CS 198 Codebreaking at Cal
Spring 2023 Course Notes

Note 2

Introduction to Randomness and Information Theory

As we saw with frequency analysis in the previous note, the study of probability is very important in cryptography. If our encryption scheme output is not random enough, attackers can often exploit this to recover the message and/or key! In this note, we will formalize a lot of the intuition behind probability through the lense of **information theory**.

XOR

The XOR operation, sometimes called the exclusive-OR, is a very powerful tool in cryptography. It's special properties lie in the fact that XOR-ing two binary strings does not lose any *information*.

For example "1010" $\oplus$ "1111":

Plaintext:	1010
Key:	1111
Ciphertext:	0101

This can be reversed by XOR-ing the ciphertext with the key again:

Cipher:	0101
Key:	1111
Plaintext:	1010

We can rewrite the equations as follows:

$$\begin{aligned} m \oplus k &= c \\ m \oplus k \oplus k &= c \oplus k \\ m &= c \oplus k \end{aligned}$$

As you can see, the plaintext is recovered in all cases. This property of XOR can be formalized as such:

$$\begin{aligned} &A \coloneqq \{0,1\} \\ &f : (A,A) \to A \\ &f(X,Y) \coloneqq X \oplus Y \\ &(\forall x,y \in A)(f(f(x,y),y) = x) \end{aligned}$$

That is to say, XOR-ing a string against a key, then XOR-ing that result against the key again will return the original string. Astute readers might notice this is equivalent to addition $\pmod 2$. Thus, a XOR cipher is equivalent to a substitution cipher with the alphabet defined as 0 and 1. The concepts from Note 1 thus apply, and we can be assured the message is recoverable in all cases.

Critically, however, the inverse only exists *if you know the key*. One could chain an arbitrary amount of XOR with different keys at different steps together, and losing the key to *any one* of those steps would result in the entire message being lost.

Information Theory

However, the phrase "lost message" begs the question, how do we quantify a message being lost? The answer is: a message is considered to be lost when the ciphertext's entropy is equal to the entropy of a random bitstring.

Definition 1: Entropy

Unlike in thermodynamics, the **entropy** of a given information source quantifies the amount of uncertainly in a random variable. While precise definitions are outside the scope of this class, the Shannon entropy (in bits) of a random variable x is given by:

$$H(x) = -\sum_i x_i \log_2(x_i)$$

where x_i represents the probability of event i occuring.

It follows from the definition of entropy that a random variable x with equal probability to be 0 or 1 (think coin flip) has 1 bit of entropy. A weighted coin would have less entropy, since we can be more certain of the outcome than of a truly random coin. This is why the entropy is higher – there is more uncertainty in the outcome.

Returning to the original question, one can see that a random bit is exactly like a coin flip, and thus has 2 equal outcomes. For a n-bit random message, there is n bits of entropy – and 2^n possible messages. Notably, the messages we analyzed in the previous note *do not* have n bits of entropy, as their key is repeated. The repeated key provides us with valuable information and allows for frequency analysis attacks. However, if the key was just as long as the message (and completely random), it is **mathematically impossible** to break the ciphertext! This holds for base 2 and other bases (one can simply convert back and forth between base 2 for proof). This type of encryption is called a one-time pad.

One-Time Pad

Definition 2: One-time Pad

A **one-time pad** is a substitution cipher utitlizing a perfectly random key at least as long as the message to be encrypted. Such a cipher is information-theoretically secure, also known as *perfect secrecy*.

Definition 3: Perfect Secrecy

A cipher that is **perfectly secret** is one in which the ciphertext provides zero information about the plaintext, such that the entropy of the ciphertext is equivalent to the entropy of a perfectly random string. An adversary with unlimited computational resources and time would not be able to crack it.

Equivalently, $P[M = m|C = c] = P[M = m]$ for all m, c, which states "the probability of the message being m given that the ciphertext is c is equal to the base probability of the message being m".

Let's consider what makes a one-time pad perfectly secret. Consider the very simple example of a single character long ciphertext – "1". Given this, could you determine whether the original bit was 0 or 1 (without knowing the key)? The nature of modular arithmetic makes it equally likely to have been 0 or 1. That is to say, the entropy of the ciphertext is equal to the entropy of a random bit – 1.

