Security Definition

- how to write them
- how to understand & interpret them
- how to prove security using the hybrid technique
- how to demonstrate insecurity using attacks against the security definition

We talked about a specific encryption scheme last week.
Let’s consider definitions for something more general.
  - (the key is still used for encrypting once, but, 1 step at a time...)

“Human ingenuity cannot concoct a cipher which human ingenuity cannot resolve.”
— Edgar Allan Poe,
“A Few Words on Secret Writing,”, 1841
Syntax of Symmetric-Key Encryption (SKE)

A symmetric-key encryption (SKE) scheme consists of the following algorithms:

- **KeyGen**: a randomized algorithm that outputs a key $k \in \mathcal{K}$.
- **Enc**: a (possibly randomized) algorithm that takes a key $k \in \mathcal{K}$ and plaintext $m \in \mathcal{M}$ as input, and outputs a ciphertext $c \in \mathcal{C}$.
- **Dec**: a deterministic algorithm that takes a key $k \in \mathcal{K}$ and ciphertext $c \in \mathcal{C}$ as input, and outputs a plaintext $m \in \mathcal{M}$.

- Sometimes we refer to the entire scheme (the collection of all algorithms) by a single variable (“OTP” in §1b: p.3), e.g., $\Sigma$
- We can write $\Sigma$.KeyGen, $\Sigma$.Enc, $\Sigma$.Dec, $\Sigma.K$, $\Sigma.M$, and $\Sigma.C$
(Perfect) Correctness, more formally

- (Last time, informally, “Correctness: \( m = \text{Dec}(k, \text{Enc}(k, m)) \)”)

- For all \((\forall) k \in \Sigma.K\) and all \(m \in \Sigma.M\),
  \[ \Pr[\Sigma.\text{Dec}(k, \Sigma.\text{Enc}(k, m)) = m] = 1 \]

- The definition is written in terms of the probability since \(\text{Enc}()\) is allowed to be randomized.
- Counterexample: \(\text{Enc}(k, m) = 0^\lambda\) (degenerate behavior)
- One might relax the perfect correctness requirement
  - but how? (we might discuss later) and why? [*]
What Doesn’t go into a Security Definition

- Syntax of the scheme/notion
  - albeit it could suggest inherent insecurity, e.g., no rand. KeyGen
- Correctness requirement (e.g., §1b: p.14)
  - Needless (!?) to say, correctness doesn’t imply security
  - e.g., the following Σ’ is always correct but would not be secure
    - Σ’.Enc(κ, m) = m ∀κ ∈ Σ’.K and ∀m ∈ Σ’.M;
    - Σ’.Dec(κ, c) = c ∀κ ∈ Σ’.K and ∀c ∈ Σ’.C;
  - although tampering correctness can be an adversarial goal in some cryptographic notion [**]
Two Styles of Security Definition

- There may be other reasonable ways to formalize security.
- We consider two styles: “Real-or-Random” & “Left-or-Right”
  - There may still be different ways to formalize for each style.

- Security always consider the attacker’s view of the system.
- What is the “interface” that honest users expose to the attacker by their use of the cryptography?
- And does that particular interface benefit the attacker?
Real vs. Random

- “[Encryption] doesn’t reveal any information about the plaintext to the attacker.” (§1b: p.14)
- “An encryption scheme \( \Sigma \) is a good one if its ciphertexts look like random junk to an attacker.”
- The view we said for OTP:

\[
\text{EAVESDROP}(m \in \{0, 1\}^\lambda):
\]
\[
k \leftarrow \{0, 1\}^\lambda
\]
\[
c := k \oplus m
\]
\[
\text{return } c
\]

- A general interface:

\[
\text{CTXT}(m \in \Sigma.M):
\]
\[
k \leftarrow \Sigma.\text{KeyGen}
\]
\[
c := \Sigma.\text{Enc}(k, m)
\]
\[
\text{return } c
\]
There are two inputs to Enc: the key and the plaintext
- The key is our source of randomness and hence security
- The key, generated according to KeyGen, is kept secret
- For now, we still consider the key is used to encrypt once

We consider that the attacker can choose the plaintexts
- A “pessimistic” choice ➔ Giving more power to attackers
- If an SKE scheme is secure against a “powerful” attacker
- then it’s also secure in “more realistic scenarios”
  - where the attacker has some uncertainty about the plaintexts.
“an encryption scheme is a good one if its ciphertexts look like random junk to an attacker when ...”

