PRFs from PRGs

Boaz Barak

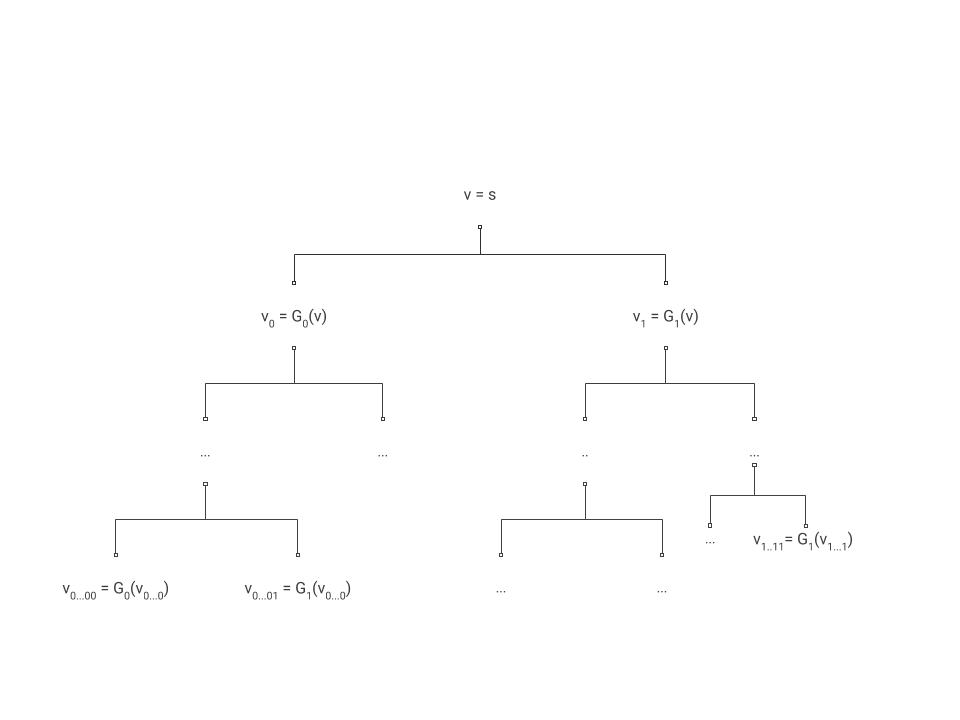
**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Pseudorandom functions from pseudorandom generators

We have seen that PRF’s (pseudorandom functions) are extremely useful, and we’ll see some more applications of them later on. But are they perhaps too amazing to exist? Why would someone imagine that such a wonderful object is feasible? The answer is the following theorem:

# 

Suppose that the PRG Conjecture is true, then there exists a secure PRF collection such that for every , maps to .



The construction of a pseudorandom function from a pseudorandom generator can be illustrated by a depth binary tree. The root is labeled by the seed and for every internal node labeled by a strong , we use that label as a seed into the PRG to label ’s two children. In particular, the children of are labeled with and respectively. The output of the function on input is the label of the leaf counting from left to right. Note that the numbering of leaf is related to the bitstring representation of and the path leaf in the following way: we traverse to leaf from the root by reading off the bits of left to right and descend into the left child of the current node for every 0 we encounter and traverse right for every 1.

# 

If the PRG Conjecture is true then in particular by the length extension theorem there exists a PRG that maps bits into bits. Let’s denote where denotes concatenation. That is, denotes the first bits and denotes the last bits of .

For , we define as

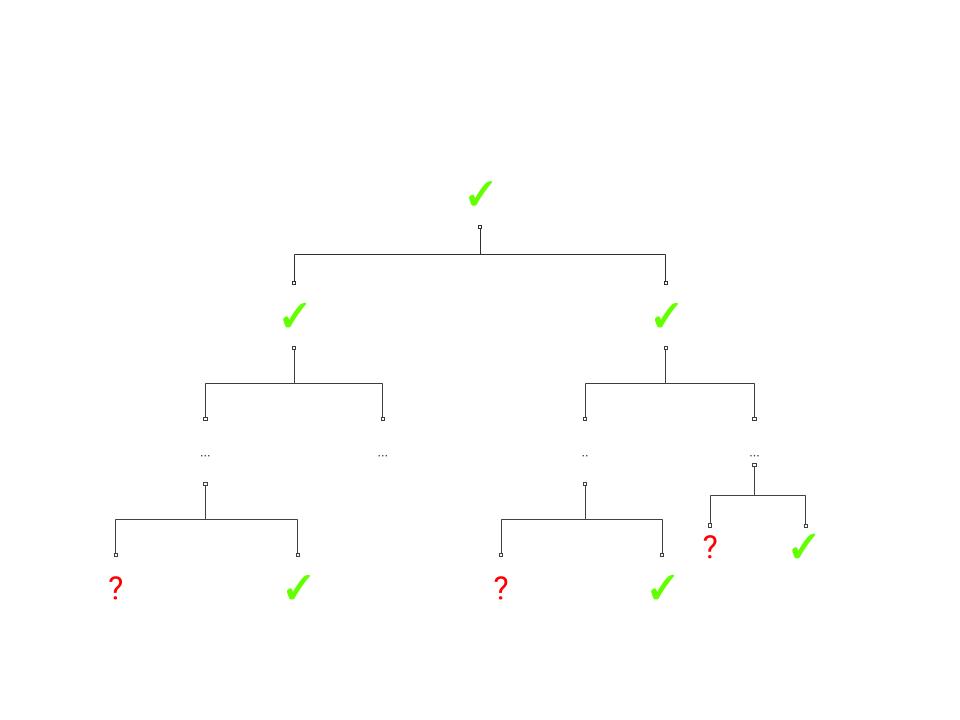
This corresponds to composed applications of for . If the bit of ’s binary string is 0 then the application of the PRG is otherwise it is . This series of successive applications starts with the initial seed .

This definition directly corresponds to the depiction in PRFfromPRGfig, where the successive applications of correspond to the recursive labeling procedure.

By the definition above we can see that to evaluate we need to evaluate the pseudorandom generator times on inputs of length , and so if the pseudorandom generator is efficiently computable then so is the pseudorandom function. Thus, “all” that’s left is to prove that the construction is secure and this is the heart of this proof.