Extending this to an arbitrary number of bits does not change the security – since the key does not repeat and is truly random, no bit gains any information from the others. If you revealed 99% of the key, you would still have no more information about the last bit.

What is even more interesting about perfect secrecy and one-time pads is that a perfectly secure ciphertext can be "decrypted" into *anything* the user wants it to be. Consider the example ciphertext, which is the integer 40 converted to binary:

$$00101000$$

Say the intended message was 87. In that case, one can find the key by XOR-ing the ciphertext and the plaintext to receive: 127. We can then verify that this key is correct: $87 \oplus 127 = 40$.

Now, say the intended message was 12. We do the same thing and get a key of 36. Verify: $12 \oplus 36 = 40$.

Consider a more concrete example – a encrypted message from a general to his lieutenants was intercepted by the enemy. Using their powerful computers, they find two promising examples – "Attack at noon!" and "Attack at dawn!", with no way of discerning between the two (or the billions of other options).

This way, we can't even analyze the possible outcomes to look for "correct" sounding outcomes – because it could be *any combination of letters* with length n. This is why it is **impossible**, not

just hard, for an attacker to recover the message. It's because the intended message has the same probability of being "correct" as every other possible message.

This is not exclusive to XOR – it would work with a basic substitution cipher like Vignere covered in Note 1 - assuming the key was equally as long as the message and perfectly random. XOR is simply popular for use in computers due it its speed while preserving critical properties of modular arithmetic.

Formal Proof of Perfect Secrecy

Hopefully the intuition behind one-time pad's perfect secrecy is clear by now. Here, we will formally prove it by the definition given above for arbitrary M, C (plaintext, ciphertext):

$$P[M = m | C = c] = P[M = m]$$

Theorem 1: One Time Pad

The one-time pad is information-theoretically secure (perfectly secure).

First we must recall the definition of the **law of total probability**, which states that

$$P[X = x] = \sum_i P[X = x \cap Y = y_i] = \sum_i P[X = x | Y = y_i] \cdot P[Y = y_i]$$

for random variables X, Y such that

$$\sum_i P[Y = y_i] = 1$$

Next, we recall **Bayes' law**, which states that

$$P[A|B] = \frac{P[B|A] \cdot P[A]}{P[B]}$$

We want to find $P[M = m | C = c]$, which we can rewrite as

$$\frac{P[C = c | M = m] \cdot P[M = m]}{P[C = c]}$$

We must first find $P[C = c]$. Using the law of total probability, we see that

$$P[C = c] = \sum_i^{2^n} P[C = c | M = m_i] \cdot P[M = m_i]$$

for an n-bit message. Next, we realize that we can rewrite $P[C = c | M = m_i]$ as $K = m_i \oplus c$, by the definition of one-time pad. Note that $K \sim \text{Uniform}[0, 2^n)$ by construction.

$$P[C = c] = \sum_{i}^{2^n} P[K = m_i \oplus c] \cdot P[M = m_i]$$
$$P[C = c] = \sum_{i}^{2^n} 2^{-n} \cdot P[M = m_i]$$
$$P[C = c] = 2^{-n} \cdot \sum_{i}^{2^n} P[M = m_i]$$
$$P[C = c] = 2^{-n}$$

Next, we substitute this and $K = m_i \oplus c$ into our Bayes' law formula from before:

$$P[M = m|C = c] = \frac{P[C = c|M = m] \cdot P[M = m]}{P[C = c]}$$
$$P[M = m|C = c] = \frac{P[K = m \oplus c] \cdot P[M = m]}{2^{-n}}$$
$$P[M = m|C = c] = \frac{2^{-n} \cdot P[M = m]}{2^{-n}}$$
$$P[M = m|C = c] = P[M = m]$$

which concludes the proof. We also note that a random n-bitstring having a Shannon entropy of n immediately implies that $n \sim \text{Uniform}[0,1]^n$. This means the definitions of every ciphertext having n bits of entropy and being perfectly secret are equivalent.

This leads us to perhaps the most important result in all of informatic-theoretic cryptography: Shannon's theorem.

