Verifying cryptographic implementations with Jasmin & EasyCrypt

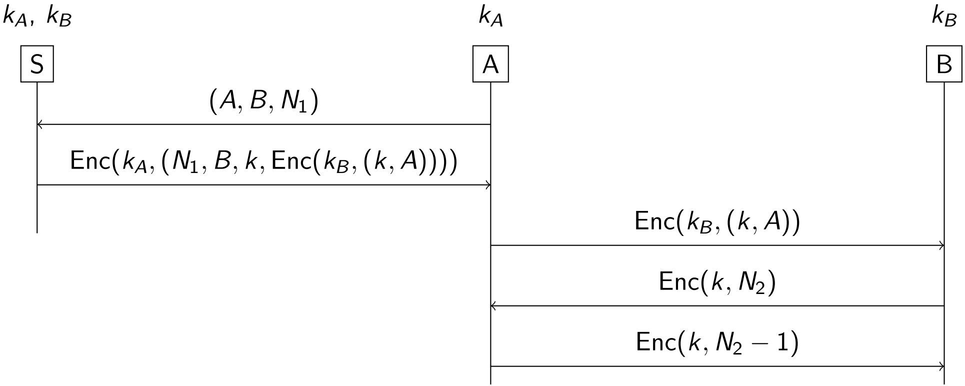
Vincent Laporte; Benjamin Grégoire
Manuel Barbosa; François Dupressoir; Pierre-Yves Strub; Tiago Oliveira

2023-09-12, ST&V 2023, Brussels

Outline

High-Assurance Implementation of Cryptography

Security Protocols



Needham–Schroeder Symmetric Key Protocol

Ensure security properties (e.g., mutual authentication)

Rely on secure primitives (e.g., a symmetric encryption scheme)

Cryptographic Primitives

Example: a nonce-based encryption scheme

- k : key
- n : nonce
- m, m' : plain-text messages
- c : ciphertext

c := Enc(k, n, m)
m' := Dec(k, n, c)

Correctness of an encryption scheme

Knowing the secret key allows to recover the plaintext:

$$Dec(k, n, Enc(k, n, m)) = m$$

Cryptographic security (IND$-CPA)

A ciphertext is indistinguishable from random:

$\underline{\text{Game IND\$-CPA-Real}_{\mathcal{A}}(\,)}$
$k \twoheadleftarrow K$
$b \twoheadleftarrow \mathcal{A}^{\text{RealEnc}(\cdot,\cdot)}(\,)$
Return b

$\underline{\text{proc RealEnc}(n, m)}$
Return $\text{Enc}(k, n, m)$

$\underline{\text{Game IND\$-CPA-Ideal}_{\mathcal{A}}(\,)}$
$b \twoheadleftarrow \mathcal{A}^{\text{IdealEnc}(\cdot,\cdot)}(\,)$
Return b

$\underline{\text{proc IdealEnc}(n, m)}$
$c \twoheadleftarrow C$
Return c

Security requires the following advantage measure to be small

$$|\Pr[\text{IND\$-CPA-Real}_{\mathcal{A}}(\,) \Rightarrow \text{true}] - \Pr[\text{IND\$-CPA-Ideal}_{\mathcal{A}}(\,) \Rightarrow \text{true}]|$$

Implementation security

Adversaries may observe the machine running a victim program.

Is any sensitive information leaked into these observations?

Constant-Time

- A popular mitigation against timing (cache-based) side-channel attacks
- Two rules
  - No branching on secret data
  - No memory access at secret addresses

Cryptographic security in spite of side-channels

We can generalize the IND\$-CPA definition

Security still holds for constant-time programs

Other Implementation-Level Requirements

Efficiency

- CPU cycles matter
- This can be assessed experimentally (through measurements)

Safety

Running the program:

- terminates
- does not crash (division by zero. . .)
- does not access arrays out of bounds, uninitialized variables

Programs are usually not safe. Only under some **precondition**.

Correctness

The program actually fulfills its specification.

Formosa Crypto

The Formosa Crypto project federates multiple projects in machine-checked cryptography and high-assurance cryptographic engineering under a single banner, to better support developers and users.

