AMERICAN UNIVERSITY OF BEIRUT

Applied Cryptography

CMPS 297AD/396AI

Fall 2025

Part 1: Provable Security

1.3: Provable Security & Computational Cryptography

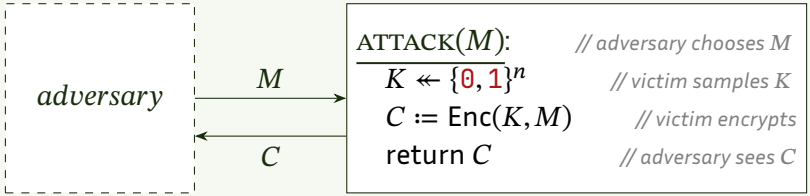
Nadim Kobeissi

https://appliedcryptography.page

Section 1

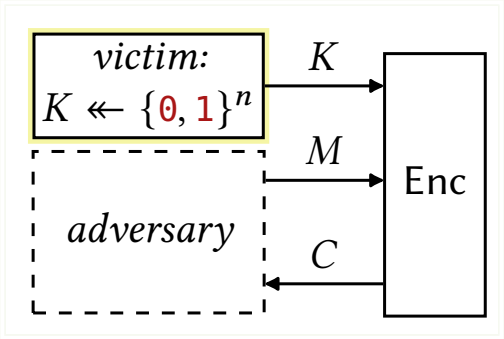
Provable Security

Last time, we defined subroutines



Subroutines

- "Victim" chooses their key.
- Adversary chooses the message and receives the ciphertext.
- We say that **the adversary has access to an encryption oracle**.



Source: The Joy of Cryptography

Attack scenarios as libraries

- This is a **library** with two **subroutines** and a **global variable** D.
- Victim holds a 6-sided die.
- The first line of the library represents an initialization step.
- Attacker can call the subroutines at any time, and:
  1. Make a guess about the current value of the die and learn whether the guess was correct.
  2. Instruct the victim to (privately) re-roll the die.

$\mathcal{L}_{\text{dice-guess}}$

$D \twoheadleftarrow \{1, 2, 3, 4, 5, 6\}$

$\underline{\text{GUESS}(G)}$:
return $D == G$

$\underline{\text{REROLL}(G)}$:
$D \twoheadleftarrow \{1, 2, 3, 4, 5, 6\}$

One-time pad

From the adversary's perspective...

$\textsc{Attack}(M)$:

$K \twoheadleftarrow \{0,1\}^n$

$C := K \oplus M$

return C

$\approx$

(indistinguishable from)

$\textsc{Junk}(M)$:

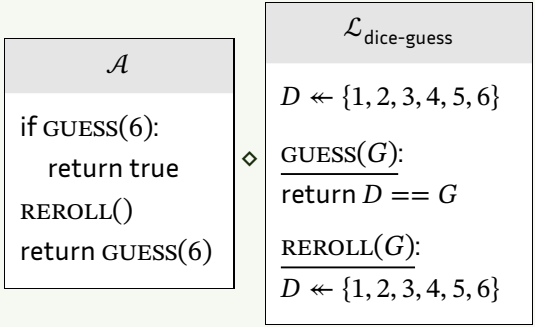
$C \twoheadleftarrow \{0,1\}^n$

return C

“Real or random?”

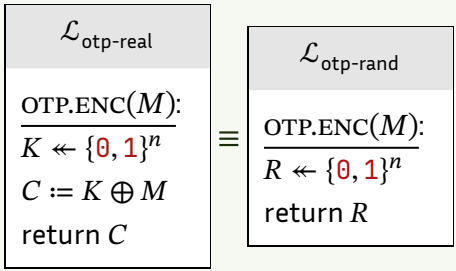
Programs and libraries

- $\mathcal{A}$ is a **program** that calls **library** $\mathcal{L}_{\text{dice-guess}}$.
- Programs can only see the output of function calls to libraries.
  - Programs can't read the values of library variables,
  - Programs can't measure how long it took to run a subroutine,
  - Etc.