Theorem 2: Shannon's Theorem

For a cipher to be perfectly secure, the key must be at least as long as the message.

Proof: Suppose not. Then we have the set of all keys K and the set of all messages M, and $|K| < |M|$. We wish to prove that there exists a ciphertext c such that no $k \in K$ is able to satisfy $E(k,m) = c$ (informally, we prove that there exists a ciphertext that cannot be constructed). Since c cannot be constructed, $P[C = c] = 0$, which contradictions our earlier proof in that $P[C = c] = 2^{-n}$ for n-bit messages. If at least one message is unable to be constructed, an attacker has better than 2^{-n} probability of guessing the right message – equivalently, the entropy of the ciphertext is less than n.

To show this is case, consider the set of all possible ciphertexts generated by our scheme as $C' = \{E(k,m) : k \in K, m \in M\}$. It follows that $|C'| < |C|$, where $|C|$ is the set of ciphertexts where $|K| = |M|$. Now choose some $c' \in M \setminus C'$. This cipher cannot be generated by the given key set, so we have finished our proof. To show this in more detail, we proceed by contradiction:

Recall that ciphers satify the requirement that $D(k,E(k,m)) = m$ for all k,m. Suppose we could in

fact generate c'. Then, $D(k, c') = m'$ for some $k \in K$. If this were the case, however, $c' \in C'$, so we reach a contradiction.

This result is extremely damning for information-theoretic cryptography, since it proves we cannot construct perfect systems unless we have enough truly random key material to match message length.

Drawbacks and Pitfalls

You might be wondering, why is the field of cryptography such a big deal when this perfect cipher exists? Unfortunately, there are some severe drawbacks to using one-time pads. Notably:

- **Setup** – the sender and receiver must have met in person to trade the key between themselves beforehand.
- **True Randomness** – generating a perfectly random key is rather difficult.
- **Convenience** – you need a **lot** of these random bits to convey messages of appreciable length.
- **Reusability** – the key can never be re-used, or else the one-time pad loses perfect secrecy.
- **Lack of Message Authentication** – an attacker can replace the ciphertext with any encrypted message they desire if they know the ciphertext and plaintext.

Due to this, there is almost no use of one-time pad in consumer contexts. Despite these drawbacks, one-time pads have been in use throughout modern history by spy agencies and anything requiring utmost secrecy.

Levels of Security

Modern cryptography requires modern levels of security, which can be classified into a few groups based on their level of protection. In the previous note, we discussed **perfect secrecy**, which entails a ciphertext is indistinguishable from random noise. A more practical version of this is **semantic security**.

Definition 4: Semantic Security

Semantic security is a property of encryption algorithms that states that a **probabilistic, polynomial-time adversary** will not be able to distinguish the ciphertext from random noise, excluding message length. Informally, an adversary should not be able to distinguish between a given ciphertext and random noise with probability greater than $\frac{1}{2}$ within polynomial time.

Note that semantic security is weaker than perfect secrecy, because it only says polynomially-bounded actors should be prevented from distinguishing them. Consider an adversary that can

enumerate every possible message/key/IV until they find one that might look like English (therefore, the data has high probability of being the ciphertext and not random noise). This would take exponential time (in the number of bits), so the scheme is still semantically secure, but not perfectly secure. One-time pad still fulfills perfect secrecy, because even with exponential time, an attacker cannot differentiate data as it could decrypt to anything.

Definition 5: Known-Plaintext Attack

A **known-plaintext attack** occurs when the attacker knows the plaintext (or some of it) and its corresponding ciphertext. Formally, they are given $(M, E(k, M))$.

Classicial cryptography is incredibly weak to known-plaintext attacks – can you figure out why this is a problem for the Caesar cipher as an example?

We wish to construct ciphers that are immune to these types of attack. This standard of security is named **indistinguishable under chosen-plaintext attack**, or **IND-CPA** for short.

Definition 6: IND-CPA

A cipher is **IND-CPA** secure if and only if an attacker cannot distinguish between the real ciphertext and random noise with probability greater than $\frac{1}{2} + \varepsilon$, even if they know the corresponding plaintext. ε is some negligible value in this case (for example, 2^{-128}).