EasyCrypt	Construction and verification of game-based cryptographic proofs
Jasmin	Programming language for high-speed secure implementations
LibJade	High-assurance software implementations of post-quantum crypto

https://formosa-crypto.org/

Secure High-Assurance Implementations of SHA-3 [CCS2019]

- Fast (optimized for AVX2)
- Secure (constant-time)
- Correct (wrt. a reference implementation)

Indifferentiability proof of the Sponge construction

- Main theorem about security of SHA-3
- Bounds the probability for an adversary to break it:
  - in particular to find collisions, preimages, or second preimages
- Theorem applies to the optimized implementation!

Formally Verifying Kyber – Episode IV [TCHES2023]

- Reference & optimized implementations
- Correctness proof
- Theoretical results about the random sampling procedures

Work in progress

- Security proof
- Verification of the fully optimized implementation
- Integrate with the existing proofs of SHA-3

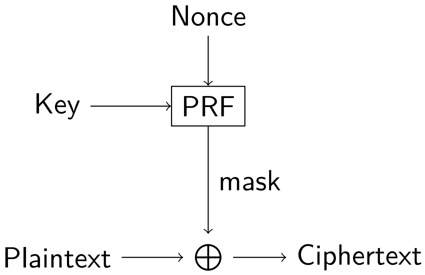
Beyond Crypto Implementations

Jasmin is also a nice tool for research on (verified) (secure) compilation:

- use machine learning to search for faster implementations;
- study counter-measures against speculative execution attacks (Spectre);
- enforce zeroing of local memory after use;
- . . .

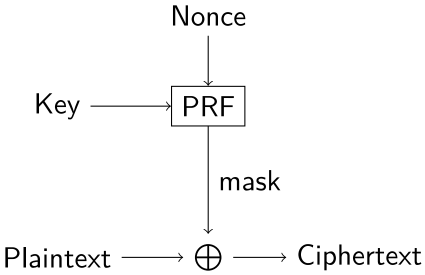
The Jasmin tool-box

Example: symmetric encryption from a PRF



$$\mathsf{Enc}(k, n, m) = m \oplus f(k, n)$$
$$\mathsf{Dec}(k, n, c) = c \oplus f(k, n)$$

Example: symmetric encryption from a PRF



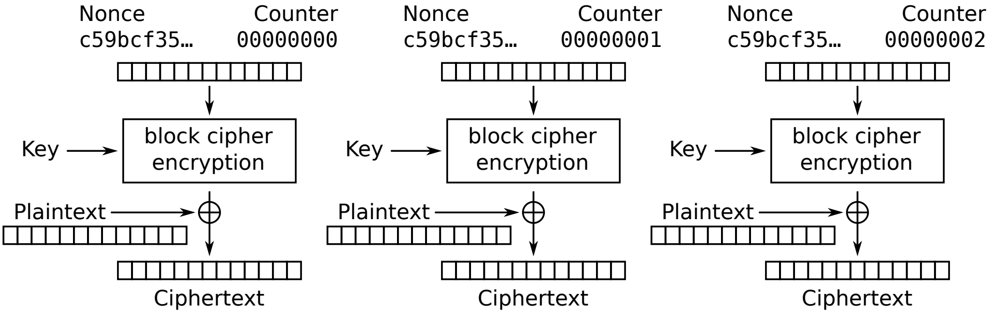
$$\mathsf{Enc}(k, n, m) = m \oplus f(k, n)$$
$$\mathsf{Dec}(k, n, c) = c \oplus f(k, n)$$

Specific choice of PRF: AES-128

Key, nonce, mask, plaintext, and ciphertext are 128-bit values.

Counter mode of operation

There are a few common (and secure) ways to turn a block cipher into a stream cipher, e.g.:



Counter (CTR) mode encryption

Example Encryption Scheme in Jasmin

Look at nbaesenc.jazz

Executable semantics

In program p, calling function f with arguments $\vec{a}$ from initial memory m terminates in final memory m' and returns values $\vec{r}$:

$$f : (\vec{a}, m) \Downarrow_p (\vec{r}, m')$$

This is the definition of the program *behaviors* (formalized in Coq).