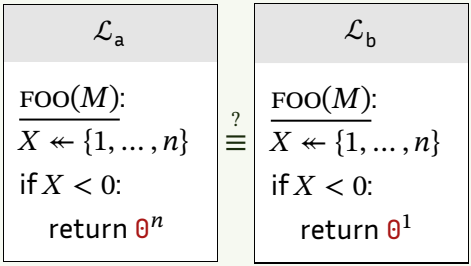
When are libraries interchangeable?

- Libraries are **interchangeable** when they:
  - Have the same interface,
  - $\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \texttt{true}] = \Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow \texttt{true}]$
- i.e. when their usage and output is indistinguishable from the adversary's perspective when they are paired with $\mathcal{A}$.
- A lot of the time, $\mathcal{A}$'s mission is to try to *distinguish* between $\mathcal{L}_1$ and $\mathcal{L}_2$.



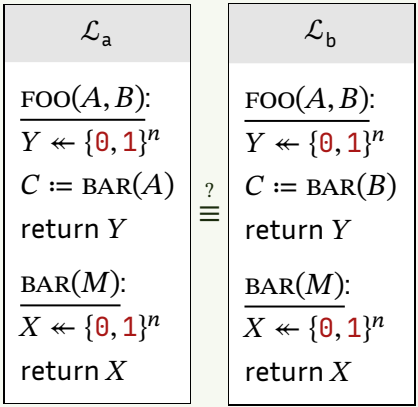
When are libraries interchangeable?

- Are these libraries interchangeable?
- **Yes!** Their only difference happens in *unreachable lines of code*.



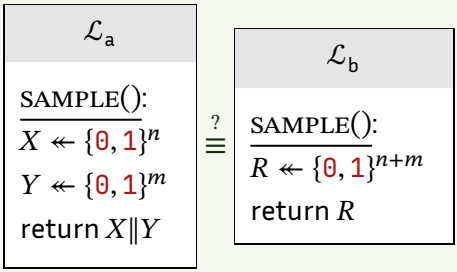
When are libraries interchangeable?

- Are these libraries interchangeable?
- **Yes!** Their only difference is the value they assign to a *variable that is never actually used*.



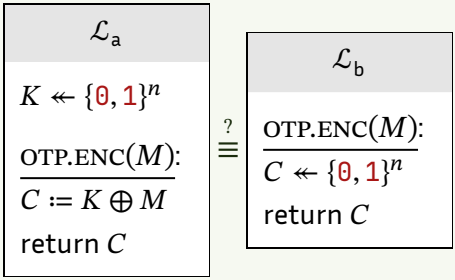
When are libraries interchangeable?

- Are these libraries interchangeable?
- $\|$ denotes string concatenation.
  - $\texttt{Po}\|\texttt{tato} = \texttt{Potato}$
- **Yes!** Outputting the concatentaion two randomly sampled uniform strings of lengths n and m is the same as outputting a single random string of length $n + m$.



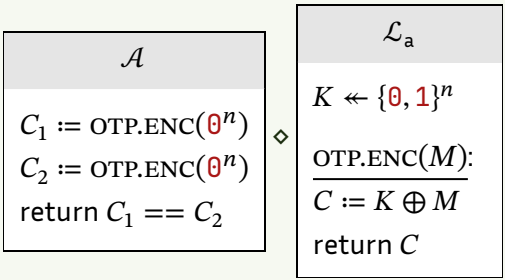
When are libraries interchangeable?

- Are these libraries interchangeable?
- **No!** The first library uses the same K for all subsequent encryptions, which breaks OTP security.
  - Recall that OTP requires keys to be only used once (hence the name).



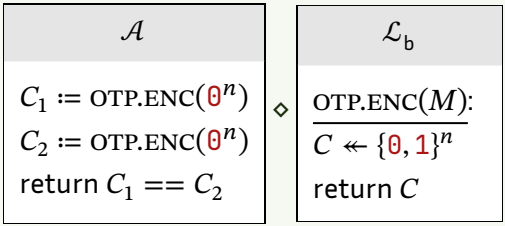
How do we show the distinguisher?

