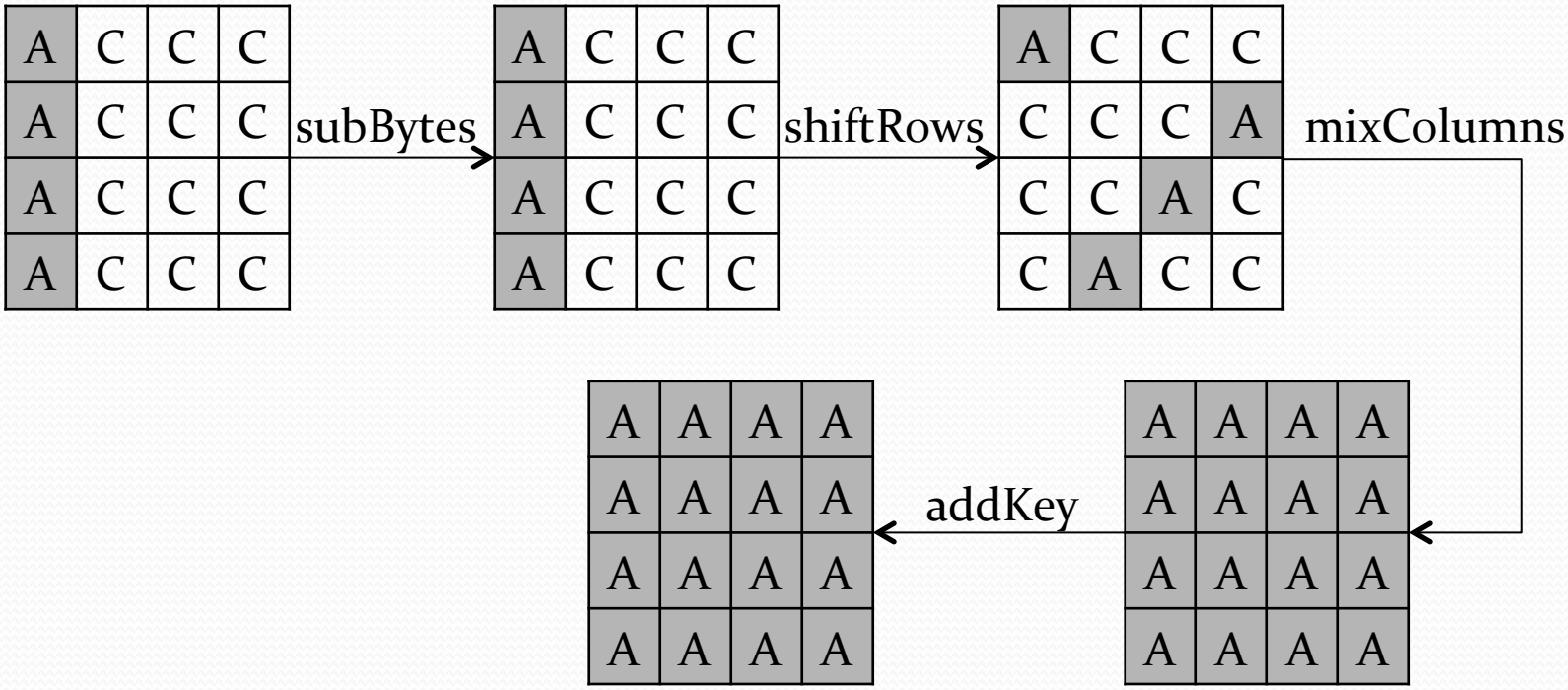
Visual description

- Round 1
  - Suppose we try all possible values for the first byte, and leave all others constant



Visual description (continued)

- Round 2
  - At the end, all bytes take on all values



Visual description (continued)

- Round 3 isn't very interesting to draw, since all bytes will remain all A's
- What is important is that at the end of round 3, all positions have the same sum
  - S=0
  - Sum the same position over multiple single-round decryptions
    - Note that the sum is computed with exclusive-or, since that is the addition operation for this field!
- The last round has no MixColumns operation
  - Only need to guess one byte of the last round key
    - The one that lines up with the sum you're computing

