Much of cryptography is accomplished by applying invertible functions to data. This may come as no surprise. When a message is encrypted $y = E(x)$ we expect to be able to recover the original message via a decryption function $x = D(y)$. Thus $D(E(x)) = x$ making D the inverse of E.

Function Properties. If we write function signature $f : A \to B$ then we are saying that f is a function with the property that for every x in set A, $f(x)$ is defined and is an element of B. Sets A and B are called the domain and codomain of f, respectively. The range of f is defined as the elements of B that actually get mapped to: $\{y \mid \exists x \in A, f(x) = y\}$.

Example 1. Let $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ and define $f : \mathbb{Z}_5 \to \mathbb{Z}_5$ as $f(x) = x^2 \bmod 5$. Then $f(0) = 0$, $f(1) = 1$, $f(2) = 4$, $f(3) = 4$, $f(4) = 1$, and f is defined for no other inputs. The domain and codomain of f are both $\mathbb{Z}_5$ and the range of f is $\{0, 1, 4\}$. □

A function is *onto* (also known as *surjective*) if every element of the codomain is mapped to by at least one element of the domain. A function is *one-to-one* (also known as *injective*) if every element of the codomain is mapped to by no more than one element of the domain. A function is *invertible* (also known as *bijective*) if every element of the codomain is mapped to by exactly one element of the domain, meaning it is both onto and one-to-one. If a function is invertible, its domain and codomain will be the same size and its codomain and range will be the same set. In Example 1, f is not invertible because elements of the codomain are not mapped-to. There is no x in the domain $\mathbb{Z}_5$ where $f(x)$ is 2 or 3. If $f : A \to B$ is invertible, then the inverse has signature $f^{-1} : B \to A$ and $f(x) = y$ implies $f^{-1}(y) = x$.

Conceptually, you could test whether a function is invertible with a simple process. Make a two-column table with the first column listing each of the function's domain elements and the second column showing what each of the domain elements maps to. If every codomain element is listed exactly once in the second column then the function is invertible, otherwise it is not.

Example 2. Following the procedure just described, we see that the function from Example 1 is not invertible. The second column does not contain every codomain element exactly once.

x	$f(x)$
0	0
1	1
2	4
3	4
4	1

□

A function is a *permutation function* if it is invertible and its domain and codomain are the same set. It has this name because when you write out such a function in tabular form the second column is a permutation of the first column.

Random Functions. When we say "let $f : A \to B$ be a random function" we mean to define f by randomly selecting from the set of all functions with the signature $A \to B$ and defining f to be the randomly selected function. A more concrete conceptualization would use a table representation for f. To select a random function $f : A \to B$, write a two-column table with the first column listing each element of set A once. Then, in the second column write a randomly chosen element of set B in each row.

Example 3. Let $F = \{apple, banana, cherry\}$. To create a random function $f : \mathbb{Z}_4 \to F$, start with a two-column table with the fist column listing each domain element once.

x	$f(x)$
0	
1	
2	
3	

Next, fill each position in the second column with a randomly chosen element of F.

x	$f(x)$
0	*apple*
1	*cherry*
2	*banana*
3	*apple*

□

Once f has been defined it is fixed for the remainder of its existence. The value $f(x)$ is a uniformly distributed element of the codomain for each x in the domain, but repeated evaluations of $f(x)$ return the same value when x is held constant.

Random Permutation Functions. When we say "let $f : A \to A$ be a random permutation function" we mean to define f by randomly selecting from the set of all permutation functions with the signature $A \to A$ and defining f to be the randomly selected permutation function. A more concrete conceptualization would use a table representation for f. To select a random permutation function $f : A \to A$, write a two-column table with the first column listing each element of set A once. Then, in the second column write a randomly chosen element of set A in each row, but without ever repeating an element. The end result of this process is that the second column is a random permutation of the first column.

Example 4. To create a random permutation function $f : \mathbb{Z}_4 \to \mathbb{Z}_4$, start with a two-column table with the fist column listing each domain element once.

x	$f(x)$
0	
1	
2	
3	

Next, fill each position in the second column with a randomly chosen element of $\mathbb{Z}_4$, but each choice must exclude any value already used.

x	$f(x)$
0	1
1	3
2	0
3	2

□

Probability Distribution. To understand the probability distribution of $f(x)$ for a randomly selected f we imagine defining f in a table but in a *lazy* manner. Imagine that instead of fully defining f before it is used we instead start with an empty second column for the definition table. Whenever $f(x)$ is needed by an application of f the table is consulted. If $f(x)$ is already defined in the table its value is returned but, if it is not yet defined, a result

is randomly selected for $f(x)$ and the table is updated. The probability distributions seen by the user of f are identical whether f is initially fully-defined or is defined lazily.