- What happens if we call $\mathcal{A} \diamond \mathcal{L}_{\mathsf{a}}$ multiple times?
  - We'll get the same output $\equiv K$ each time!
- $\Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{a}} \Rightarrow \mathtt{true}\right] = 1$,



How do we show the distinguisher?

- What happens if we call $\mathcal{A} \diamond \mathcal{L}_{\mathsf{a}}$ multiple times?
  - We'll get the same output $\equiv K$ each time!
- $\Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{a}} \Rightarrow \texttt{true}\right] = 1$,
- $\Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{b}} \Rightarrow \texttt{true}\right] = \frac{1}{2^n}$.



OTP: why key re-use is bad

$$
\begin{aligned}
C_i \oplus C_j &= (K \oplus M_i) \oplus (K \oplus M_j) \\
&= K \oplus K \oplus M_i \oplus M_j \\
&= \mathtt{0}^n \oplus M_i \oplus M_j \\
&= M_i \oplus M_j
\end{aligned}
$$

- $\Pr\left[\mathcal{A} \diamond \mathcal{L}_{\mathsf{a}} \Rightarrow \mathtt{true}\right] = 1^a$

[a] Interactive demo: https://www.douglas.stebila.ca/teaching/visual-one-time-pad/

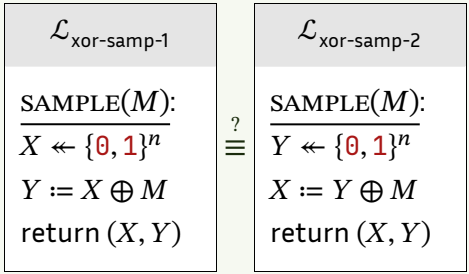
$\mathcal{A}$
$M_1 \twoheadleftarrow \{\mathtt{0},\mathtt{1}\}^n$
$M_2 \twoheadleftarrow \{\mathtt{0},\mathtt{1}\}^n$
$C_1 \coloneqq \text{OTP.ENC}(M_1)$
$C_2 \coloneqq \text{OTP.ENC}(M_2)$
return $C_1 \oplus C_2 == M_1 \oplus M_2$

$\diamond$

$\mathcal{L}_{\mathsf{a}}$
$K \twoheadleftarrow \{\mathtt{0},\mathtt{1}\}^n$
$\underline{\text{OTP.ENC}(M)}$:
$C \coloneqq K \oplus M$
return C

Proving two libraries are interchangeable

- Are these libraries interchangeable?
- Let's find out!
- **Goal:** transform $\mathcal{L}_{\text{xor-samp-1}}$ to $\mathcal{L}_{\text{xor-samp-2}}$, proving that each transformation step does not effect any change on calling programs.



Proving two libraries are interchangeable

Step 1

- We start at $\mathcal{L}_{\text{xor-samp-1}}$.

$\mathcal{L}_{\text{xor-samp-1}}$

$\underline{\textsc{sample}(M)\text{:}}$

$X \twoheadleftarrow \{0,1\}^n$

$Y \coloneqq X \oplus M$

return (X, Y)

Proving two libraries are interchangeable

Step 2

- Let's add a new variable X'.
- Note that $X = X'$:
  - $X' = Y \oplus M = (X \oplus M) \oplus M = X$.

$\mathcal{L}_{\text{hyb-1}}$

$\underline{\text{SAMPLE}(M)}$:

$X \twoheadleftarrow \{0,1\}^n$

$Y := X \oplus M$

$X' := Y \oplus M$

return (X, Y)

Proving two libraries are interchangeable

Step 3

- So, we can return (X', Y) without any change on the library's effect.
  - $X' = Y \oplus M = (X \oplus M) \oplus M = X$.