Exercise: Prove the IND-CPA security of a one-time pad assuming keys are not reused.

A more intuitive view of IND-CPA can be found through the IND-CPA game, which involves two players: the attacker and a oracle.

IND-CPA Game

1. The attacker can ask for any number of encryptions on arbitrary messages. Formally, they can request $E(k, m)$ for any m.
2. The attacker generates some plaintext m of their choice and sends it to the oracle.
3. The oracle computes $c_0 = E(k, m)$ and $c_1 = E(k, m')$ for some $m' \neq m$.
4. The oracle flips a fair coin ($b = \{0, 1\}$) and returns c_b to the attacker.
5. If the attacker can determine whether $c_b = E(k, m)$ with probability $> \frac{1}{2} + \varepsilon$, they win. Otherwise, the game repeats as many times as the attacker wants.

This game tests whether the attacker can reliably distinguish the outcome of an encryption if they know the plaintext and not the key. They win if they can do so **even once**! Note that this does not mean they have to guess correctly only once, but rather have enough information in any given round to be able to guess with slightly better than random odds. Notably, **any scheme that is entirely deterministic cannot be IND-CPA secure!** To see why, note that the attacker can play the game multiple times. First they ask for the encryption of m, and receive $E(k, m)$. They then send m over to play the game, and can just compare if $c_b = E(k, m)$ like they received earlier. Therefore, they win with probability 1.

Note that the lack of IND-CPA security does not necessarily mean attackers can read the message or leak the key. It simply means they can learn **some** information about the plaintext from the ciphertext. The severity of that varies by scheme, but all modern cryptosystems are functionally required to be IND-CPA secure anyway.

Definition 7: IND-CCA (indistinguishable under chosen-ciphertext attack)

A cipher is **IND-CCA** secure if and only if an attacker cannot distinguish between the real ciphertext and random noise with probability greater than $\frac{1}{2} + \varepsilon$, even if they know the corresponding plaintext **and are able to decrypt arbitrary messages as well**.

IND-CCA is the next step up from IND-CPA. A chosen-ciphertext attacks occurs when the adversary is allowed to encrypt **and** decrypt arbitrary messages at will by asking an oracle. There are actually two forms of IND-CCA: IND-CCA1 (non-adaptive) and IND-CCA2 (adaptive). IND-CCA1 follows the same game as IND-CPA, except they are only allowed to ask for decryptions **before** the "game" starts. In IND-CCA2, the adversary can ask for decryptions even **after** the oracle has returned the two challenge ciphertexts, with the caveat that they cannot ask for the decryption of the challenge plaintexts (which would render the game trivial).

We will mostly focus on IND-CPA security.

Randomness

At the heart of many cryptographic algorithms lies a reliance on a stream of secure random bits – where normal PRNGs fall short. Also known as cryptographically-secure psuedorandom number generators, these systems are extraordinarily complex in nature and are a large attack surface for those looking to break a cryptosystem.

Definition 8: Psuedo-Random Number Generators

A **psuedorandom number generator**, also known as PRNG for short, is a deterministic, software-based random number generator. Given a starting seed x, it is *guaranteed* that all generated numbers will be fixed regardless of any other factors.

Definition 9: Cryptographically-Secure Psuedo-Random Number Generators

A **cryptographically-secure psuedorandom number generator**, also known as CSPRNG, is a PRNG that is random enough to be used in cryptographic applications, alongside some key security properties.

First, how does a PRNG work? Why can't computers just come up with random numbers? The answer lies in a computer's biggest strength – its rigid, logical nature. A computer can only execute preset instructions – there is no way for it to "invent" randomness (entropy). Without outside influence, there is no way to create a non-deterministic RNG, and as such is impossible to use for cryptography.

Such a drawback is why computer-based RNGs are called PRNGs, as they aren't *really* random – they just appear random to the naked eye. However, as we will shortly see, there exist a multitude of hidden patterns within some RNGs.

A key property of CSPRNGs is **rollback resistance**, which is the property of being infeasible to figure out previous RNG states given a state at time t. Informally, if the attacker learns the PRNG state at some time, they still can't figure out what was output before that time.