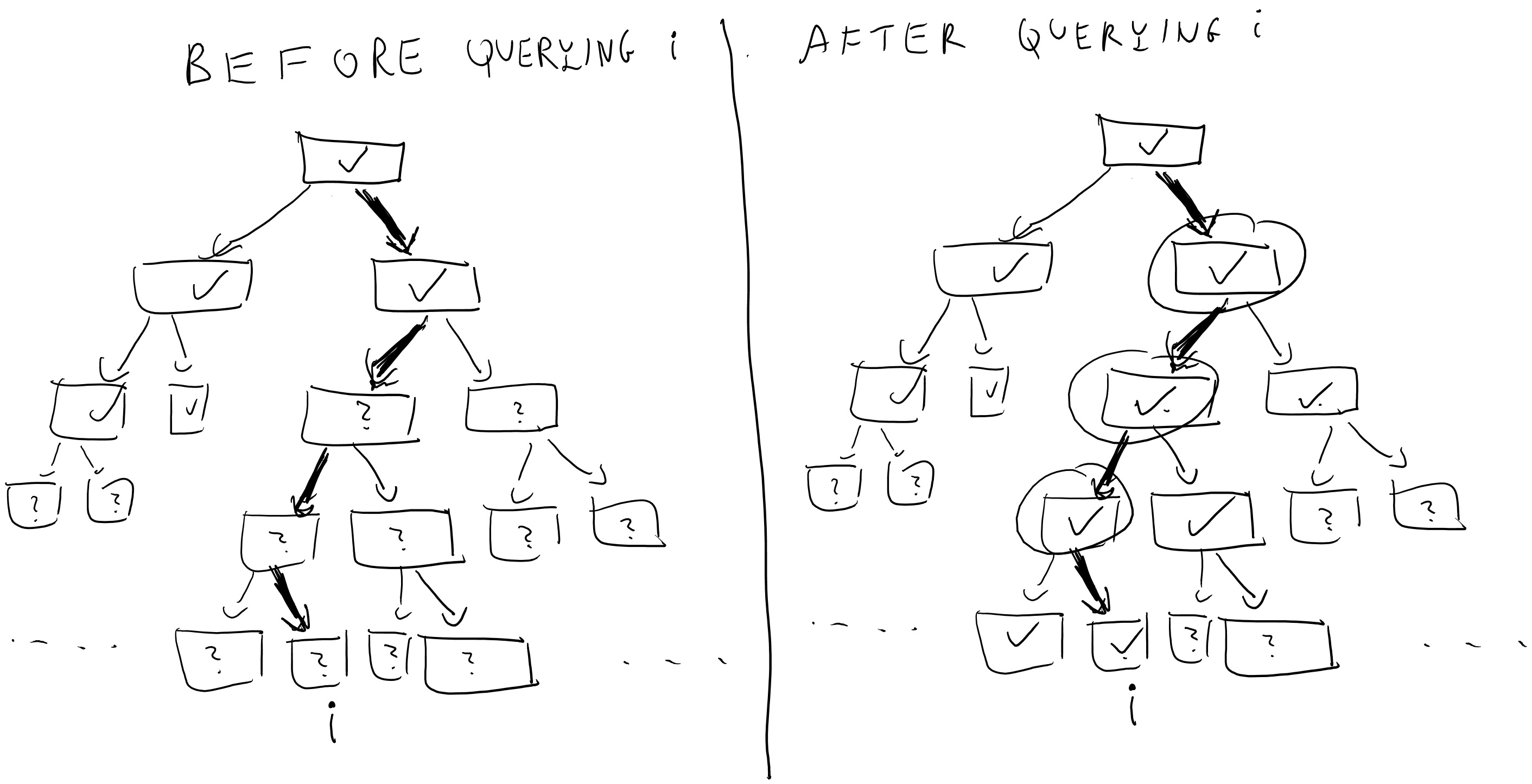
I’ve mentioned before that the first step of writing a proof is convincing yourself that the statement is true, but there is actually an often more important zeroth step which is understanding what the statement actually *means*. In this case what we need to prove is the following:

Given an adversary that can distinguish in time a black box for from a black-box for a random function with advantage , we need to come up with an adversary that can distinguish in time an input of the form (where is random in ) from an input of the form where is random in with bias at least .



In the “lazy evaluation” implementation of the black box to the adversary, we label every node in the tree only when we need it. Subsequent traversals do not reevaluate the PRG, leading to reuse of the intermediate seeds. Thus for example, two sibling leaves will correspond to a single call to , where is their parent’s label, but with the left child receiving the first bits and the right child receiving the second bits of . In this figure check marks correspond to nodes that have been labeled and question marks to nodes that are still unlabeled.

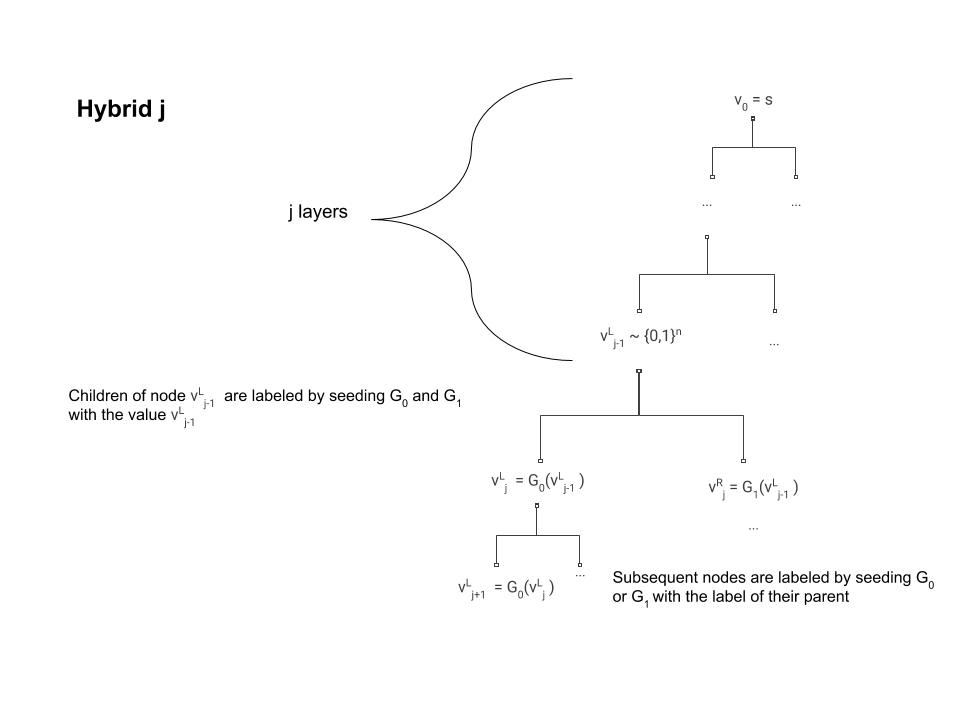
Let us consider the “lazy evaluation” implementation of the black box for illustrated in lazyevalprffig. That is, at every point in time there are nodes in the full binary tree that are labeled and nodes which we haven’t yet labeled. When makes a query , this query corresponds to the path in the tree. We look at the lowest (furthest away from the root) node on this path which has been labeled by some value , and then we continue labelling the path from downwards until we reach . In other words, we label the two children of by and , and then if the path involves the first child then we label its children by and , and so on and so forth (see oracleevaltreefig). Note that because and correspond to a single call to , regardless of whether the traversals continues left or right (i.e. whether the current level corresponds to a value 0 or 1 in ) we label both children at the same time.