$\mathcal{L}_{\text{hyb-2}}$

$\textsc{sample}(M)$:

$X \twoheadleftarrow \{0,1\}^n$

$Y \coloneqq X \oplus M$

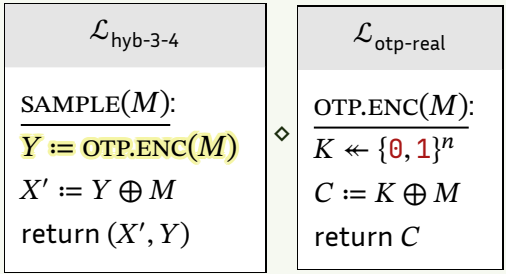
$X' \coloneqq Y \oplus M$

return (X', Y)

Proving two libraries are interchangeable

Step 4

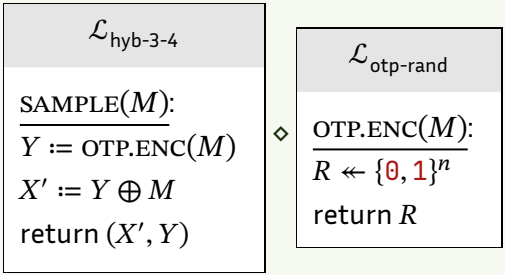
- The first two lines of $\text{SAMPLE}(M)$ are the same as $\text{OTP.ENC}(M)$, so we link it and use it instead.



Proving two libraries are interchangeable

Step 5

- The first two lines of $\text{SAMPLE}(M)$ are the same as $\text{OTP.ENC}(M)$, so we link it and use it instead.
- Recall that $\mathcal{L}_{\text{otp-real}} \equiv \mathcal{L}_{\text{otp-rand}}$!
- So, we replace $\mathcal{L}_{\text{otp-real}}$ with $\mathcal{L}_{\text{otp-rand}}$.



Proving two libraries are interchangeable

Step 6

- The first two lines of $\text{SAMPLE}(M)$ are the same as $\text{OTP.ENC}(M)$, so we link it and use it instead.
- Recall that $\mathcal{L}_{\text{otp-real}} \equiv \mathcal{L}_{\text{otp-rand}}$!
- So, we replace $\mathcal{L}_{\text{otp-real}}$ with $\mathcal{L}_{\text{otp-rand}}$.
- We inline $\mathcal{L}_{\text{otp-rand}}$ back into our main library.

$\mathcal{L}_{\text{hyb-5}}$

$\text{SAMPLE}(M)$:

$Y \twoheadleftarrow \{0,1\}^n$

$X' \coloneqq Y \oplus M$

return (X', Y)

Proving two libraries are interchangeable

Step 7

- The first two lines of $\text{SAMPLE}(M)$ are the same as $\text{OTP.ENC}(M)$, so we link it and use it instead.
- Recall that $\mathcal{L}_{\text{otp-real}} \equiv \mathcal{L}_{\text{otp-rand}}$!
- So, we replace $\mathcal{L}_{\text{otp-real}}$ with $\mathcal{L}_{\text{otp-rand}}$.
- We inline $\mathcal{L}_{\text{otp-rand}}$ back into our main library.
- Finally, we rename X' to X.

$\mathcal{L}_{\text{hyb-6}}$

$\text{SAMPLE}(M)$:

$Y \twoheadleftarrow \{0,1\}^n$

$X := Y \oplus M$

return (X, Y)

Proving two libraries are interchangeable

Result

- Interchangeable!
- Note how we had to show equivalence **at each step**.

$\mathcal{L}_{\text{hyb-6}}$		$\mathcal{L}_{\text{xor-samp-2}}$
$\underline{\text{SAMPLE}(M)}$:		$\underline{\text{SAMPLE}(M)}$:
$Y \twoheadleftarrow \{0,1\}^n$	$\equiv$	$Y \twoheadleftarrow \{0,1\}^n$
$X \coloneqq Y \oplus M$		$X \coloneqq Y \oplus M$
return (X, Y)		return (X, Y)

Cryptographic primitives

- Specific algorithms like OTP are important in cryptography, but OTP is just one instance of an **encryption scheme**.
- In cryptography, useful abstractions like “encryption scheme” are called **primitives**.
- Three things are important when defining a cryptographic primitive:
  1. **Syntax**: The basic raw interface - algorithms, inputs/outputs and their types.
  2. **Correctness**: Basic functionality without adversaries - e.g., “decryption should be the inverse of encryption”.
  3. **Security**: Guarantees that hold in specific attack scenarios with adversaries.