All proofs are made relative to this definition.

We gain *trust* by using it (execute & verify programs, verify static analyses, verify program transformations, . . .)

A compiler to assembly

- Produces (predictable) assembly for x86_64
- Experimental support for ARMv7 architecture (Cortex-M4)
- Complies with standard ABI
  - for interoperability with other languages
  - look at bindings/

Safety: jasminc -checksafety ...

- Programs are usually **not** safe
  - Restrictions on the initial state
- Returns a sufficient pre-condition for safety (a predicate)
- Overapproximation because of undecidability
- The design of the programming language encourages the use high-level features that make safety verification doable automatically

When the safety checker infers precondition P (for a function f in program p), then for all initial state satisfying this precondition, there exists a corresponding final state:

$$\forall \vec{a}\ m, P(\vec{a}, m) \implies \exists \vec{r}\ m', f : (\vec{a}, m) \Downarrow_p (\vec{r}, m')$$

No formal proof of this property.

Constant-time: jasminc -checkCT ...

Function signatures can be decorated with #public and #secret annotations.

An automated source-level checker validates that no sensitive values flow to:

- branching conditions (including loop guards)
- array indices and dereferenced pointers

Approximations are unavoidable

- Memory contents are assumed to be #secret
- When needed, assignments can be annotated with #declassify to claim (admit) they produce public values (e.g., at the end of encryption)

EasyCrypt models: jasminc -oec ...

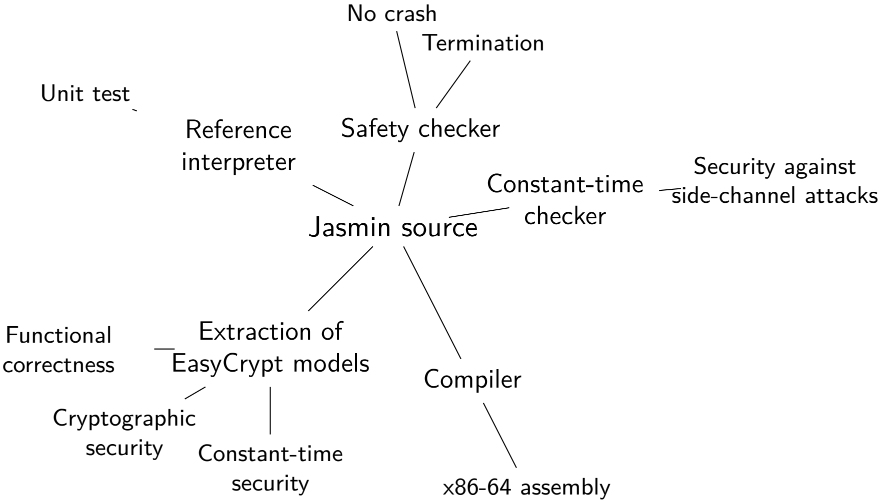
Reasoning about semantics of source programs is better done in a dedicated proof assistant.

Extract an EasyCrypt model from a Jasmin source program.

For safe inputs to the Jasmin program, the EasyCrypt program computes the same outputs (as the Jasmin program).

No formal proof of this statement.