If our lazy table is modeling a random function $f : A \to B$, then $f(x)$ is a uniformly distributed element of B for each x in A. This is clear from our lazy method. For each x in A, $f(x)$ is chosen uniformly from B.

If our lazy table is modeling a random permutation function $f : A \to A$, then the distribution of $f(x)$ changes with each new invocation of f. On the first invocation $f(x)$, the table has no entries in the second column and so $f(x)$ is free to be defined as any element of A. Thus, for all x and y in A, $\Pr[f(x) = y] = 1/|A|$, meaning the first time a random permutation function is invoked its resulting value is uniformly distributed. However, after n different invocations of f, where $f(x_1) = y_1$, $f(x_2) = y_2$, ..., $f(x_n) = y_n$, and $f(x)$ is a new invocation, the table is consulted. Row x is empty but n other rows have definitions. Because f is a random permutation function, $f(x)$ must be chosen randomly from among the $|A| - n$ remaining codomain elements. Symbolically, let $x \neq x_1 \neq \cdots x_n$ and $y \neq y_1 \neq \cdots y_n$ be elements of A, then $\Pr[f(x) = y \mid f(x_i) = y_i \text{ for } 0 \leq i \leq n] = 1/(|A| - n)$.

If $f : A \to A$ is a randomly chosen permutation function then $f(x)$ is a uniformly distributed element of A *for the first invocation of* f because $f(x)$ is free to be any value of A and each value of A is equally likely. A second invocation $f(x')$ on a different input $x' \neq x$ is not uniform, however, because $f(x')$ is not allowed to equal $f(x)$ when f is invertible. To put these facts into symbols, for all x and y in A, $\Pr[f(x) = y] = 1/|A|$, and for all $x \neq x'$ and $y \neq y'$ in A, $\Pr[f(x') = y' \mid f(x) = y] = 1/(|A| - 1)$. The first probability is saying that without any other knowledge, $f(x)$ is uniformly distributed, and the second probability is saying that once it's known that $f(x) = y$ then $f(x')$ is uniformly distributed among the remaining $|A| - 1$ candidates. These probabilities are easy to see if you *lazily* define f as needed. Imagine starting with an empty second column for the definition table of f. Whenever $f(x)$ is needed the table is consulted. If $f(x)$ is defined its value is returned but if it is not yet defined a result is randomly selected from the remaining $|A| - 1$ elements.

Number of Functions. How many different functions exist with this signature? This can be computed using the table-definition approach. Once the first column of the table is filled, each different second column represents a different function. So, counting how many different functions exist is equivalent to counting how many different second-columns exist. There are $|B|$ candidates for the element in the first row, $|B|$ candidates for the element in the second row, etc. Since there are $|A|$ rows, there are $|B|^{|A|}$ different second columns and hence $|B|^{|A|}$ different functions $f : A \to B$.

How many permutation functions $f : A \to A$ exist? Again we can use the table-definition of f as a counting aid. The first entry of the second column can have any of the $|A|$ codomain elements assigned to it. But, once that entry is filled, the next entry cannot use the same value, which means there are $|A| - 1$ candidates for the second entry. The third entry has $|A| - 2$ candidates and so on. When only one entry remains to be filled there is only one candidate left. There are thus $(|A|) \cdot (|A| - 1) \cdot (|A| - 2) \cdot (|A| - 3) \cdots (1) = |A|!$ different second columns and permutation functions. Another way of counting is to simply recognize that in a permutation function the second column is a permutation of the first and includes every element of A. It is well known that there are $|A|!$ permutations of a sequence of length $|A|$.

Example 5. How many C functions take an `int` as input and return an `int` as output (and have no other side effects such as changing other program state)? This is asking how many functions exist with the signature $\mathsf{Int} \to \mathsf{Int}$ where Int is the set of all C `int`s. If we assume there are 2^{32} different `int`s, which is usually the case since `int`s are usually represented as 32 bits, then the number of functions with this signature is $|\mathsf{Int}|^{|\mathsf{Int}|} = (2^{32})^{(2^{32})} = 2^{2^{37}}$.

How many such functions are permutation functions? If we were filling in a table to represent the C function's input/output behavior, the first entry would have 2^{32} possible candidates, the second entry would have $2^{32} - 1$ possible candidates, etc, until the last entry has but one candidate. There are therefore $2^{32}!$ ways to define the

permutation function. □

Mathematical Permutation Functions. Mathematics gives us several methods for devising permutation functions. Several of which have been used historically to encrypt text.