Cryptographic primitives

Σ: a symmetric-key encryption scheme

- $\Sigma.\textbf{KeyGen}() = K$
  - Input: none
  - Output: key $K \in \Sigma.\mathcal{K}$ (the "key space").
- $\Sigma.\textbf{Enc}(K, M) = C$
  - Input: key $K \in \Sigma.\mathcal{K}$, plaintext $M \in \Sigma.\mathcal{M}$ (the "message space").
  - Output: ciphertext $C \in \Sigma.\mathcal{C}$.
- $\Sigma.\textbf{Dec}(K, C) = M$
  - Input: key $K \in \Sigma.\mathcal{K}$, ciphertext $C \in \Sigma.\mathcal{C}$ (the "ciphertext space").
  - Output: plaintext $M \in \Sigma.\mathcal{M}$.

Correctness of a SKE

Correctness for SKE

An SKE scheme Σ is correct if encryption and decryption are inverses, in the following sense:

$$\Pr\left[\Sigma.\mathsf{Dec}(K, \Sigma.\mathsf{Enc}(K, M)) = M\right] = 1$$

for all $M \in \Sigma.\mathcal{M}$ and $K \in \Sigma.\mathcal{K}$.

- The definition involves a probability because $\Sigma.\mathsf{Enc}$ may be a randomized algorithm.
- This means that decryption should **always** recover the original message.
- Even if encryption adds randomness, decryption must be deterministic for each key-ciphertext pair.

One-time secrecy of a SKE

One-time Secrecy for SKE

An SKE scheme Σ has one-time secrecy if the following libraries are interchangeable:

$\mathcal{L}^{\Sigma}_{\text{ots-real}}$

$\text{OTS.ENC}(M)$:
$K \twoheadleftarrow \Sigma.\mathcal{K}$
$C \coloneqq \Sigma.\text{Enc}(K, M)$
return C

$\equiv$

$\mathcal{L}^{\Sigma}_{\text{ots-rand}}$

$\text{OTS.ENC}(M)$:
$C \twoheadleftarrow \Sigma.\mathcal{C}$
return C

An encryption scheme has one-time secrecy if its ciphertexts are uniformly distributed, when keys are sampled uniformly, kept secret, and used for only one encryption, and no matter how the plaintexts are chosen.

Reminder: AND ($\wedge$)

- Let's replace $\oplus$ with $\wedge$. What would happen?
- Output no longer uniform!

A	B	A $\wedge$ B
0	0	0
0	1	0
1	0	0
1	1	1

Table: Truth table for AND operation

$\underline{\text{ATTACK}(M):}$

$K \twoheadleftarrow \{0,1\}^n$

$C := K \wedge M$

return C

Creating a distinguisher program

- Can you write a distinguisher program showing that these libraries are **not** interchangeable?

$\mathcal{A}$

$M \coloneqq \mathtt{0}^n$

$C \coloneqq \text{OTS.ENC}(M)$

return $C == \mathtt{0}^n$

$\mathcal{L}_{\text{ots-real}}$

$\text{OTS.ENC}(M)$:

$K \twoheadleftarrow \{\mathtt{0},\mathtt{1}\}^n$

$C \coloneqq K \wedge M$

return C

$\overset{?}{\equiv}$

$\mathcal{L}_{\text{ots-rand}}$

$\text{OTS.ENC}(M)$:

$C \twoheadleftarrow \{\mathtt{0},\mathtt{1}\}^n$

return C

- $\Pr\left[A \diamond \mathcal{L}_{\text{ots-real}} \Rightarrow \mathtt{true}\right] = 1$
- $\Pr\left[A \diamond \mathcal{L}_{\text{ots-rand}} \Rightarrow \mathtt{true}\right] = \frac{1}{2^n}$

Section 2

Computational Cryptography

Computational notions in cryptography

- Security definitions in theoretical cryptography can be too strict:
  - Even attacks requiring a trillion years of computation.
  - Even attacks with probability lower than winning the lottery 100 times.
- Modern provable security takes a more practical approach:
  - We dismiss attacks with “astronomically” high computational cost.
  - We dismiss attacks with “astronomically” tiny success probability.
- Instead of “no attack can succeed, not even in principle”,
- We prove: “every attack has either astronomically high cost or astronomically small success probability”.