“each key is secret and used to encrypt only one plaintext, even when the attacker chooses the plaintexts.”

Consider the attacker as a calling program to subroutine:
- can choose the input argument, but
- can’t see values of privately-scoped variables
  - e.g., key $k$
  - (like eavesdrop, a fresh $k$ is chosen each time)
This \texttt{ctxt} subroutine should have the same effect on every calling program (i.e., our attacker) as a \texttt{ctxt} subroutine that (explicitly) samples its output uniformly.

“\(\Sigma\) is secure if, when you plug its KeyGen and Enc algorithms into the template of the \texttt{ctxt} subroutine, the below two implementations of \texttt{ctxt} induce identical behavior in every calling program.”

\begin{align*}
\text{.ctxt}(m \in \Sigma.\mathcal{M}) : & \frac{k \leftarrow \Sigma.\text{KeyGen}}{c := \Sigma.\text{Enc}(k, m)} \quad \text{return } c \\
\text{vs.} & \frac{c \leftarrow \Sigma.C}{\text{return } c}
\end{align*}
A Simple Proof that OTP is RoR-secure

\[
\begin{align*}
\text{CTXT}(m \in \Sigma.M): & \\
& k \leftarrow \Sigma.\text{KeyGen} \\
& c := \Sigma.\text{Enc}(k, m) \\
& \text{return } c \\
\end{align*}
\]

\[
\begin{align*}
\text{CTXT}(m \in \Sigma.M): & \\
& c \leftarrow \Sigma.\text{C} \\
& \text{return } c \\
\end{align*}
\]

real vs. random

\[
\begin{align*}
\text{CTXT}(m): & \\
& k \leftarrow \{0, 1\}^\lambda \quad \text{// KeyGen of OTP} \\
& c := k \oplus m \quad \text{// Enc of OTP} \\
& \text{return } c \\
\end{align*}
\]

\[
\begin{align*}
\text{CTXT}(m): & \\
& c \leftarrow \{0, 1\}^\lambda \quad \text{// C of OTP} \\
& \text{return } c \\
\end{align*}
\]
Left vs. Right

“Σ is secure if, if encryptions of $m_L$ look like encryptions of $m_R$ to an attacker, when each key is secret and used to encrypt only one plaintext, even when the attacker chooses $m_L$ and $m_R$.”

ROR: “an encryption scheme is a good one if its ciphertexts look like random junk to an attacker when each key is secret and used to encrypt only one plaintext, even when the attacker chooses the plaintexts.”

(Exercise: Prove OTP is LoR-secure)

(See Exercise 2.15 for an alternative formalization)
### Programming-Like Terminologies

- A *library* \( L \) is a collection of subroutines & private/static var.
- A library’s *interface* consists of the names, argument types, and output type of all of its subroutines.
- \( \mathcal{A} \diamond L \): a program \( \mathcal{A} \) includes calls to subroutines in \( L \) (*linking*)
- \( \mathcal{A} \diamond L \Rightarrow z \) to denote the event that \( \mathcal{A} \diamond L \) outputs the value \( z \)

An example of \( \mathcal{A} \):
- choosing a random \( m \) and hoping that \( \text{ctxt}(m) \) is just \( m \)?

\[
\begin{align*}
\text{ctxt}(m) :& \quad k \leftarrow \{0, 1\}^\lambda \\
& \quad c := k \oplus m \\
& \quad \text{return } c
\end{align*}
\]

\[
\text{Pr}[\mathcal{A} \diamond L \Rightarrow \text{true}] = 1/2^\lambda.
\]

\[
\begin{align*}
\mathcal{A} :& \quad m \leftarrow \{0, 1\}^\lambda \\
& \quad c := \text{ctxt}(m) \\
& \quad \text{return } m = c
\end{align*}
\]
Another Example (as in the textbook)

- Guessing a string picked uniformly at random:

```
L
s ← {0, 1}^λ
RESET():
  s ← {0, 1}^λ
GUESS(x ∈ {0, 1}^λ):
  return x = s
```

- Several sub-routines co-exist.
- Code outside of any subroutine is run once at initialization time.
- Variables defined at initialization are available in all subroutine scopes (but still not to the calling program).
Let $L_0$ and $L_1$ be two libraries that have the same interface.