The Jasmin tool-box

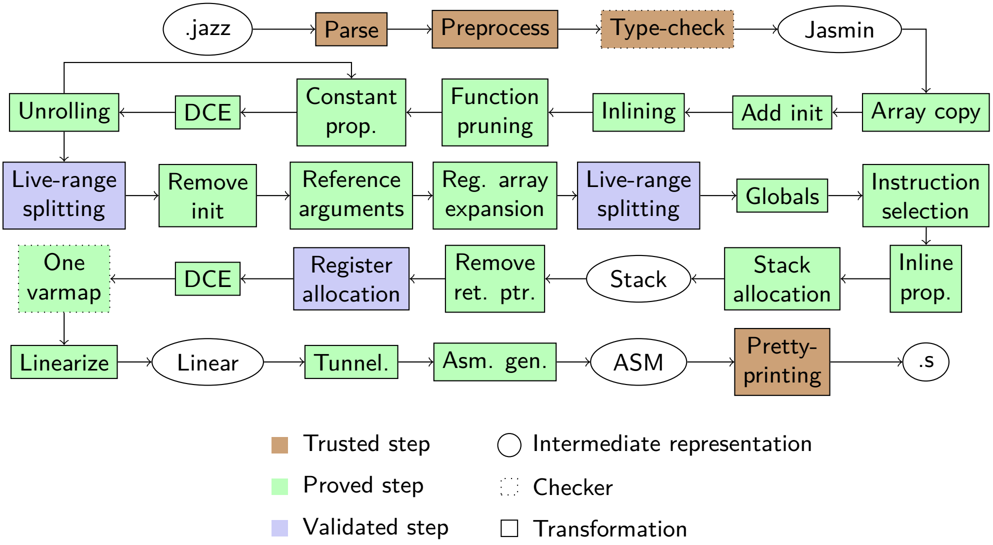


Verified Compilation

What we run is not what has been verified

- Source vs. assembly
- Can we trust the compiler?

Program transformations in the Jasmin compiler



Compiler correctness

Forward simulation

If the compilation of source program S succeeds and produces target program T,
if from the initial state i, S terminates with final result r,
then from the same initial state i, T also terminates with final result r.

Overlooked details

- Initial states may not be the same
  - Global data must be in the target memory
  - The "stack pointer" (RSP) must point to a valid region of memory
- The target stack must be large enough
  - i.e., the compiler does not enforce the absence of "stack overflow"

Note about safety

No guaranties about unsafe executions

Preservation of functional properties

Compiler correctness implies

If a property holds for all source behaviors,
then it holds for all target behaviors.

When the source program is a *function* (deterministic, terminating)

Then the target program is the *same* function.

- Cryptographic primitives are usually functions
- even PRNGs!

Probabilities

When a function consumes random data

Reasoning about the *distribution* of the results in terms of the distribution of the inputs can be done at the source level.

Probabilistic properties of functions are preserved (example: IND$-CPA)

Given a secret key, an adversary cannot distinguish (with non-negligible probability) the encryption function from random sampling
This property is independent of the implementation.

- Unless the adversary has access to non-functional properties of the implementation

Non-preservation

Non-deterministic programs

A *correct* compiler may not preserve distributions.
For instance, a source program that tosses a coin may be correctly compiled to the constant program that always returns *heads*.

Changing the representation of values

E.g., booleans implemented as 63-bit machine integers.

$$S : b \mapsto \neg b \qquad\qquad T : n \mapsto 1 - n$$

How to map *invalid* target values to source values?
There is no way to express at the source level the target behavior.

Non-functional properties

The theorem does not say anything about things that cannot be described by *behaviors*.