Shift Cipher. Addition is invertible. $f(x) = x + k$ has inverse $f^{-1}(y) = y + -k$. This was used by Caesar, with $k = 3$, to send encrypted messages. If we let A=0, B=1, ..., Z=25 and perform our operations mod 26 then we have exactly the permutation $\mathbb{Z}_{26} \to \mathbb{Z}_{26}$ that Caesar used.

x	$f(x)$
0	3
1	4
2	5
...	...
22	25
23	0
24	1
25	2

Multiplicative Cipher. Multiplication is invertible. $f(x) = x \cdot k$ has inverse $f^{-1}(y) = y \cdot k^{-1}$. This works because $f^{-1}(f(x)) = f^{-1}(x \cdot k) = (x \cdot k) \cdot k^{-1} = x \cdot (k \cdot k^{-1}) = x \cdot 1 = x$, but only works if $k \cdot k^{-1} = 1$. If we want to stick to integers so that we can represent text characters as above and avoid using a fraction for k^{-1} we can instead do our multiplications mod an integer c. The integer value k is guaranteed to have an integer multiplicative inverse k^{-1} mod c if and only if $\gcd(k, c) = 1$. For example $\gcd(2, 5) = 1$, so there is a $2 \cdot 2^{-1} \bmod 5 = 1$. This means that $f(x) = x \cdot 2 \bmod 5$ is an invertible function $\mathbb{Z}_5 \to \mathbb{Z}_5$.

x	$f(x)$
0	0
1	2
2	4
3	1
4	3

Affine Cipher. If you compose two permutation functions with identical domains, the result is a permutation function with the same domain. This is used as a way to improve the quality of a permutation function by making it more complex. Let $f(x) = x \cdot k_1 \bmod c$ be a multiplicative cipher and $g(x) = x + k_2 \bmod c$ be a shift cipher (with $\gcd(k_1, c) = 1$). Each is a permutation function with signature $\mathbb{Z}_c \to \mathbb{Z}_c$. Their composition $g(f(x)) = x \cdot k_1 + k_2 \bmod c$ is also a permutation function with signature $\mathbb{Z}_c \to \mathbb{Z}_c$.

RSA. The RSA function is used for public-key cryptography. Without explaining it right now, here's an example of an RSA permutation with signature $\mathbb{Z}_{15} \to \mathbb{Z}_{15}$: $f(x) = x^3 \bmod 15$.

Computationally Efficient Permutation Functions: AXR. Mathematical functions can be expensive when implemented – multiplication and division of large numbers takes a lot of CPU cycles – so most cryptography uses computationally cheap invertible operations instead. If variable `x` is declared as a 32-bit `unsigned int` then there are several operations that can be applied to `x` that are invertible.

```
x = x + k;       // Because x is 32 bits long, computes (x = x + k) mod 2^32 automatically
x = x - k;       // Restores x back to it's original value
x = x ^ k;       // Bitwise xor of x and k
x = x ^ k;       // xor is its own inverse: 0^1^1=0, 0^0^0=0, 1^1^1=1, 1^0^0=1
x = ROTL(x, k);  // Rotate the bits in x left k positions
x = ROTR(x, k);  // Restores x back to it's original value
```

Because these operations are each invertible and have signature $\{0,1\}^{32} \to \{0,1\}^{32}$, they can be composed and the result is an invertible function with signature $\{0,1\}^{32} \to \{0,1\}^{32}$. These three operations – addition, XOR, rotation – are so commonly used in cryptography that entire classes of permutation functions are designed using just them and are called AXR ciphers. Some other simple operations on 32-bit values – such as other bitwise operations or bit-shift – are also efficient, but not invertible.

Example. Write an invertible C function that takes an `unsigned int` as a parameter and returns an `unsigned int` as its result. Use each of the AXR operations twice.

```
unsigned int permutation(unsigned int x) {
    x = x + 0x82FA6C79;
    x = (x << 17) | (x >> 15);  // How to express x = ROTL(x,17) in C;
    x = x ^ 0x29AE69D0;
    x = x + 0x6A3E9B37;
    x = (x << 3) | (x >> 29);  // How to express x = ROTL(x,3) in C;
    x = x ^ 0x802BFE62;
    return x;
}
```
□

Computationally Efficient Permutation Functions: Feistel. Another way to get an invertible function is with a Feistel structure. In such a structure, the input is split in half, each half is then used in conjunction with a mixing function to obscure the other half. For example, let $f : \{0,1\}^{16} \to \{0,1\}^{16}$ be any function; it doesn't have to be invertible when used in a Feistel structure. Then the following is invertible.

```
unsigned int permutation(unsigned int x) {
    let a be the first 16 bits of x
    let b be the second 16 bits of x
    a = a xor f(b)
    b = b xor f(a)
    x = concat(a,b)
    return x;
}
```

Sample Problems

These problems all come from prior quizzes or exams. They therefore represent some of the types of problems you should be able to complete after reading and understanding this document.