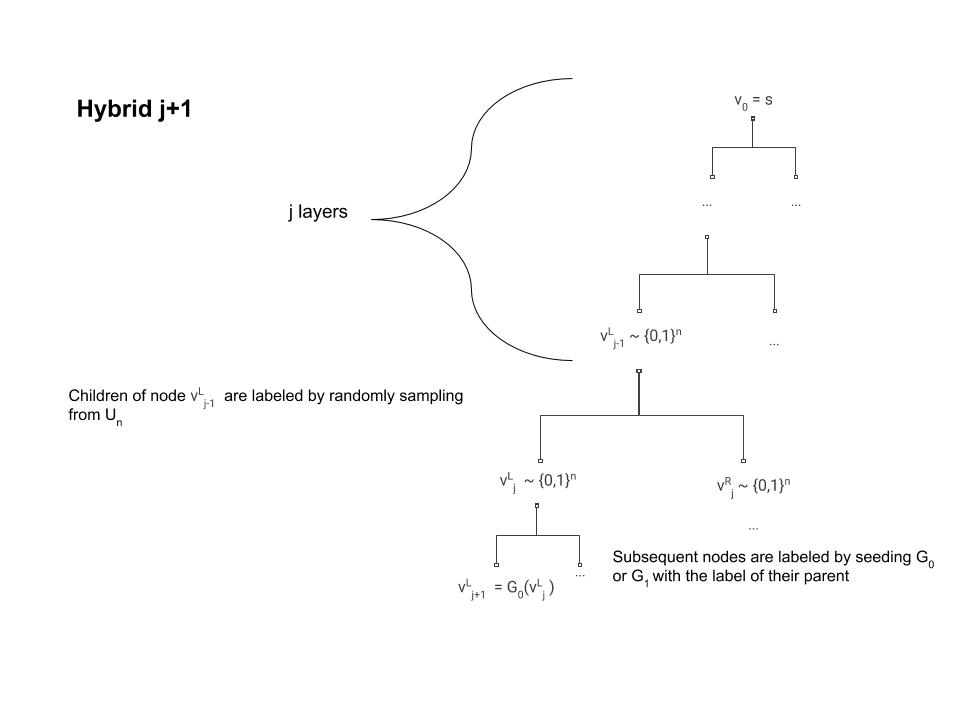
When the adversary queries , the oracle takes the path from to the root and computes the generator on the minimum number of internal nodes that is needed to obtain the label of the leaf.

A moment’s thought shows that this is just another (arguably cumbersome) way to describe the oracle that simply computes the map . And so the experiment of running with this oracle produces precisely the same result as running with access to . Note that since has running time at most , the number of times our oracle will need to label an internal node is at most (since we label at most nodes for every query ).

We now define the following hybrids: in the hybrid, we run this experiment but in the first times the oracle needs to label internal nodes, then instead of labelling the child of by (where is the label of ), the oracle simply labels it by a random string in .

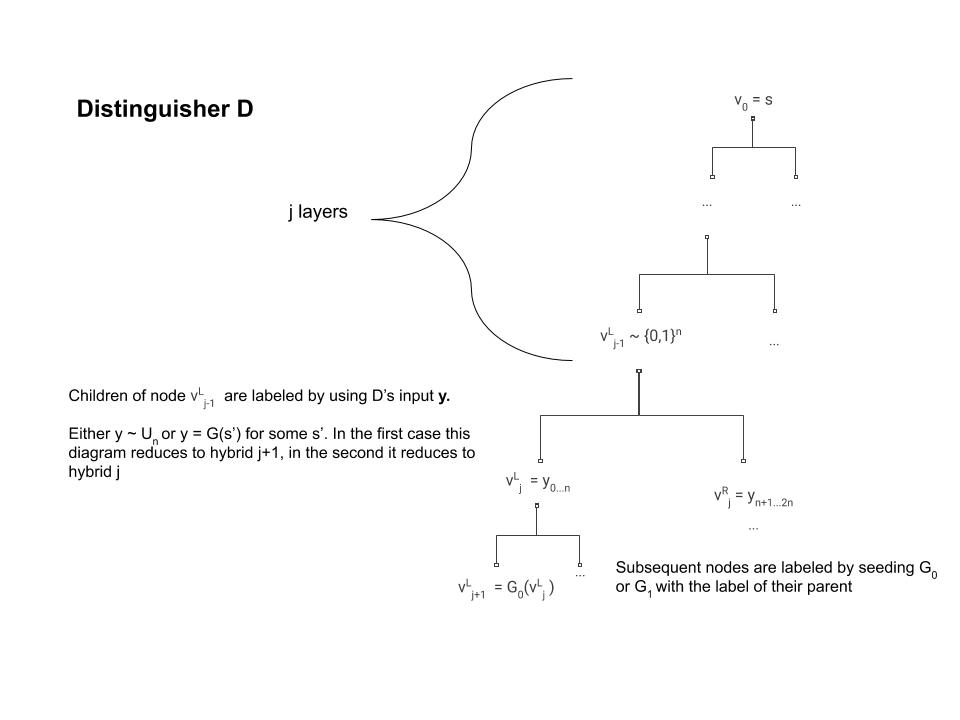


In the hybrid all lables up to the first layers of the tree are drawn uniformly at random from . All subsequent children’s labels are produced in the usual way by seeding with the label of the parent and assigning the first bits () to the left child and the last bits () to the right child. For example, for some node at the level, we generate pseudorandom string and label the left child and the right child . Note that the labeling scheme for this diagram is different from that in the previous figures. This is simply for easy of exposition, we could still index our nodes via the path reaching them from the root.



The hybrid differs from the in that the process of assigning random labels continues until the layer of the tree as opposed to the . The hybrids are otherwise completely identically constructed.

Note that the hybrid corresponds to the case where the oracle implements the function , while in the hybrid all labels are random and hence implements a random function. By the hybrid argument, if can distinguish between the hybrid and the hybrid with bias then there must exists some such that it distinguishes between the hybrid (pictured in hybridj) and the hybrid (pictured in hybridj1) with bias at least . We will use this and to break the pseudorandom generator.



Distinguisher D is similar to hybrid , in that the nodes in the first layers are assigned completely random labels. When evaluating along a particular path through $v\_{j-1}^{L}, rather than labeling the two children by applying to its label, it simply splits the input into two strings ,. If is truly random, is identical to hybrid . If for some random seed , then simulates hybrid .

We can now describe our distinguisher (see distinguisherd) for the pseudorandom generator. On input a string will run and the oracle inside its belly with one difference- when the time comes to label the node, instead of doing this by applying the pseudorandom generator to the label of its parent (which is what should happen in the oracle) it uses its input to label the two children of .

Now, if was completely random then we get exactly the distribution of the oracle, and hence in this case simulates internally the hybrid. However, if for some randomly sampled , though it may not be obvious at first, we actually get the distribution of the oracle.