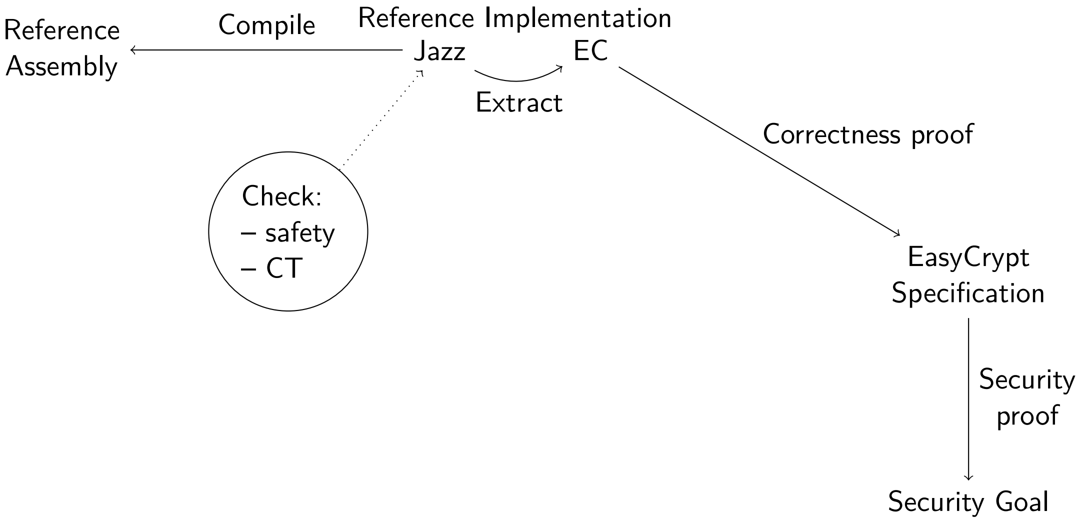
Constant-time preservation

The compiler (always) preserves the constant-time property.

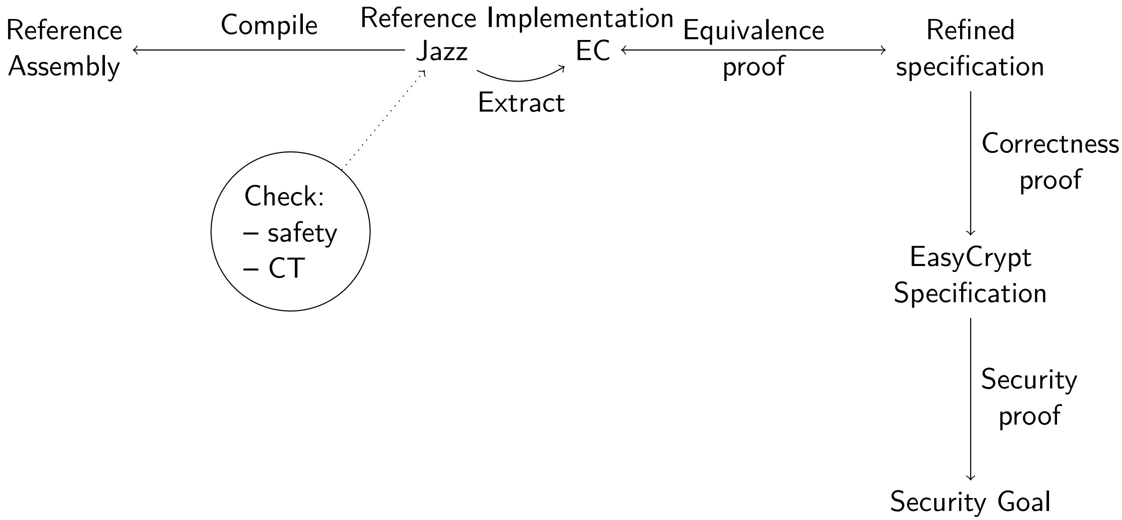
Formal (machine-checked) proof of this statement for version 21.0 of the compiler [CCS21].

This is a stronger property than compiler-correctness.