S	S	S	S
S	S	S	S
S	S	S	S
S	S	S	S

Only partial decryption necessary

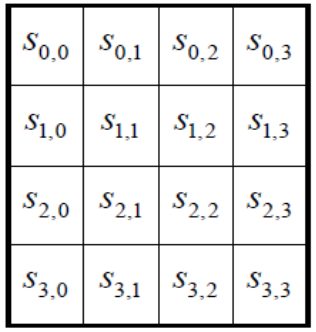
- You don't even need to do a full decrypt of the last round
- ShiftRows changes byte positions
  - You're only concerned with one byte anyway
  - You can ignore this
- Only two parts are necessary to reverse
  - Addition of the guessed key byte
  - Its value through inverse S-box

Finishing the attack

- Repeat this process for all 16 bytes of the last round key
- Each byte will have approximately two candidates
  - One of them is the correct byte
- 2^{16} = 65536 full key candidates
  - So much better than 2^{128}
  - We can iterate through all these quickly
- For each full key candidate
  - Calculate the master key from the last round key
  - Choose a plaintext-ciphertext pair
  - Calculate ciphertext' = encrypt(plaintext, master)
  - If ciphertext' == ciphertext, then master is the secret key

Exercise

- Open up “integral.py”
- All the AES code you need is there
  - encrypt(.) encrypts four rounds
  - backup(.) does a one-round decryption using your guess at the last round key
  - round2master(.) derives the master key from the last round key
  - Plaintext and key inputs are 16-element lists
    - Read into the state column-wise
    - s[0][0] = p[0], s[0][1] = p[4], s[0][2] = p[8], s[0][3] = p[12]
      s[1][0] = p[1], s[1][1] = p[5], s[1][2] = p[9], s[1][3] = p[13],
      s[2][0] = p[2], s[2][1] = p[6], s[2][2] = p[10], s[2][3] = p[14],
      s[3][0] = p[3], s[3][1] = p[7], s[3][2] = p[11], s[3][3] = p[15]



- There is a set of plaintexts and ciphertexts in the code
- Objective: find the key that they were encrypted under

Exercise (continued)

- You'll need to fill in the following functions:
  - Integrate
    - Given an index (0-15), find the integral for that byte
  - Integral
    - Call integrate to find plausible round key bytes
    - Loop over all plausible round key byte combinations
      - Derive master key from round key (use round2master())

- Tips
  - Remember, this is an integral where addition means exclusive-or
  - Pay attention to rows vs. columns

An attack by any other name…

- The square attack on AES is very similar
  - By similar, I mean that it is the exact same thing we just did, but with a different name
- The saturation attack on AES is also very similar
  - Hang on, this is the same attack again!
- People sometimes rename things
- Here's why this attack has (at least) three names
  - AES is based on Square block cipher, which has this attack
    - Hence "square attack"
  - A plaintext byte takes on all values
    - That byte is "saturated"
    - Hence "saturation attack"
  - The ciphertext bytes are summed, which is a discrete integration
    - Hence "integral attack"
- They are all the same in this case, but not in general

Summary: symmetric attacks

- Generic attacks that can be applied to any block cipher
  - Time-memory trade-off
  - Slide attacks
    - Only works if transforms repeat
- Targeted attacks
  - Specific to a cryptosystem
  - Linear
    - Approximate function with linear expressions
  - Differential
    - Use expected difference characteristics
  - Integral
    - Correct key results in expected sum

Closing remarks

Over so soon?

Summary

- Cryptanalysis is the art and science of code breaking
- Modern ciphers are either symmetric or asymmetric
  - Require different attack strategies
- Covered asymmetric attacks
  - Generic attacks on asymmetric systems built on factoring
    - Pollard's rho
    - Pollard's p-1
  - Specific attacks on RSA
  - Generic attacks on asymmetric systems built on the discrete log problem
    - Pollard's rho
    - Index calculus

Summary (continued)

- Covered symmetric attacks
  - Hellman's time-memory trade-off
  - Slide attacks
  - Linear cryptanalysis
  - Differential cryptanalysis
  - Integral cryptanalysis on four-round AES

- I hope everyone had a good time and comes back for more! ☺