The equivalence between hybrid and distinguisher under the condition that is non obvious, because in hybrid , the label for the children of was supposed to be the result of applying the pseudorandom generator to the label of and not to some other random string (see distinguisherd). However, because was labeled *before* the step then we know that it was actually labeled by a random string. Moreover, since we use lazy evaluation we know that step is the *first* time where we actually use the value of the label of . Hence, if at this point we *resampled* this label and used a completely independent random string then the distribution of and would be *identical*. Hence the distribution of , for drawn from , is identical to the distribution, , of the hybrid, and thus if had advantage in breaking the PRF then will have advantage in breaking the PRG thus obtaining a contradiction.

This proof is ultimately not very hard but is rather confusing. I urge you to also look at the proof of this theorem as is written in Section 7.5 (pages 265-269) of the KL book.

# 

While this construction reassures us that we can rely on the existence of pseudorandom functions even on days where we remember to take our meds, this is not the construction people use when they need a PRF in practice because it is still somewhat inefficient, making calls to the underlying pseudorandom generators. There are constructions (e.g., HMAC) based on hash functions that require stronger assumptions but can use as few as two calls to the underlying function. We will cover these constructions when we talk about hash functions and the random oracle model. One can also obtain practical constructions of PRFs from *block ciphers*, which we’ll see later in this lecture.

## Securely encrypting many messages - chosen plaintext security

Let’s get back to our favorite task of *encryption*. We seemed to have nailed down the definition of secure encryption, or did we?

# 

Try to think what kind of security guarantees are *not* provided by the notion of computational secrecy we saw in compsecdef

Our current definition requires talks about encrypting a *single* message, but this is not how we use encryption in the real world. Typically, Alice and Bob (or Amazon and Boaz) setup a shared key and then engage in many back and forth messages between one another. At first, we might think that this issue of a single long message vs. many short ones is merely a technicality. After all, if Alice wants to send a sequence of messages to Bob, she can simply treat them as a single long message. Moreover, the way that *stream ciphers* work, Alice can compute the encryption for the first few bits of the message she decides what will be the next bits and so she can send the encryption of to Bob and later the encryption of . There is some truth to this sentiment, but there are issues with using stream ciphers for multiple messages. For Alice and Bob to encrypt messages in this way, they must maintain a *synchronized shared state*. If the message was dropped by the network, then Bob would not be able to decrypt correctly the encryption of .

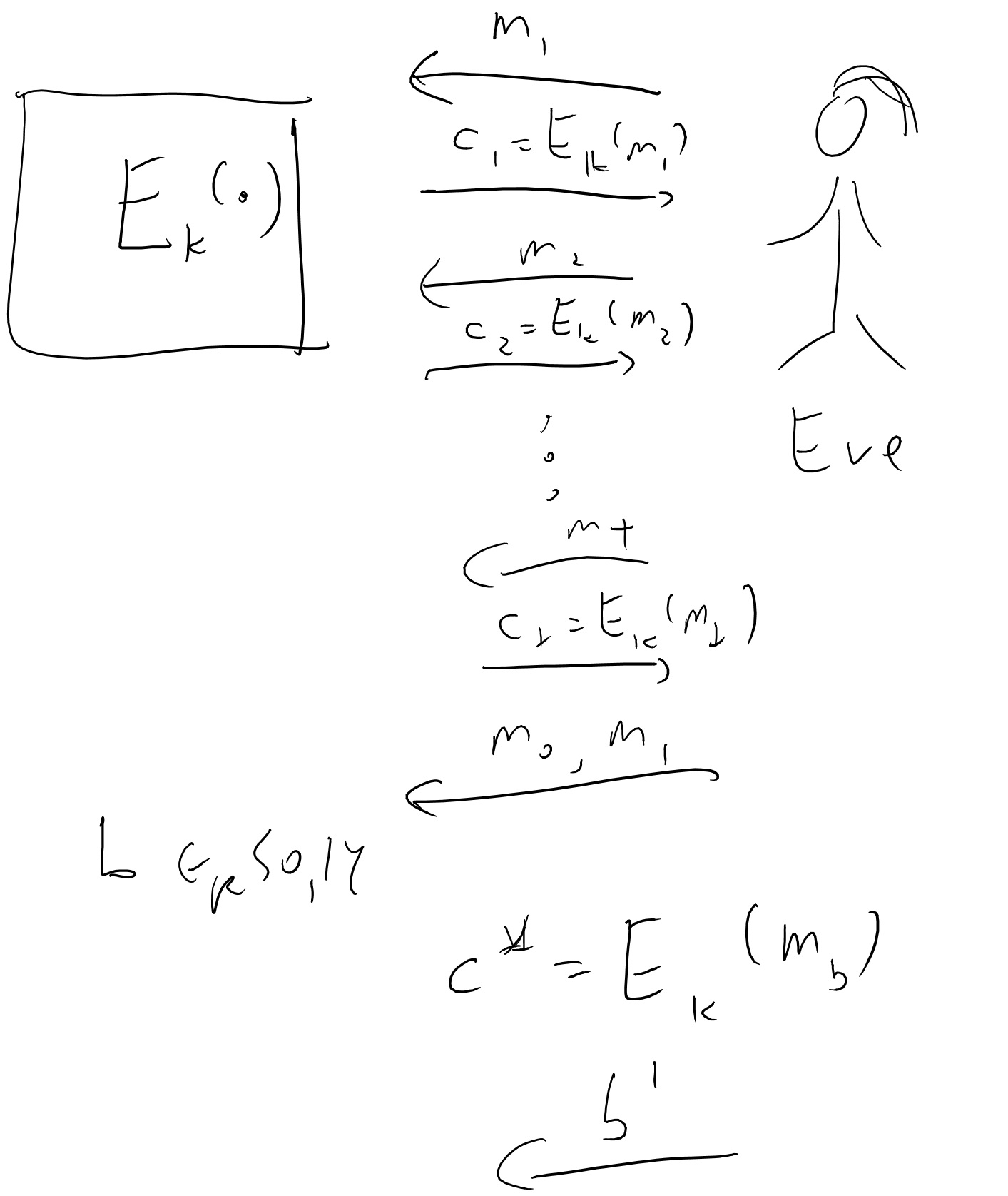
There is another way in which treating many messages as a single tuple is unsatisfactory. In real life, Eve might be able to have some impact on *what* messages Alice encrypts. For example, the Katz-Lindell book describes several instances in World War II where Allied forces made particular military maneuver for the sole purpose of causing the axis forces to send encryptions of messages of the Allies’ choosing. To consider a more modern example, today Google uses encryption for all of its search traffic including (for the most part) the *ads* that are displayed on the page. But this means that an attacker, by paying Google, can cause it to encrypt arbitrary text of their choosing. This kind of attack, where Eve *chooses* the message she wants to be encrypted is called a *chosen plaintext attack*. You might think that we are already covering this with our current definition that requires security *for every* pair of messages and so in particular this pair could chosen by Eve. However, in the case of multiple messages, we would want to allow Eve to be able to choose *after* she saw the encryption of .