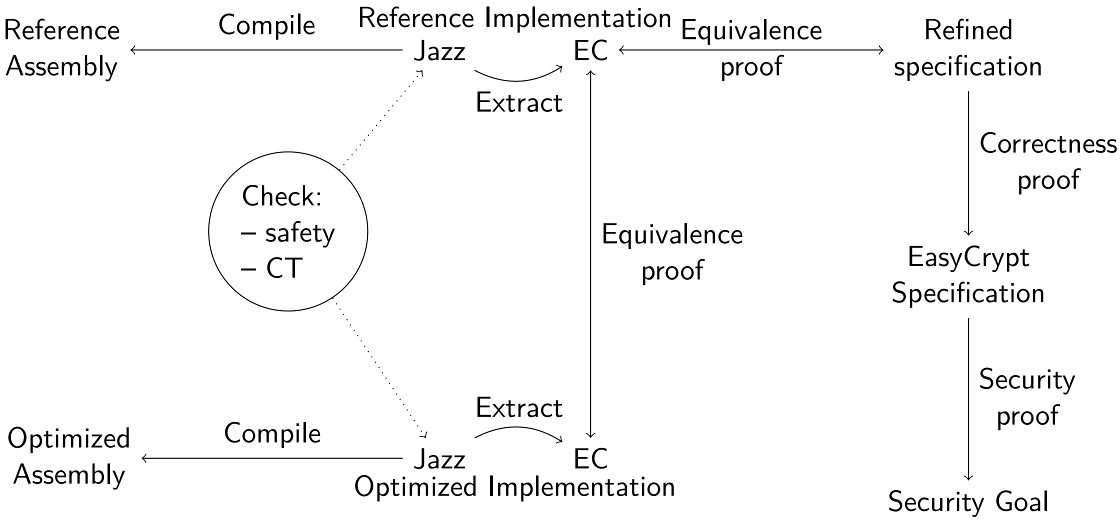
Verification of Jasmin Programs



Verification of Jasmin Programs



Verification of Jasmin Programs



Some Unique Features of the Jasmin Language

Overview of the Jasmin programming language

Look at aes.jinc.

High-level structure, with low-level control

A few data-types

- int; bool; machine words u8, . . . , u256; arrays of words, e.g., u128[11]
- convenient syntax for vector (SIMD) instructions

Functions

- inline fn or #inline calls
- return address can be passed in a register or on the stack

Structured control flow

- usual if-then-else (no goto)
- two kinds of loops:
  - for loops: unrolled
  - while loops: preserved

Low-level programming

- Explicit storage class
  - param, inline: compile-time use only
  - global, stack: memory
  - reg: registers
- Direct access to target instructions & flag registers
  - jasminc -help-intrinsics to get the list
  - Flags are plain variables

Common uses of intrinsics & flags

- Initialize to zero using a XOR: #set0
- Branch on the result of an arithmetic operation
- A single comparison with more than two outcomes

See src/low-level.jazz

Quizz (see src/array.jazz)

Can we tell something about the first returned value?

```
// Defines fn f(reg u8 x y) ⟶ reg u8
require "array.jinc"

inline
fn quizz0(reg u8 x) ⟶ reg u8, reg u8 {
    reg u8 r, y;

    r = 0;
    y = f(r, x);
    return r, y;
}
```

Quizz (see src/array.jazz)

Can we tell something about the first returned value?

```
// Defines fn f(reg u8 x y) ⟶ reg u8
require "array.jinc"

inline
fn quizz0(reg u8 x) ⟶ reg u8, reg u8 {
    reg u8 r, y;

    r = 0;
    y = f(r, x);
    return r, y;
}
```

```
// Defines fn g(stack u8[1] x, reg u8 y) ⟶ stack u8[1]
require "array.jinc"

inline
fn quizz1(reg u8 x) ⟶ stack u8[1], stack u8[1] {
    stack u8[1] r, y;

    r[0] = 0;
    y = g(r, x);
    return r, y;
}
```

Arrays: an explicit and powerful way to structure memory

Things made easier

- Modular reasoning is possible
- Sizes are explicit
  - Useful for proving safety
- Alias analysis is trivial
  - Arrays may overlap only when they have the same name

Caveat

Ensuring call-by-value semantics without copy is tricky (the compiler rejects programs)

Verification with EasyCrypt

Verification overview

1. Start from a Jasmin implementation
2. Extract from it an EasyCrypt model
3. Prove it equivalent to a hand-written refined (detailed) specification
4. Show it refines a higher-level specification
5. Prove security of the specification

Formal Verification with EasyCrypt: correctness

Agenda

1. Specify a correct nonce-based encryption scheme (proof/NbEnc.eca)
2. Specify the construction with a PRF, and prove it correct (proof/NbPRFEnc.eca)
3. Refine the construction with AES as PRF and prove it equivalent to the Jasmin implementation (proof/NbAESEnc_proof.ec)

Scheme correctness

$\underline{\text{Game Correctness}_{\text{Scheme}}(k, n, m)}$

$c \twoheadleftarrow \text{Scheme.enc}(k, n, m)$
$m' \twoheadleftarrow \text{Scheme.dec}(k, n, c)$
Return $m' = m$