We say that $L_0$ and $L_1$ are interchangeable, i.e., $L_0 \equiv L_1$ if for all programs $A$ that output a boolean value

\[
\Pr[A \diamond L_0 \Rightarrow \text{true}] = \Pr[A \diamond L_1 \Rightarrow \text{true}]
\]

We can also call $A$ as a distinguisher
Textbook Examples useful in Our Proofs

\[
\begin{align*}
\text{FOO}(x): &\quad \text{if } x \text{ is even: return 0} \\
&\quad \text{else if } x \text{ is odd: return 1} \\
&\quad \text{else: return -1} \\
\text{FOO}(x): &\quad \text{if } x \text{ is even: return 0} \\
&\quad \text{else if } x \text{ is odd: return 1} \\
&\quad \text{else: return } \infty \\
\text{FOO}(x): &\quad k \leftarrow \{0, 1\}^λ \\
&\quad y \leftarrow \{0, 1\}^λ \\
&\quad \text{return } k \oplus y \oplus x \\
\text{FOO}(x): &\quad \text{return } \text{BAR}(x) \\
\text{BAR}(a, b): &\quad k \leftarrow \{0, 1\}^λ \\
&\quad \text{return } k \oplus a \\
\text{FOO}(x): &\quad \text{return } \text{BAR}(x, x) \\
\text{BAR}(a, b): &\quad k \leftarrow \{0, 1\}^λ \\
&\quad \text{return } k \oplus a \\
\text{FOO}(x): &\quad \text{if } k \text{ not defined: return } \infty \\
&\quad k \leftarrow \{0, 1\}^λ \\
&\quad \text{return } k \oplus x \\
\text{FOO}(x): &\quad \text{for } i = 1 \text{ to } n: \\
&\quad \text{BAR}(x, i) \\
&\quad \text{for } i = n + 1 \text{ to } λ: \\
&\quad \text{BAR}(x, i) \\
\text{FOO}(x): &\quad z \leftarrow \{0, 1\}^{2λ} \\
&\quad \text{return } z \\
\end{align*}
\]
“One-time Uniform Ciphertexts”

\[ L^\Sigma_{ots$\$\text{-real}} \]

\[
\begin{align*}
\text{ctxt}(m \in \Sigma.M): \\
k &\leftarrow \Sigma.\text{KeyGen} \\
c &\leftarrow \Sigma.\text{Enc}(k, m) \\
\text{return } c
\end{align*}
\]

\[ L^\Sigma_{ots$\$\text{-rand}} \]

\[
\begin{align*}
\text{ctxt}(m \in \Sigma.M): \\
c &\leftarrow \Sigma.C \\
\text{return } c
\end{align*}
\]

\[ \equiv \]

the “\$” symbol to denote randomness, as in coin tossing

Security of OTP:

\[ L_{\text{otp-real}} \]

\[
\begin{align*}
\text{EAVESDROP}(m \in \{0, 1\}^\lambda): \\
k &\leftarrow \{0, 1\}^\lambda // \text{OTP.KeyGen} \\
\text{return } k \oplus m // \text{OTP.Enc}(k, m)
\end{align*}
\]

\[ L_{\text{otp-rand}} \]

\[
\begin{align*}
\text{EAVESDROP}(m \in \{0, 1\}^\lambda): \\
c &\leftarrow \{0, 1\}^\lambda // \text{OTP.C} \\
\text{return } c
\end{align*}
\]

\[ \equiv \]

(L_{\text{otp-real}} and L_{\text{otp-rand}} will be recalled in this chapter later)
One-time Security (Left-or-Right Style)