The concrete approach to provable security

- In the concrete approach to provable security, we aim to be as quantitative as possible about security claims.
- We rarely say definitively that a cryptographic algorithm “is secure”, as we did with OTP.
- Instead, we use statements like:
  - *“Any attack that expends at most 2^{80} effort can succeed with probability no better than 2^{-64}.”*
- It is up to the user to judge whether this quantitative level of security is acceptable based on their use-case.
- This gives users a concrete basis for security decisions.

Monetary cost of huge computations

- One way to think about huge computations is their monetary cost.
- The table below shows roughly how much a computation involving 2^n CPU cycles would cost on the cheapest available Amazon EC2 cloud computing service:

Clock cycles	Approx cost (USD)	Point of reference
2^{50}	$3.50	cup of coffee
2^{55}	$100	dinner at a high-end restaurant
2^{65}	$130,000	apartment in Achrafieh
2^{75}	$130 million	budget of one of the Harry Potter movies
2^{85}	$140 billion	GDP of Hungary
2^{92}	$20 trillion	GDP of the United States
2^{99}	$2 quadrillion	all of human economic activity since 300,000 BCE
2^{128}	a lot!	a billion human civilizations' worth of effort

Understanding tiny probabilities

- Let's put extremely small probabilities in perspective:
  - In 2009, Patricia Demauro rolled dice 154 consecutive times without getting a 7 in craps.
  - Probability of this event: $(30/36)^{154} \approx 2^{-40.6}$
  - Often cited as one of the most improbable documented events in gambling history.
- Other extremely unlikely gambling events:
  - In 1943, a roulette wheel reportedly landed on red 32 consecutive times.
  - Probability: $(18/38)^{31} \approx 2^{-33.4}$
  - Winning the American Powerball lottery: $2^{-28.1}$
  - Winning Powerball two consecutive weeks: $2^{-56.2}$
- These examples help us intuitively grasp what probabilities like 2^{-40} or 2^{-50} actually mean.

Computational scale of Bitcoin mining

- Let's consider cryptocurrencies based on proof-of-work, like Bitcoin.
- These systems incentivize users to perform truly obscene amounts of computation.
- In Bitcoin's proof-of-work mechanism:
  - Users race to perform SHA-256 hash computations as fast as possible.
  - The collective Bitcoin network has performed approximately 2^{95} SHA-256 hashes in total.
  - In the last 12 months alone: approximately $2^{93.6}$ hashes.
  - Current market cap for Bitcoin: approximately 400 billion USD.
- This shows the enormous scale of computation that economic incentives can support.

The asymptotic approach to provable security

- The concrete approach gives practical guidance (e.g., key sizes) but requires managing tedious quantitative details.
- The asymptotic approach:
  - Makes qualitative, all-or-nothing statements
  - Considers behavior as key size approaches infinity
  - Hides tedious quantitative details
- Similar to asymptotic analysis using big-O notation.
- Example: Instead of calculating that an operation takes exactly $16n^2 + 24n + 74$ steps, we can simply say it takes $O(n^2)$ steps

AES: An example of asymptotic security analysis

- The Advanced Encryption Standard (AES) is a common symmetric-key encryption algorithm.
- It comes in different variants: AES-128, AES-192, AES-256.
- In asymptotic analysis, we'd say:
  - AES is secure against all polynomial-time adversaries.
  - Any successful attack must take exponential time in the key length.
- This hides the concrete details about specific computational costs.

- Concrete security for AES-128:
  - Best known attack: $\approx 2^{126.1}$ operations
- For AES-256:
  - Best known attack: $\approx 2^{254.4}$ operations
  - Far beyond capability of any conceivable computation
- Asymptotic approach: Both are "computationally secure"

Polynomial running time

- In the asymptotic approach, we focus only on adversaries that run in polynomial time.
- The security parameter λ determines the level of security:
  - Usually the length of secret keys in bits
  - All algorithms have access to λ as a global variable
  - Example: AES-128 uses $\lambda = 128$, AES-256 uses $\lambda = 256$. ("256-bit security")

Polynomial Running Time

An algorithm runs in polynomial time if there is a polynomial p such that the algorithm takes at most $O(p(n))$ steps on inputs of length n.