The nonce-based symmetry encryption scheme Scheme is correct when, for all key k, nonce n and plaintext message m, the probability for this game to return true is one:

$$\Pr[\,\text{Correctness}_{\text{Scheme}}(k, n, m) \Rightarrow \text{true}\,] = 1.$$

Functional correctness

The Jasmin implementation (modeled by the pWhile procedure $\mathrm{M.enc}$) is equivalent to the refined specification $\mathrm{Scheme.enc}$, as expressed in pRHL:

$$\{=_{\{k,n,m\}}\}\,\mathsf{M.enc} \sim \mathsf{Scheme.enc}\,\{=_{\{\mathsf{res}\}}\}$$

Formal Verification with EasyCrypt: security

Agenda

1. Specify the intended security goal (proof/NbEnc.eca)
2. State the cryptographic assumption (proof/RFth.eca)
3. Prove security of the generic construction (proof/NbPRFEnc.eca)

(Nonce-based) IND\$-CPA security

$\underline{\text{Game IND\$-CPA-Real}_{\mathcal{A}}(\,)}$
$k \twoheadleftarrow K$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{RealEnc}(\cdot,\cdot)}(\,)$
Return b

$\underline{\text{proc RealEnc}(n, m)}$
Return $\mathrm{Enc}(k, n, m)$

$\underline{\text{Game IND\$-CPA-Ideal}_{\mathcal{A}}(\,)}$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{IdealEnc}(\cdot,\cdot)}(\,)$
Return b

$\underline{\text{proc IdealEnc}(n, m)}$
$c \twoheadleftarrow C$
Return c

Security requires the following advantage measure to be small

$$\left|\Pr\left[\,\text{IND\$-CPA-Real}_{\mathcal{A}}(\,) \Rightarrow \mathsf{true}\,\right] - \Pr\left[\,\text{IND\$-CPA-Ideal}_{\mathcal{A}}(\,) \Rightarrow \mathsf{true}\,\right]\right|$$

Restrictions on attacker power

If we don't restrict class of attackers:

- always one attacker with large advantage.

Restrictions on attacker power

If we don't restrict class of attackers:

- always one attacker with large advantage.

Restrictions that come up explicitly in EasyCrypt:

- Do *not* place two queries with the same nonce n
- Place at most q oracle queries (RP/RF switch in exercise)

Restrictions on attacker power

If we don't restrict class of attackers:

- always one attacker with large advantage.

Restrictions that come up explicitly in EasyCrypt:

- Do *not* place two queries with the same nonce n
- Place at most q oracle queries (RP/RF switch in exercise)

Restrictions on attacker power that will be implicit:

- IND\$-CPA attacker executes in at most t steps
- we assume that PRF/PRP cannot be broken in $\sim t$ steps

Pseudorandom Functions

Let f be a function of type $f : \{0,1\}^\lambda \times \{0,1\}^\kappa \to \{0,1\}^\ell$.

$$\begin{array}{ll}
\underline{\text{Game PRF-Real}_{\mathcal{A}}(\,)} & \underline{\text{Game PRF-Ideal}_{\mathcal{A}}(\,)} \\
k \twoheadleftarrow \{0,1\}^\lambda & T \leftarrow \{\,\} \\
b \twoheadleftarrow \mathcal{A}^{\mathsf{f}(k,\cdot)}(\,) & b \twoheadleftarrow \mathcal{A}^{\mathsf{F}(\cdot)}(\,) \\
\text{Return } b & \text{Return } b \\
 & \\
 & \underline{\text{proc F}(x)\text{:}} \\
 & \text{If } x \notin T\text{: } T[x] \twoheadleftarrow \{0,1\}^\ell \\
 & \text{Return } T[x]
\end{array}$$

F is a truly random function (lazily sampled).

f is pseudorandom if the following advantage measure is small

$$|\Pr[\text{PRF-Real}_{\mathcal{A}}(\,) \Rightarrow \text{true}] - \Pr[\text{PRF-Ideal}_{\mathcal{A}}(\,) \Rightarrow \text{true}]|$$

Security proof: Step #1

Standard game hop: modify IND\$-CPA-Real game.