- Formal definition:
  An encryption scheme $\Sigma$ has one-time secrecy if:

\[
\begin{array}{ll}
\mathcal{L}_{\text{ots-L}}^\Sigma & \equiv \\
\text{EAVESDROP}(m_L, m_R \in \Sigma.\mathcal{M}) & \\
  & k \leftarrow \Sigma.\text{KeyGen} \\
  & c \leftarrow \Sigma.\text{Enc}(k, m_L) \\
  & \text{return } c
\end{array}
\]

\[
\begin{array}{ll}
\mathcal{L}_{\text{ots-R}}^\Sigma & \equiv \\
\text{EAVESDROP}(m_L, m_R \in \Sigma.\mathcal{M}) & \\
  & k \leftarrow \Sigma.\text{KeyGen} \\
  & c \leftarrow \Sigma.\text{Enc}(k, m_R) \\
  & \text{return } c
\end{array}
\]
Common Pitfalls

- $L_0 \equiv L_1$: $\Pr[A \diamond L_0 \Rightarrow \text{true}] = \Pr[A \diamond L_1 \Rightarrow \text{true}]$

- $A$ being simultaneously linked to both libraries

- Two different executions:
  - one where $A$ is linked only to $L_0$ for its entire lifetime, and
  - one where $A$ is linked only to $L_1$ for its entire lifetime

- "I can’t choose what to enc, I have to ask $A$ to choose"

- "I am safe to encrypt things even if the attacker sees the resulting ciphertext and even if she has some influence or partial information on what I’m encrypting"
I write down the source code of two libraries which are “designed” based on the known crypto. algorithms.

Attacker’s goal: write an effective distinguisher $A$

Kerckhoffs’ Principle: $A$ knows every fact in the universe, except:

1. which of the two possible libraries it is linked to
2. the random choices that the library will make
   - e.g., $A$ is linked to a library that executes the “$k \leftarrow \{0,1\}^\lambda$”
   - but $A$ doesn’t know the value of $k$ chosen at runtime during any particular execution.
Just for illustration, Dec() is not defined.

- Core observation of insecurity:
  It can never encrypt a bit 0 into a bit 1.

- Uniform ciphertext: how to choose $m$ to make $c$ non-uniform?

- Left-or-Right secrecy: how to choose two “differentiating” $m$’s?
Demonstrating Insecurity with Attacks

---

**Left-hand side is algorithm \( \mathcal{A} \) we designed for attack.**

- Right-hand side inserts the insecure algorithm into two libraries (or “templates”).
- The choice \( m \) of \( \mathcal{A} \) makes:
  - \( \Pr[\mathcal{A} \triangle L_{\text{ots$\$$-real}} \Rightarrow \text{true}] = 1. \)
  - \( \Pr[\mathcal{A} \triangle L_{\text{ots$\$$-rand}} \Rightarrow \text{true}] = 2^{-\lambda}. \)

\[\mathcal{A}: \]
\[
c := \text{ctxt}(\theta^\lambda) \\
\text{return } c = \theta^\lambda
\]

\[\mathcal{L}^\Sigma_{\text{ots$\$$-real}} \]
\[
\text{ctxt}(m): \\
k \leftarrow \{0, 1\}^\lambda \\
c := k \& m \\
\text{return } c
\]

\[\mathcal{A}: \]
\[
c := \text{ctxt}(\theta^\lambda) \\
\text{return } c = \theta^\lambda
\]

\[\mathcal{L}^\Sigma_{\text{ots$\$$-rand}} \]
\[
\text{ctxt}(m): \\
c \leftarrow \{0, 1\}^\lambda \\
\text{return } c
\]
Breaking Left-or-Right Security

- Left-hand side is algorithm $\mathcal{A}$ we designed for attack.
- Right-hand side inserts the insecure algorithm into two libraries (or "templates").
- The choices of $\mathcal{A}$ make:
  - $\Pr[\mathcal{A} \land L_{ots\_L} \Rightarrow true] = 1$.
  - $\Pr[\mathcal{A} \land L_{ots\_R} \Rightarrow true] = 2^{-\lambda}$.
Hybrid Technique for proving security

- Week 1 proved that OTP satisfies the uniform ciphertexts property by carefully calculating certain probabilities.
  - Not a sustainable way to always do a “first principle” proof for more complicated schemes