Negligible functions

Negligible Functions

A function f is negligible if it approaches zero faster than $1/p(\lambda)$, for every polynomial p. Formally:

- For every polynomial p, there exists λ_0 such that $f(\lambda) < 1/p(\lambda)$ for all $\lambda > \lambda_0$.
- For every polynomial p, we have $p(\lambda) \in O(1/f(\lambda))$.

- **You can probably ignore the definition and just think about this intuitively:**
- In cryptography, we want attack probabilities to be negligible in the security parameter λ.
- Negligible probability: essentially zero for practical purposes when λ is large enough.

Birthday paradox

- In a classroom of 50 students, what's the probability that at least two share a birthday?
  - 2%?
  - 20%?
  - 50%?
  - 97%?
- Many people guess around 15-20%, but the actual probability is about 97%!
- This is counterintuitive because **we're not looking for a specific birthday - we're looking for *any* match among all possible pairs**.
- With 50 students, we have $\binom{50}{2} = 1,225$ possible pairs to check![a]

[a] $\binom{50}{2}$ is a binomial coefficient. It means: *"how many ways can I choose two different items from a set of 50?"*

Birthday paradox

- With just 23 people, probability exceeds 50%
- Formula for n people:

$$P = 1 - \frac{365!}{(365 - n)! \cdot 365^n}$$

- Implication: Finding collisions in a space of size N happens with roughly $\sqrt{N}$ samples.
- This is why many cryptographic systems need large output spaces!

Birthday probabilities

- Many cryptographic algorithms fail if two executions sample the same random value.
- General question: If we take q independent uniform samples from a set of N items, what's the probability some value is chosen more than once?
- This probability is called $\textsc{birthday}(q, N)$

$$\textsc{birthday}(q, N) = 1 - \prod_{i=1}^{q-1} \left(1 - \frac{i}{N}\right)$$

- Surprisingly high probability!
- With $q \approx 1.2\sqrt{N}$, probability ≈ 0.5
- For birthdays: only need 23 people for $> 50\%$ chance

Indistinguishability in computational security

Indistinguishability

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two libraries with the same interface. The **advantage** of a calling program $\mathcal{A}$ in distinguishing $\mathcal{L}_1$ and $\mathcal{L}_2$ is:

$$\left|\Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \texttt{true}\right] - \Pr\left[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow \texttt{true}\right]\right|$$

$\mathcal{L}_1 \approx \mathcal{L}_2$, if every polynomial-time calling program has only negligible advantage in distinguishing them.

- Previously, libraries were interchangeable when probabilities were **identical**.
- Now, libraries are indistinguishable when probabilities are **negligibly close**.
- Again: this is intuitive, no need to stress about the formal definition.

The “bad event” proof technique

Definition

Bad Event Technique

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be libraries that each include a boolean variable named bad, and assume that after bad is set to `true` it remains `true` forever. We say that the bad event is triggered if the library ever sets `bad := true`.

If $\mathcal{L}_1$ and $\mathcal{L}_2$ have identical source code, except for statements reachable only when `bad := true`, then:

$$\left|\Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \texttt{true}\right] - \Pr\left[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow \texttt{true}\right]\right| \leq \Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \text{TRIGGER}(\texttt{bad})\right]$$

The “bad event” proof technique

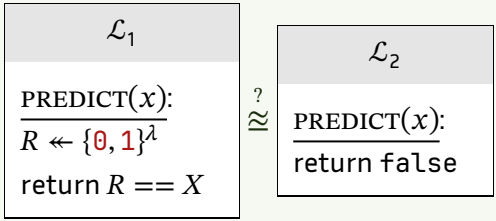
Key insights

- $\mathcal{A}$’s advantage is bounded by $\Pr\left[\mathcal{A} \diamond \mathcal{L}_1 \textsc{trigger}(\mathsf{bad})\right]$.
- Practical application:
  - Define a sequence of hybrid libraries.
  - Identify “bad events” between consecutive hybrids.
  - Show these events occur with negligible probability.
- Enables us to focus on analyzing specific failure cases rather than full behavior.