1) Let's say that you know an affine cipher is being used with the integer ring $\mathbb{Z}_{10}$, and you know plaintext integers 9 and 2 map to ciphertext integers 2 and 1 (respectively). What key is in use? *Hint: This is two equations and two unknowns but with a mod attached. The mod doesn't effect the techniques you learned in algebra, so do those but keep the mods at the end of each equation.*

2) For $\mathbb{Z}_{15}$ list all the values that have multiplicative inverses and the multiplicative inverse of each.

3) Let's say that $p : \{0,1\}^b \to \{0,1\}^b$ is a random invertible function and $f : \{0,1\}^b \to \{0,1\}^b$ is a random function, and that x and y are different b-bit strings. Evaluate the following probabilities: (i) $\Pr[f(x) = f(y)]$, (ii) $\Pr[p(x) = p(y)]$, (iii) $\Pr[f(0) = x \wedge f(1) = y]$, (iv) $\Pr[p(0) = x \wedge p(1) = y]$. *Note:* $\{0,1\}^b$ *is the set of all* b*-bit strings.*

4) For each of the following, write a C statement that if executed after the given one would return `x` to its previous value. If not possible, give two `x` inputs that map to the same `x` output. In each, `x` is an `unsigned int`. (a) `x = x + 2;` (b) `x = x >> 2;` (c) `x = x ^ 2;` (d) `x = x | 2;` (bitwise "or")

5) How many functions are there that map one byte to one byte? Explain your answer in one short sentence. *How big is the domain and range? How many different bytes are there?*

6) AES is a permutation function from 16 bytes to 16 bytes. How many permutation functions are there that map 16 bytes to 16 bytes? Explain your answer in one short sentence. *How big is the domain and range? How many different 16 bytes strings are there?*

7) In one sentence explain what is the difference between an invertible function and an invertible permutation function?

8) Write an invertible function on eight bytes using a Feistel construction that takes two `unsigned ints` as an array and uses `((x << 1) + x)` as its mixing function. The high half of the eight bytes is the first element of the array and the low half is the second. The result of the function should be put back into the array. Here's the function header:

```
void permute(unsigned blk[2]) {
```

9) For the following, write a C statement that when executed after the given one returns `x` to its previous value. If not possible, give the two simplest `x` input values mapping to the same `x` output. In each, `x` is an `unsigned int`. (a) `x = x >>> 2;` (right rotation) (b) `x = x / 2;` (c) `x = x & 2;` (bitwise "and")

10) Problem 5 on this quiz has you write a function that maps an input of eight bytes to an output of eight bytes. Mathematically speaking, how many functions are there that map this size input to this size output? Explain your answer in one short sentence.

11) How many of the functions that map an input of eight bytes to an output of eight bytes are permutation functions? Explain your answer in one short sentence.

12) Let $f(x) = 3x \bmod 7$ be a function $f : \mathbb{Z}_7 \to \mathbb{Z}_7$. What is the inverse of this function? Either depict the inverse as a function diagram (ie, two-ovals with arrows depicting the mapping), or mathematically (ie, $f^{-1}(y) = \ldots y \ldots$).

13) Let's say you have a function $f : A \to B$ what properties are required of f for it to be invertible? If f is invertible, what does this imply about A and B?

14) Let's say you have a function $f : A \to B$ what properties are required of f for it to be a permutation? If f is a permutation, what does this imply about A and B?

15) Below is a permutation function written in C. It takes eight bytes as input – represented as an array of two `unsigned ints` – and puts the result back into the same array. Write its inverse using the below function header.

```
void permute(unsigned blk[2]) {
    blk[0] = blk[0] ^ ((blk[1] << 1) + blk[1]);
    blk[1] = blk[1] ^ ((blk[0] << 1) + blk[0]);
}
```

```
void inverse_permute(unsigned blk[2]) {
```

16) Which numbers in $\mathbb{Z}_{30}$ have a multiplicative inverse? Show your work.

17) What is 7's multiplicative inverse in $\mathbb{Z}_{30}$? Document your search by showing everything you try.

18) Recall that the affine cipher (when using letters A=0..Z=25) is defined $e_{a,b}(x) = ax + b \bmod 26$. If the ciphertext "TUD" was created using key $(a, b) = (3, 20)$ what was the plaintext? Show your work.

19) Which numbers in $\mathbb{Z}_{24}$ have a multiplicative inverse? What is $4/7 \bmod 24$? Show your work.

20) You intercept a large ciphertext created using a shift cipher on English plaintext and discover B is the most common letter in the ciphertext. What can you assume is the shift key? The word WAWC is found in the ciphertext. Is your assumed shift key correct? Explain.