- Lemmas to consider the “hybrid” libraries:
  1. \((A \diamond L_1) \diamond L_2 \equiv A \diamond (L_1 \diamond L_2)\), or “Associativity” of \(\diamond\)
  2. If \(L_{\text{left}} \equiv L_{\text{right}}\), for any library \(L^*\), we have \(L^* \diamond L_{\text{left}} \equiv L^* \diamond L_{\text{right}}\)
Associativity of $\Diamond$

- $(\mathcal{A} \Diamond L_1) \Diamond L_2 \equiv \mathcal{A} \Diamond (L_1 \Diamond L_2)$
- A compound calling program $\mathcal{A} \Diamond L_1$ linked to $L_2$ vs. $\mathcal{A}$ linked to a compound library $L_1 \Diamond L_2$

- “Proof”: Splitting up a program into different source files doesn’t affect its functionality.
If $L_{\text{left}} \equiv L_{\text{right}}$, then $L^* \diamond L_{\text{left}} \equiv L^* \diamond L_{\text{right}}$

- We have two combined libraries: $L^* \diamond L_{\text{left}}$ and $L^* \diamond L_{\text{right}}$
- Consider an arbitrary calling program $A$, we want output distribution of $A \diamond (L^* \diamond L_{\text{left}})$ and $A \diamond (L^* \diamond L_{\text{right}})$ are the same
- We apply associativity to “change the perspective”
- $\Pr[A \diamond (L^* \diamond L_{\text{left}}) \Rightarrow true]$
- $= \Pr[(A \diamond L^*) \diamond L_{\text{left}} \Rightarrow true]$ // associativity
- $= \Pr[(A \diamond L^*) \diamond L_{\text{right}} \Rightarrow true]$ // our if cond.: $L_{\text{left}} \equiv L_{\text{right}}$
- $= \Pr[A \diamond (L^* \diamond L_{\text{right}}) \Rightarrow true]$ // associativity
Standard Hybrid Technique

Proving security is now made like “mathematical proof”

- You want to prove \( L_0 = R_0 \).
- \( L_0 \) and \( R_0 \) look “significantly” different! 😱
- You start with \( L_0 \).
- Make some small modifications.
- They remain equal.
- You end up with \( R_0 \).

- You want to prove \( L_0 \equiv L_1 \).
- You start with \( L_0 \).
- Make a sequence of small modifications.
- Each modification has no effect on calling program / adversary.
  - e.g., slide #16
- Sequence of modifications ends with \( L_1 \).
Now let’s do it!

- **2-otp-proof.pdf** for proving OTP’s left-or-right secrecy
  - The argument there uses the fact that OTP has uniform ciphertext.

- Your “homework”:
  - Reading 1: proof in textbook for “double OTP”
  - Reading 2: example in textbook to show what goes wrong if “blindly” applying the same proof over an insecure scheme
    - Usually the boundary between secure and insecure is razor thin
    - The proof cannot go through => The “source” of insecurity
Compare/Contrast Security Definitions

- Uniform ciphertext $\Rightarrow$ Left-or-right secrecy
  - 2-otp-proof.pdf, the same argument also goes through for any one-time LoR-secure encryption
- Left-or-right secrecy $\nRightarrow$ Uniform ciphertext
  - You just need a counterexample, but we only saw 1 one-time LoR-secure encryption scheme: OTP, which has uniform ctxt.
  - Then make a “contrived example” that remains LoR-secure.
  - OTP’ where $\text{OTP'}.Enc() := \text{OTP'}.Enc() \ || 0$
    - i.e., $\text{OTP'}.C = \{0, 1\}^{\lambda+1}$, if $\text{OTP}.C = \{0, 1\}^\lambda$
    - Yet, if you forcibly define $\text{OTP'}.C$ to be $\{x \ || \ 0, x \in \text{OTP}.C\}$, it is then “secure”
We just see an edge case in the security definition itself!
- Ciphertext space should have no effect on the functionality?
- but “functionality” in the above statement means decryption correctness, what if someone use the ctxt. for “other purpose”?

A definition cannot really be “wrong”
- unless it is self contradictory, not well defined, etc.

It is useless to prove a scheme being secure under a “useless” definition.
- e.g., A scheme X is provably secure under a “too-weak-to-model-the-real-world” model