EasyCrypt and Jasmin Tutorial

VeriCrypt @ Indocrypt 2020

Manuel Barbosa (mbb@fc.up.pt)
Benjamin Grégoire (benjamin.gregoire@inria.fr)
François Dupressoir (f.dupressoir@bristol.ac.uk)
Vincent Laporte (vincent.laporte@inria.fr)
Pierre-Yves Strub (pierre-yves@strub.nu)

December 11th 2020

What to expect

What we will cover (using a very simple example)

The Jasmin language and compiler:

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs
- Extract Jasmin to EasyCrypt for constant-time verification

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs
- Extract Jasmin to EasyCrypt for constant-time verification

The EasyCrypt proof assistant:

- Specify syntax and security models for crypto protocols

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs
- Extract Jasmin to EasyCrypt for constant-time verification

The EasyCrypt proof assistant:

- Specify syntax and security models for crypto protocols
- Specify crypto assumptions and concrete crypto protocols

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs
- Extract Jasmin to EasyCrypt for constant-time verification

The EasyCrypt proof assistant:

- Specify syntax and security models for crypto protocols
- Specify crypto assumptions and concrete crypto protocols
- Prove crypto protocols correct and secure

What we will cover (using a very simple example)

The Jasmin language and compiler:

- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs
- Extract Jasmin to EasyCrypt for constant-time verification

The EasyCrypt proof assistant:

- Specify syntax and security models for crypto protocols
- Specify crypto assumptions and concrete crypto protocols
- Prove crypto protocols correct and secure
- Prove functional correctness and security of Jasmin programs

What we will cover (using a very simple example)

The Jasmin language and compiler:

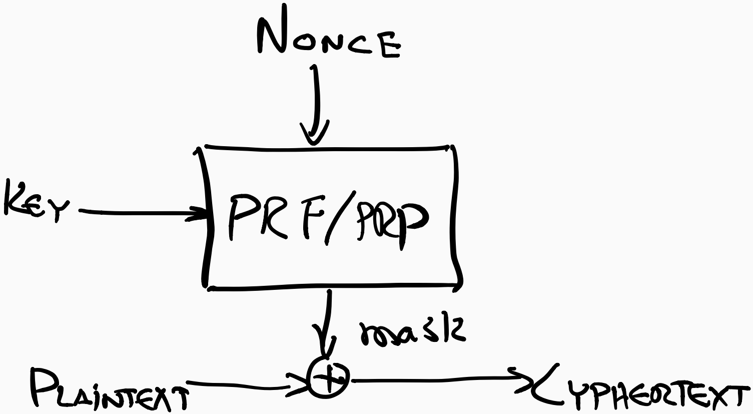
- Write high-speed crypto code and compile it to assembly
- Automatically safety-check Jasmin programs
- Extract Jasmin to EasyCrypt for correctness/security proofs
- Extract Jasmin to EasyCrypt for constant-time verification

The EasyCrypt proof assistant:

- Specify syntax and security models for crypto protocols
- Specify crypto assumptions and concrete crypto protocols
- Prove crypto protocols correct and secure
- Prove functional correctness and security of Jasmin programs
- Prove Jasmin programs constant-time secure

The example

The example: textbook symmetric encryption from PRF/PRP



The example: implementation view

How to implement PRF/PRP?

- We will use AES-NI: hardware support
- Processor instructions give:
  - Implementation of AES round
  - Implementation of AES round-key computation
  - Assistance in preparing key for computation of round keys
  - All instructions operate over 128-bit registers
- Pre-implemented Jasmin function provides AES
  - code can be found in `src/aeslib`
  - note key schedule recomputed in each call (can be optimized)

AES in Jasmin

```
inline fn aes(reg u128 key, reg u128 in) → reg u128 {
  reg u128 out;
  reg u128[11] rkeys;

  rkeys = keys_expand(key);
  out   = aes_rounds(rkeys, in);
  return out;
}

inline fn invaes(reg u128 key, reg u128 in) → reg u128 {
  reg u128 out;
  reg u128[11] rkeys;

  rkeys = keys_expand_inv(key);
  out   = invaes_rounds(rkeys, in);
  return out;
}
```

The example encryption scheme in Jasmin

```
inline fn xor(reg u128 a, reg u128 b) → reg u128 {
    reg u128 r;
    r = a^b;
    return r;
}

export fn enc(reg u128 k, reg u128 n, reg u128 p) → reg u128 {
    reg u128 mask,c;
    mask = aes(k,n);
    c = xor(mask,p);
    return(c);
}

export fn dec(reg u128 k, reg u128 n, reg u128 c) → reg u128 {
    reg u128 mask,p;
    mask = aes(k,n);
    p = xor(mask,c);
    return(p);
}
```