All that leads us to the following definition, which is a strengthening of our definition of computational security:

# 

An encryption scheme is *secure against chosen plaintext attack (CPA secure)* if for every polynomial time , Eve wins with probability at most in the game defined below:

1. The key is chosen at random in and fixed.
2. Eve gets the length of the key as input.[[1]](#footnote-40)
3. Eve interacts with for rounds as follows: in the round, Eve chooses a message and obtains .
4. Then Eve chooses two messages , and gets for .
5. Eve *wins* if she outputs .



In the CPA game, Eve interacts with the encryption oracle and at the end chooses , gets an encryption and outputs . She *wins* if

cpasecuredef is illustrated in cpasecgamefig. Our previous notion of computational secrecy (i.e., compsecdef) corresponds to the case that we skip Step 3 above. Since Step 3 only gives the adversary more power (and hence is only more likely to win), CPA security (cpasecuredef) is *stronger* than computational secrecy (compsecdef), in the sense that every CPA secure encryption is also computationally secure. It turns out that CPA security is *strictly stronger*, in the sense that without modification, our stream ciphers cannot be CPA secure. In fact, we have a stronger, and intially somewhat surprising theorem:

# 

There is no CPA secure where is *deterministic*.

# 

The proof is very simple: Eve will only use a single round of interacting with where she will ask for the encryption of . In the second round, Eve will choose and , and get she wil then output if and only if .



Insecurity of deterministic encryption

This proof is so simple that you might think it shows a problem with the definition, but it is actually a real problem with security. If you encrypt many messages and some of them repeat themselves, it is possible to get significant information by seeing the repetition pattern (que the XKCD cartoon again, see xkcdnavajotwofig). To avoid this issue we need to use a *randomized* (or *probabilistic*) encryption, such that if we encrypt the same message twice we *won’t* see two copies of the same ciphertext.[[2]](#footnote-47) But how do we do that? Here pseudorandom functions come to the rescue:

# 

Suppose that is a PRF collection where , then the following is a CPA secure encryption scheme: and .

# 

I leave to you to verify that . We need to show the CPA security property. As is usual in PRF-based constructions, we first show that this scheme will be secure if was an actually random function, and then use that to derive security.

Consider the game above when played with a completely random function and let be the random string chosen by in the round and the string chosen in the last round. We start with the following simple but crucial claim:

**Claim:** The probability that for some is at most .

**Proof of claim:** For any particular , since is chosen independently of , the probability that is . Hence the claim follows from the union bound. QED

Given this claim we know that with probability (which is ), the string is distinct from any string that was chosen before. This means that by the lazy evaluation principle, if is a completely random function then the value can be thought of as being chosen at random in the final round independently of anything that happened before. But then amounts to simply using the one-time pad to encrypt . That is, the distributions and (where we think of as fixed and the randomness comes from the choice of the random function ) are both equal to the uniform distribution over and hence Eve gets absolutely no information about .

This shows that if was a random function then Eve would win the game with probability at most . Now if we have some efficient Eve that wins the game with probability at least then we can build an adversary for the PRF that will run this entire game with black box access to and will output if and only if Eve wins. By the argument above, there would be a difference of at least in the probability it outputs when is random vs when it is pseudorandom, hence contradicting the security property of the PRF.

## Pseudorandom permutations / block ciphers

Now that we have pseudorandom functions, we might get greedy and want such functions with even more magical properties. This is where the notion of *pseudorandom permutations* comes in.

# 

Let be some function that is polynomially bounded (i.e., there are some such that for every ). A collection of functions where for is called a *pseudorandom permutation (PRP) collection* if:

1. It is a pseudorandom function collection (i.e., the map is efficiently computable and there is no efficient distinguisher between with a random and a random function).
2. Every function is a permutation of (i.e., a one to one and onto map).
3. There is an efficient algorithm that on input returns .

The parameter is known as the *key length* of the pseudorandom permutation collection and the parameter is known as the *input length* or *block length*. Often, and so in most cases you can safely ignore this distinction.