Game IND\$-CPA-Real$_{\mathcal{A}}()$	Game IND\$-CPA-Modified$_{\mathcal{A}}()$
$k \twoheadleftarrow K$	$T \leftarrow \{\}$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{RealEnc}(\cdot,\cdot)}()$	$b \twoheadleftarrow \mathcal{A}^{\mathsf{ModifiedEnc}(\cdot,\cdot)}()$
Return b	Return b

proc RealEnc(n, m)	proc ModifiedEnc(n, m)
Return $m \oplus \mathsf{f}(k, n)$	If $n \notin T$: $T[n] \twoheadleftarrow \{0,1\}^{\ell}$
	Return $m \oplus T(n)$

We replaced $\mathsf{f}(k, \cdot)$ with a truly random function (lazily sampled).

Security proof: Step #2

If $\mathcal{A}$ notices the change we break f as a PRF.

Attacker $\mathcal{B}$ against the PRF property of f:

- Runs $\mathcal{A}$ and answers encryption queries (n, m):
  - calls its own oracle on n to get mask
  - returns $m \oplus$ mask to $\mathcal{A}$
- When $\mathcal{A}$ terminates $\mathcal{B}$ uses output as its own.

Security proof: Step #2

If $\mathcal{A}$ notices the change we break f as a PRF.

Attacker $\mathcal{B}$ against the PRF property of f:

- Runs $\mathcal{A}$ and answers encryption queries (n, m):
  - calls its own oracle on n to get mask
  - returns $m \oplus$ mask to $\mathcal{A}$
- When $\mathcal{A}$ terminates $\mathcal{B}$ uses output as its own.

Observations:

- If $\mathcal{B}(\mathcal{A})$ is run in the PRF-Real game:
  - Output matches $\mathcal{A}$'s output in IND\$-CPA-Real
- If $\mathcal{B}(\mathcal{A})$ is run in the PRF-Ideal game:
  - Output matches to $\mathcal{A}$'s output in IND\$-CPA-Modified

Security proof: Step #3

$\mathcal{A}$'s view in modified game matches the IND\$-CPA ideal game.

$\underline{\text{Game IND\$-CPA-Modified}_{\mathcal{A}}(\,)}$
$T \leftarrow \{\,\}$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{ModifiedEnc}(\cdot,\cdot)}(\,)$
Return b

$\underline{\text{Game IND\$-CPA-Ideal}_{\mathcal{A}}(\,)}$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{IdealEnc}(\cdot,\cdot)}(\,)$
Return b

$\underline{\text{proc ModifiedEnc}(n, m)}$
If $n \notin T$: $T[n] \twoheadleftarrow \{0,1\}^{\ell}$
Return $m \oplus T(n)$

$\underline{\text{proc IdealEnc}(n, m)}$
$c \twoheadleftarrow C$
Return c

Nonce-respecting adversary:

- T values always fresh random strings.
- XOR operation produces totally random string (OTP).
- Oracle outputs are identically distributed in both games.
- $\mathcal{A}$'s output is identically distributed in both games.

Security proof: Step #4

Wrapping up:

$$\Pr[\text{IND\$-CPA-Real}_{\mathcal{A}}() \Rightarrow \text{true}] = \Pr\left[\text{PRF-Real}_{\mathcal{B}(\mathcal{A})}() \Rightarrow \text{true}\right]$$

$$\Pr[\text{IND\$-CPA-Modified}_{\mathcal{A}}() \Rightarrow \text{true}] = \Pr\left[\text{PRF-Ideal}_{\mathcal{B}(\mathcal{A})}() \Rightarrow \text{true}\right]$$

$$\Pr[\text{IND\$-CPA-Modified}_{\mathcal{A}}() \Rightarrow \text{true}] = \Pr[\text{IND\$-CPA-Ideal}_{\mathcal{A}}() \Rightarrow \text{true}]$$

Implies $\mathcal{A}$'s advantage is exactly that of $\mathcal{B}(\mathcal{A})$:

- substitute last equation in middle equation
- subtract middle equation from first

$\mathcal{B}(\mathcal{A})$ is as efficient as $\mathcal{A}$ and makes same number of queries.