Uses of RNGs

Why do we need RNGs to begin with? A source of truly random (or at least really close to truly random) bits is of utmost important for many algorithms. In fact, you'd be very hard pressed to find a modern cryptosystem that did not use random bits somewhere (without randomness, there's no sense of security). Key generation, choosing prime numbers, generating salts, and more rely upon a secure source of random bits.

A famous example of a PRNG's widespread use (and consequent importance of remaining secure) was the NSA's Dual_EC_DRGB. This was a PRNG that was recommended for use by the National Institute of Standards and Technology (NIST), which is a major authority on the security of modern cryptosystems (potential conflicts of interest within the US government are left as a thought exercise to the reader). As such, many companies such as RSA Security used it to implement their cryptographic algorithms.

Later, it was discovered that the NSA had introduced a backdoor into the system in which a part that was considered computationally infeasible to reverse was actually trivial for a party knowing a certain constant. This vulnerability would have allowed them to break the encryption of millions of users.

Methods of RNG

There are two primary sources of random number generation in modern computers – software RNG and hardware RNG.

Software RNG relies on existing computer hardware and pre-programmed instructions to generate psuedorandom numbers, whereas hardware RNG monitors random physical phenomena and converts it to random bits. This phenomena may include thermal noise, hard drive RPM, and sometimes quantum physics. While these phenomena are technically predictable, in practice it is impossible for an attacker to know the exact physical state down to subatomic particles, and is thus secure in theory (we will later see how this security does not always hold in practice).

Definition 10: Software RNG

Software RNG uses programs and existing hardware to generate psuedorandom numbers. They cannot produce their own entropy.

Definition 11: Hardware RNG

Hardware RNG involves using dedicated hardware to collect entropy from outside physical processes theoretically not related to computer execution.

Hardware RNGs are rare outside of mission-critical cryptographic systems such as those used by cloud providers or government agencies. However, there is growing inclusion of these modules to consumer systems.

Now, how do software RNGs actually generate "random" numbers? The most common method is the linear congruential generator, which is a fancy term for a function like this:

$$f_{n+1} = \alpha f_n + \beta \pmod{n}$$
$$\alpha, \beta, n \in \mathbb{Z}$$

The most notable drawback of such a system is its cap at n, that is, it can never generate a number equal or greater to n. If an adversary were to discover this, it would cut a potential search space for a hypothetical key by an incredible amount.

Moreover, this system is trivally weak to rollback attacks. If the attacker compromises our system and learns f_n, α, β, they can always compute $f_{n-1} \equiv \alpha^{-1}(f_n - \beta) \mod n$, eventually constructing all previous outputs.

Relation to Information Theory

Randomness is inherently tied to information theory, like we covered in a previous note. We can measure the "strength" of a CSPRNG by it's level of **seeded entropy**, or how much entropy the original seed had. If we can assume the CSPRNG functions properly, its outputs will be only as random as its initial seeding. This is why having a high-entropy seed is crucial! A low-entropy seed will allow attackers to more easily predict the outputs, or even learn the state altogether.

For example, there are many **password-based key derivation functions** (PBKDFs) that allow a user to generate infinitely many keys based off one original password. However, these outputs will never have any more entropy than the original input!

Unfortunately, most passwords have awful entropy. A huge amount of users use passwords known to be common, and more still use words from a dictionary as part of the password. This means attackers can guess passwords extremely fast!

Exploiting Weak Randomness

Let's consider the simplified, but plausible situation in which you are trying to break the PRNG of a computer in order to attack something else in the future. You know that it sets it's state bits as a concatentation of the following:

- CPU temperature at 1 second after boot
- UNIX timestamp at boot
- Process ID of the PRNG process

If one were to reconstruct the initial state, it is possibly to determine the output of the PRNG forever. In this case, the initial state is considerably easy to narrow down to a plausible range: CPU temperatures fluctuate a bit, but stay within a range of a few Celsius at most at boot, the timestamp is easily recovered by a root-level program, and process IDs follow a predictable pattern. Therefore, you might only have to guess a few hundred possible configurations at most due to its low entropy, compared to the "full" initial state of 2^{64} possibilities or so.

Contributors:
- Ryan Cottone