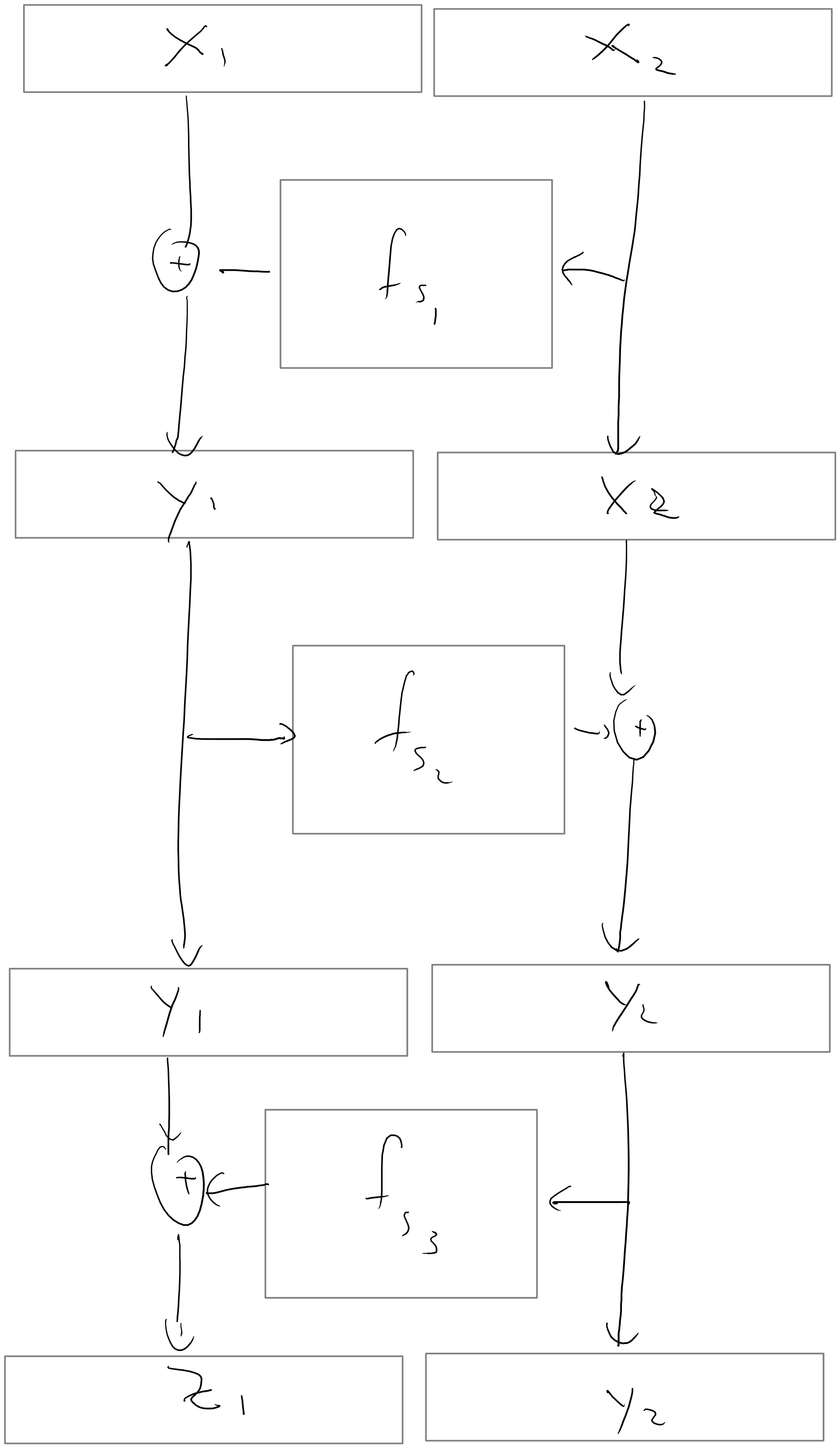
# 

At first look PRPdef might seem not to make sense, since on one hand it requires the map to be a permutation, but on the other hand it can be shown that with high probability a random map will *not* be a permutation. How can then such a collection be pseudorandom? The key insight is that while a random map might not be a permutation, it is not possible to distinguish with a polynomial number of queries between a black box that computes a random function and a black box that computes a random permutation. Understanding why this is the case, and why this means that PRPdef is reasonable, is crucial to getting intuition to this notion, and so I suggest you pause now and make sure you understand these points.

As usual with a new concept, we want to know whether it is possible to achieve and is useful. The former is established by the following theorem:

# 

If the PRF conjecture holds (and hence by prfthm also if the PRG conjecture holds) then there exists a pseudorandom permutation collection.



We build a PRP on bits from three PRFs on bits by letting where , and .

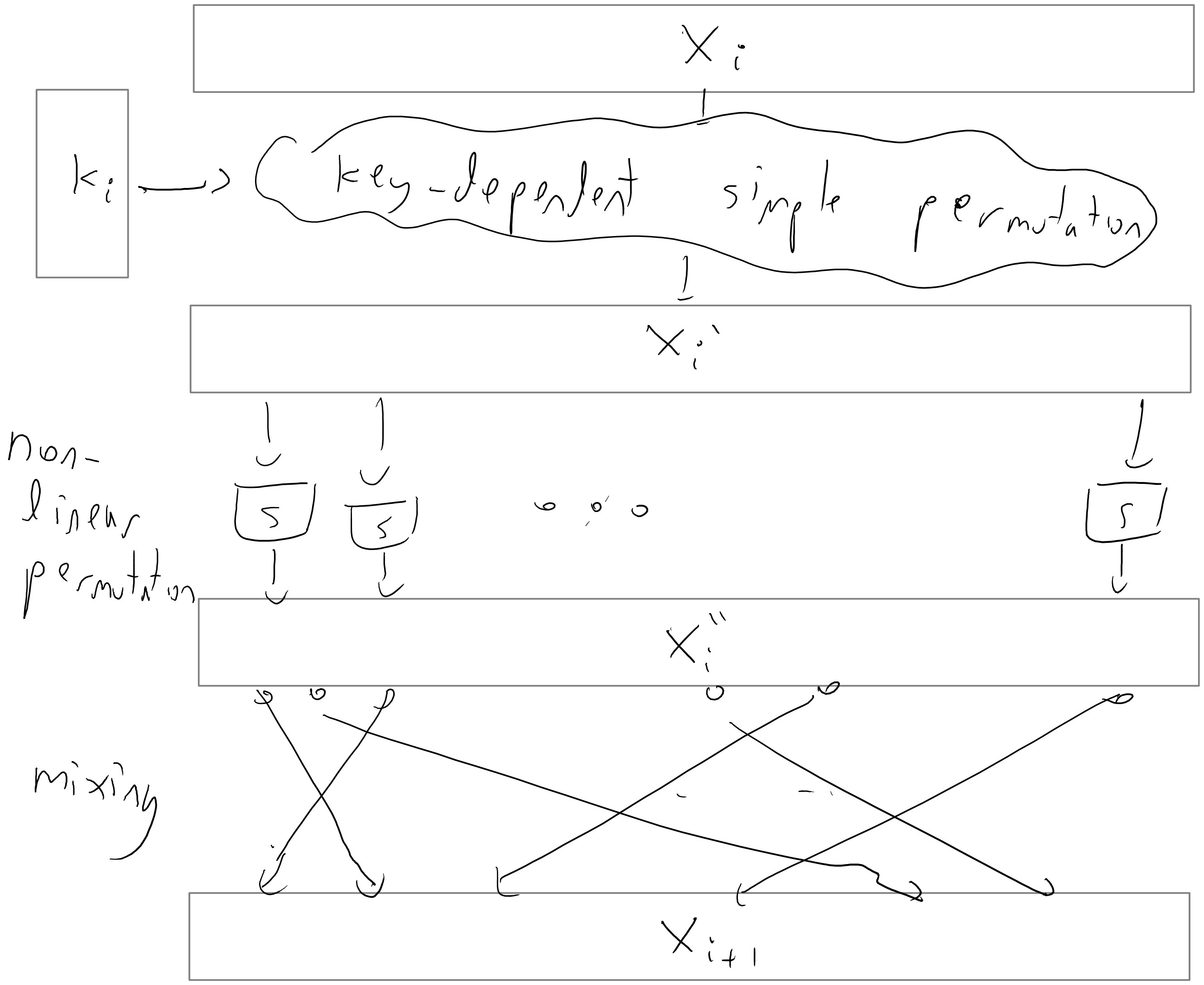
We will not show the proof of this theorem here, but feistelfig illustrates how the construction of a pseudorandom permutation from a pseudorandom function looks like. The construction (known as the Luby-Rackoff construction) uses several rounds of what is known as the [Feistel Transformation](https://en.wikipedia.org/wiki/Feistel_cipher) that maps a function into a permutation using the map . For an overview of the proof see Section 4.5 in Boneh Shoup or Section 7.6 in Katz-Lindell.

The more common name for a pseudorandom permutation is *block cipher* (though typically block ciphers are expected to meet additional security properties on top of being PRPs). The constructions for block ciphers used in practice don’t follow the construction of PRPfromPRF (though they use some of the ideas) but have a more ad-hoc nature.