A version using memory (proofs more technical)

```
export fn enc(reg u64 cptr, reg u64 kptr, reg u64 nptr, reg u64 pptr) {
    reg u128 mask,k,n,p,c;
    k = (u128)[kptr];
    n = (u128)[nptr];
    mask = aes(k,n);
    p = (u128)[pptr];
    c = xor(mask,p);
    (u128)[cptr] = c;
}

export fn dec(reg u64 pptr, reg u64 kptr, reg u64 nptr, reg u64 cptr) {
    reg u128 mask,k,n,p,c;
    k = (u128)[kptr];
    n = (u128)[nptr];
    mask = aes(k,n);
    c = (u128)[cptr];
    p = xor(mask,c);
    (u128)[pptr] = p;
}
```

The example: provable security view

The construction in crypto terms

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

Fix:

- the key space $K := \{0,1\}^\lambda$
- the nonce space $N := \{0,1\}^\kappa$
- the message space $M := \{0,1\}^\ell$
- the ciphertext space $C := M$

Key generation: sampling uniformly at random from K

Encryption: $\mathsf{Enc}(k, n, m) := m \oplus \mathsf{f}(k, n)$

Decryption: $\mathsf{Dec}(k, n, c) := c \oplus \mathsf{f}(k, n)$

(Nonce-based) IND$-CPA security

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

proc RealEnc$(n, m)$
Return Enc$(k, n, m)$
```

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

proc IdealEnc$(n, m)$
$c \twoheadleftarrow C$
Return $c$
```

Security requires the following advantage measure to be small

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

Pseudorandom Functions

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

```
Game PRF-Real$_\mathcal{A}$( )
$k \twoheadleftarrow \{0,1\}^\lambda$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{f}(k,\cdot)}()$
Return $b$
```

```
Game PRF-Ideal$_\mathcal{A}$( )
$T \leftarrow \{\,\}$
$b \twoheadleftarrow \mathcal{A}^{\mathsf{F}(\cdot)}()$
Return $b$
```

```
proc F($x$):
If $x \notin T$:  $T[x] \twoheadleftarrow \{0,1\}^\ell$
Return $T[x]$
```

F is a truly random function (lazily sampled).

f is pseudorandom if the following advantage measure is small

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

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

Security proof: Step #1

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

```
Game IND$-CPA-Real_A()
    $k \twoheadleftarrow K$
    $b \twoheadleftarrow \mathcal{A}^{\mathsf{RealEnc}(\cdot,\cdot)}()$
    Return $b$

proc RealEnc($n, m$)
    Return $m \oplus \mathsf{f}(k, n)$
```

```
Game IND$-CPA-Modified_A()
    $T \leftarrow \{\}$
    $b \twoheadleftarrow \mathcal{A}^{\mathsf{ModifiedEnc}(\cdot,\cdot)}()$
    Return $b$

proc ModifiedEnc($n, m$)
    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.

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

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

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

```
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:

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

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

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

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.

The example: verification view

What do we want to prove?

Obvious to everyone: implementation is *secure*!

What do we want to prove?

Obvious to everyone: implementation is *secure*!

Obvious to crypto practitioners: implementation is *constant-time*.

What do we want to prove?

Obvious to everyone: implementation is *secure*!

Obvious to crypto practitioners: implementation is *constant-time*.

What does it mean for the implementation to be *secure*?

What do we want to prove?

Obvious to everyone: implementation is *secure*!

Obvious to crypto practitioners: implementation is *constant-time*.

What does it mean for the implementation to be *secure*?

- code *is* a secure encryption scheme?
- code *implements* a specific secure encryption scheme?
- are these the same question?
- are these both true?

The standardisation perspective

We have three artifacts:

- crypto-style specification and proof of security (e.g.,paper)
- technical specification (e.g., a standard)
- low-level optimized implementation (e.g., assembly)

The standardisation perspective

We have three artifacts:

- crypto-style specification and proof of security (e.g.,paper)
- technical specification (e.g., a standard)
- low-level optimized implementation (e.g., assembly)

Natural questions:

- Is the technical specification secure in the crypto sense?
- Is the implementation correct wrt the technical spec?
- If both true, is the implementation secure in the crypto sense?
- How does constant-time fit into this picture?