The “bad event” proof technique

Example

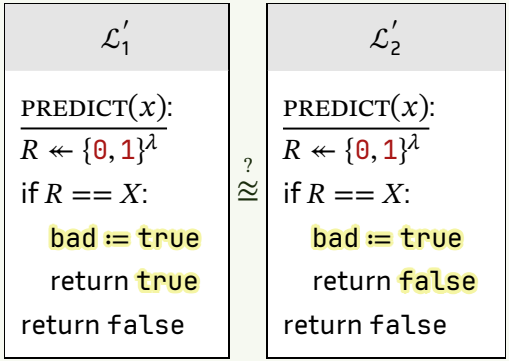
- Are these libraries **computationally indistinguishable?**
- Yes! Let’s prove it use the bad event technique.



The “bad event” proof technique

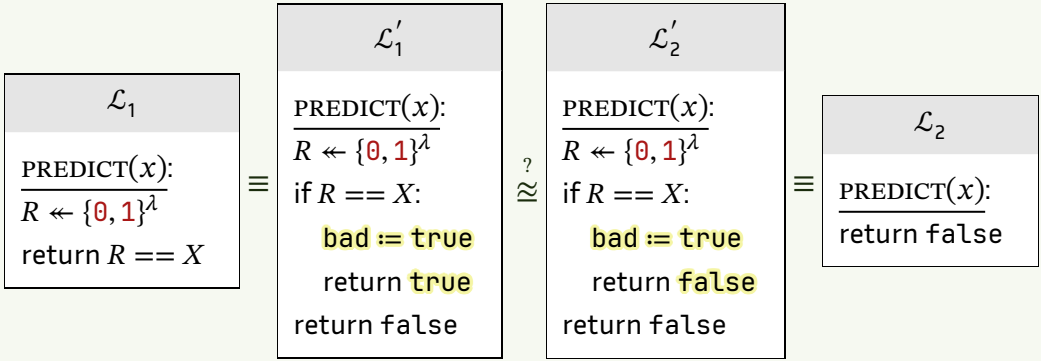
Example

- Are these libraries **computationally indistinguishable?**
- Yes! Let’s prove it use the bad event technique.
- Note: these libraries are interchangeable with $\mathcal{L}_1$ and $\mathcal{L}_2$ (as seen on next slide).



The “bad event” proof technique

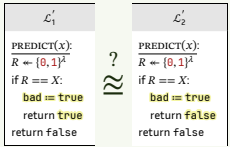
Just to be clear



The “bad event” proof technique

Example

- We need to analyze how likely it is that the bad event happens:
  - The bad event occurs when $R == X$.
  - Since R is randomly chosen from a huge set, this match is extremely unlikely.
  - Even if an adversary makes many attempts, the chance of seeing this bad event remains tiny.
  - As we increase the security parameter λ, the chance becomes vanishingly small.
- Therefore, the libraries are computationally indistinguishable!
- For any $\mathcal{A}$, computationally speaking, we’d get the same distribution of outputs.



The “end-of-time” strategy for bad events

- Sometimes analyzing bad events can be complex, especially when values are chosen by the adversary.
- The end-of-time strategy:
  1. Postpone all bad-event logic to the end of the library execution.
  2. Collect information during normal execution.
  3. Check for bad events only at the very end.
- Advantages:
  - Simplifies analysis by separating normal behavior from bad-event checking.
  - Makes it easier to bound the probability of bad events.
  - Particularly useful for complex cryptographic proofs.
- We’ll use this in the next topic!

AMERICAN UNIVERSITY OF BEIRUT

Applied Cryptography

CMPS 297AD/396AI

Fall 2025

Part 1: Provable Security

1.3: Provable Security & Computational Cryptography

Nadim Kobeissi

https://appliedcryptography.page