One of the first modern block ciphers was the [Data Encryption Standard (DES)](https://goo.gl/XiCvjs) constructed by IBM in the 1970’s. It is a fairly good cipher- to this day, as far as we know, it provides a pretty good number of security bits compared to its key. The trouble is that its key is only bits long, which is no longer outside the reach of modern computing power. (It turns out that subtle variants of DES are far less secure and fall prey to a technique known as [differential cryptanalysis](https://goo.gl/GAvbh8); the IBM designers of DES were aware of this technique but kept it secret at the behest of the NSA.)

Between 1997 and 2001, the U.S. national institute of standards (NIST) ran a competition to replace DES which resulted in the adoption of the block cipher Rijndael as the new [advanced encryption standard (AES)](https://goo.gl/1HnqFb). It has a block size (i.e., input length) of 128 bits and a key size (i.e., seed length) of 128, 196, or 256 bits.

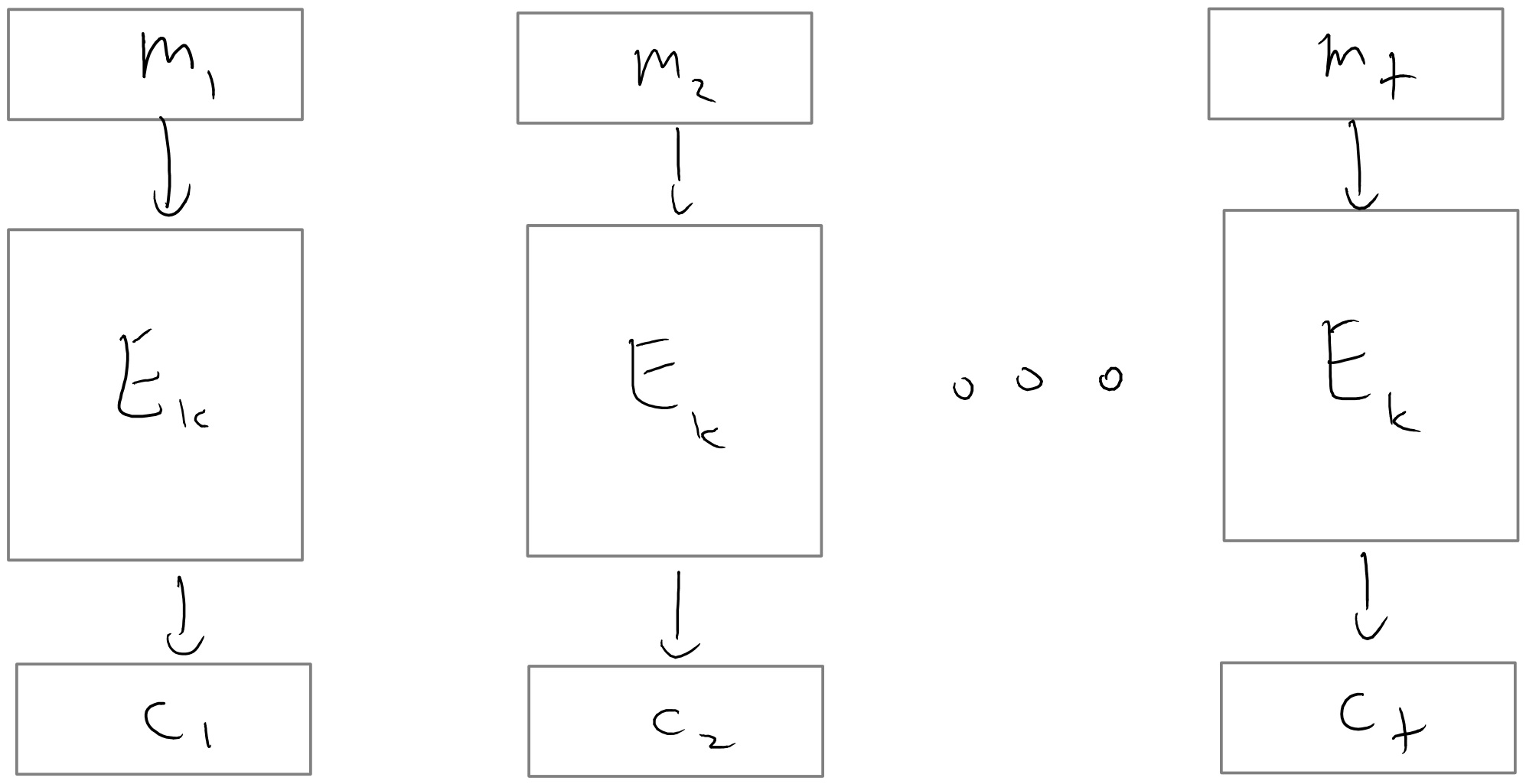
The actual construction of AES (or DES for that matter) is not extremely illuminating, but let us say a few words about the general principle behind many block ciphers. They are typically constructed by repeating one after the other a number of very simple permutations (see blockcipherfig). Each such iteration is called a *round*. If there are rounds, then the key is typically expanded into a longer string, which we think of as a tuple of strings via some pseudorandom generator known as the *key scheduling algorithm*. The -th string in the tuple is known as the *round key* and is used in the round. Each round is typically composed of several components: there is a “key mixing component” that performs some simple permutation based on the key (often as simply as XOR’ing the key), there is a “mixing component” that mixes the bits of the block so that bits that were initially nearby don’t stay close to one another, and then there is some non-linear component (often obtained by applying some simple non-linear functions known as “S boxes” to each small block of the input) that ensures that the overall cipher will not be an affine function. Each one of these operations is an easily reversible operations, and hence decrypting the cipher simply involves running the rounds backwards.



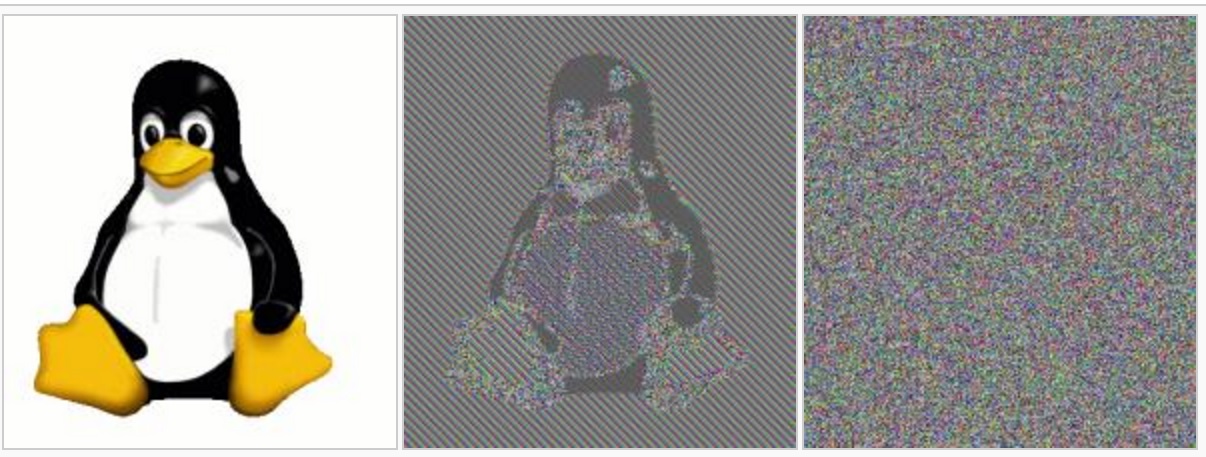
A typical round of a block cipher, is the round key, is the block before the round and is the block at the end of this round.