Verification goals [FSE'16]

The following machine-checked proofs for Enc and Dec:

- Technical specification is provably IND$-CPA secure
  - specify standard algorithms in EasyCrypt
  - specify IND$-CPA game in EasyCrypt
  - prove specification secure assuming AES is a PRF or PRP
  - consider details a la real-world cryptography
  - e.g., data formats, error messages, compression, etc.

Verification goals [FSE'16]

The following machine-checked proofs for Enc and Dec:

- Technical specification is provably IND\$-CPA secure
    - specify standard algorithms in EasyCrypt
    - specify IND\$-CPA game in EasyCrypt
    - prove specification secure assuming AES is a PRF or PRP
    - consider details a la real-world cryptography
    - e.g., data formats, error messages, compression, etc.
- Implementation is functionally correct:
    - semantics of implementation language in EasyCrypt
    - representation mapping between impl/spec types
    - prove, for all implementation inputs:
        - if, input represents value in spec type, then
        - output represents correct spec result

Verification guarantees [FSE'16]

Together imply implementation is IND$-CPA secure when:

- IND$-CPA is adapted to run code on adversarial implementation inputs
- Representation mapping is bijective
- Mapping can be computed efficiently both ways
- Ideal randomness assumed to be sampled at spec level

Verification guarantees [FSE'16]

Together imply implementation is IND$-CPA secure when:

- IND$-CPA is adapted to run code on adversarial implementation inputs
- Representation mapping is bijective
- Mapping can be computed efficiently both ways
- Ideal randomness assumed to be sampled at spec level

Constant-time guarantees security against timing attacker:

- Can assume adversary gets trace of memory addresses
- According to implementation language semantics

Verification guarantees [FSE'16]

Together imply implementation is IND$-CPA secure when:

- IND$-CPA is adapted to run code on adversarial implementation inputs
- Representation mapping is bijective
- Mapping can be computed efficiently both ways
- Ideal randomness assumed to be sampled at spec level

Constant-time guarantees security against timing attacker:

- Can assume adversary gets trace of memory addresses
- According to implementation language semantics

More precise/general statements can be made, but beyond scope.

Exercises

Extend full example to 256-bit messages

What changes if we want to encrypt 256-bit messages?

- Messages now take two AES blocks
- One bit of the input to the AES must be reserved for a counter
- Nonce must be at least one bit smaller
- This is 2-block counter mode in the nonce-based setting

The proof steps are the same modulo two samplings per message.

Implementation can still use only registers.

Optimize AES key schedule computation

Previous example, AES is computed multiple times with the same key.

This means key schedule can be computed only one time.

Changing this has implications in correctness proof:

- not immediate that all round keys can be kept in registers (spilling?)
- current proof in AES library assumes key schedule is computed each time

Refine security proof to make PRP assumption explicit

Security proof should be modified as follows:

- Hop 1: modify scheme to use ideal permutation
  - Reduce hop to PRP advantage against f
  - This step is similar to first hop in proof we saw
- Hop 2: modify scheme to use random function
  - Use generic RF/RP switching lemma
  - Advantage in distinguishing RF from RP bounded by

$$\frac{q \cdot (q-1)}{2^{\ell}}$$

Here ℓ is the block size.

- Final hop: wrap up as in proof we saw by using OTP argument

Extend full example to any number of blocks (AES CTR)

Extending the Jasmin implementation:

- Start from the example using memory input/output
- Reserve, e.g., 32-bits of the AES input for the counter
- Additional input for message length
- While loop needed to process all blocks
- Must deal with non-aligned messages/ciphertexts
- Optimize AES-NI usage to compute key scheduling only once

Extending the security proof:

- not much changes wrt to the 2-block case
- `while` rule adds some extra technicalities.

Correctness proof very annoying due to the use of memory.

Take-aways

Main take-aways on Jasmin

Using Jasmin for writing high-speed code:

+ It is a new language for optimized low-level code

+ Programming in Jasmin requires no knowledge of verification

+ Safety of Jasmin programs checked automatically

– Currently we only support x86-64 platforms

Main take-aways on Jasmin

Using Jasmin for writing high-speed code:

+ It is a new language for optimized low-level code

+ Programming in Jasmin requires no knowledge of verification

+ Safety of Jasmin programs checked automatically

− Currently we only support x86-64 platforms

Jasmin correctness and constant-time:

+ Jasmin correctness in EasyCrypt = standard Hoare logic

+ Jasmin CT in EasyCrypt = mostly automatic

Main take-aways on EasyCrypt

+ Specifying crypto in EC requires no knowledge of verification

+ Specifying game-hops in EC requires no knowledge of verification

– Proofs are not automatic, although some automation exists

– Multidisciplinary team required for getting end-to-end results

Thank you for attending!