## Encryption modes

How do we use a block cipher to actually encrypt traffic? Well we could use it as a PRF in the construction above, but in practice people use other ways.[[3]](#footnote-64) The most natural approach would be that to encrypt a message , we simply use where is the PRP/block cipher. This is known as the *electronic code book (ECB) mode* of a block cipher (see ecbonefig). Note that we can easily decrypt since we can compute . However, this is a *deterministic* encryption and hence cannot be CPA secure. Moreover, this is actually a real problem of security on realistic inputs (see ecbtwofig).

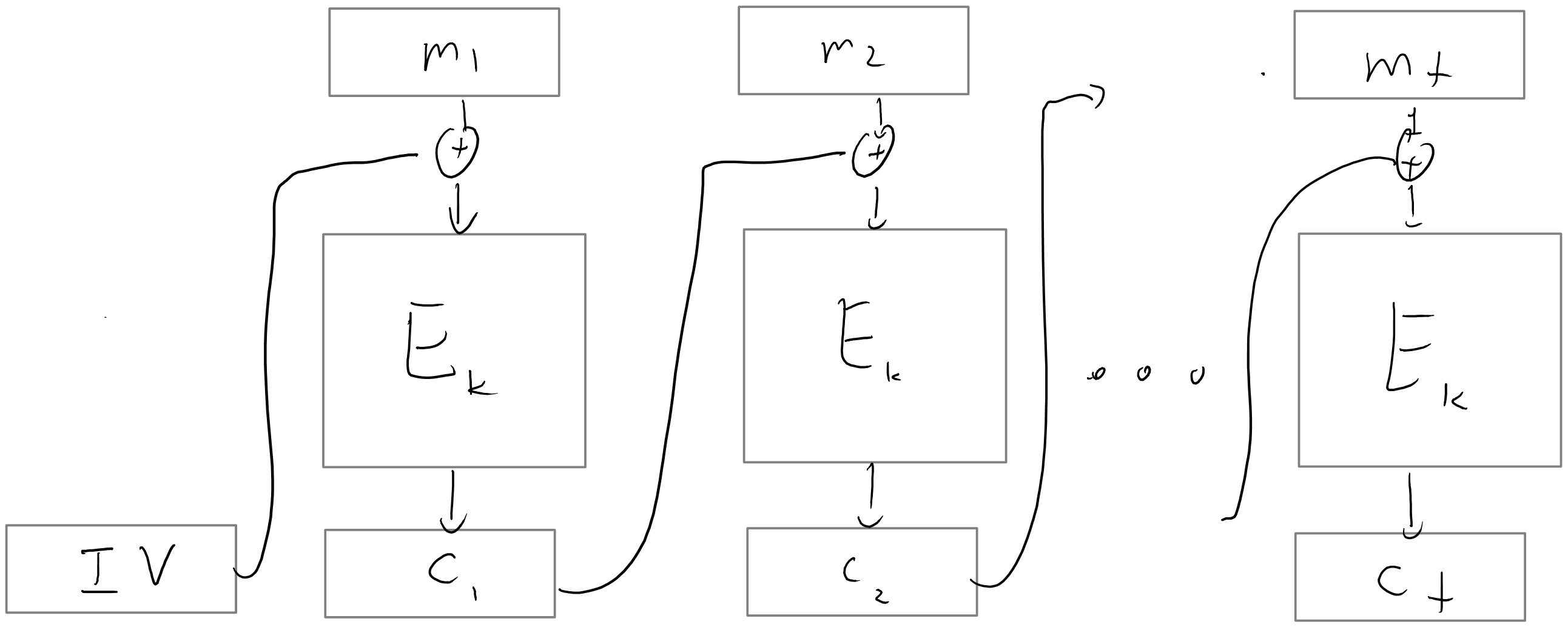


In the Electronic Codebook (ECB) mode every message is encrypted deterministically and independently



An encryption of the Linux penguin (left image) using ECB mode (middle image) vs CBC mode (right image). The ECB encryption is insecure as it reveals much structure about the original image. Image taken from Wikipedia.

A more secure way to use a block cipher to encrypt is the *cipher block chaining mode* where we XOR every message with the previous ciphertext (cbcmodefig). For the first message we XOR a string known as the *initialization vector* or IV. Note that if we lose a block to traffic in the CBC mode then we are unable to decrypt the next block, but can recover from that point onwards. It turns out that using this mode with a random IV can yield CPA security, though one has to be careful in how you go about it, see the exercises.



In the Cypher-Block-Chaining (CBC) the encryption of the previous message is XOR’ed into the current message prior to encrypting. The first message is XOR’ed with an *initialization vector* (IV) that if chosen randomly, ensures CPA security.

In the *output feedback mode (OFB)* we encrypt the all zero string using CBC mode to get a sequence of pseudorandom outputs that we can use as a stream cipher. Perhaps the simplest mode is *counter mode* where we convert a block cipher to a stream cipher by using the stream where is a random string in which we identify with (and perform addition modulo ) . For a modern block cipher this should be no less secure than CBC or OFB and has advantages that we can easily compute it in parallel.

A fairly comprehensive study of the different modes of block ciphers is in [this document by Rogaway](http://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf). His conclusion is that if we simply consider CPA security (as opposed to the stronger notions of *chosen ciphertext security* we’ll see in the next lecture) then counter mode is the best choice, but CBC, OFB and CFB are widely implemented due to legacy reasons. ECB should not be used (except as a building block as part of a construction achieving stronger security).

1. Giving Eve the key as a sequence of s as opposed to in binary representation is a common notational convention in cryptography. It makes no difference except that it makes the input length for Eve of length , which makes sense since we want to allow Eve to run in time. [↑](#footnote-ref-40)
2. If the messages are guaranteed to have *high entropy* which roughly means that the probability that a message repeats itself is negligible, then it is possible to have a secure deterministic private-key encryption, and this is sometimes used in practice. (Though often some sort of randomization or padding is added to ensure this property, hence in effect creating a randomized encryption.) Deterministic encryptions can sometimes be useful for applications such as efficient queries on encrypted databases. See [this lecture](https://goo.gl/GWJLFd) in Dan Boneh’s coursera course. [↑](#footnote-ref-47)
3. Partially this is because in the above construction we had to encode a plaintext of length with a ciphertext of length meaning an overhead of 100 percent in the communication. [↑](#footnote-ref-64)