Machine-Checked Proofs for AES: High-Assurance Security

v 1.0

Ji, Yong-hyeon

(`hacker3740@kookmin.ac.kr`)

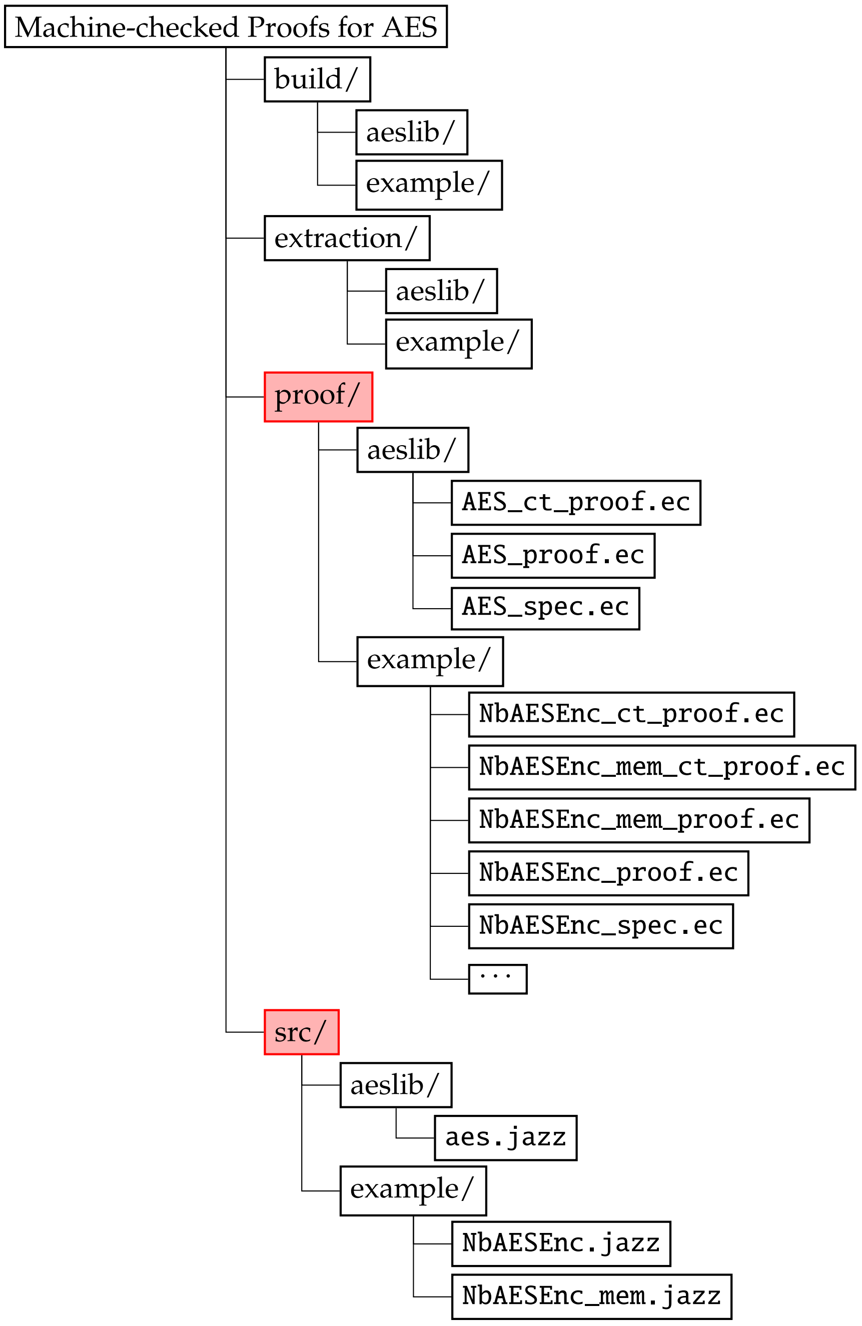
Department of Information Security, Cryptology, and Mathematics

College of Science and Technology

Kookmin University

December 30, 2024

File Structure



Copyright

Changelog

v1.0	2024-12-24	Initial release:

Contents

1 Preliminaries

1.1 Cryptosystem and Encryption Scheme

Cryptosystem

Definition 1. A **cryptosystem** is a five-tuple

$$(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D}),$$

where

(i) $\boxed{\mathcal{P}}$ is a finite set of plaintexts[a].

(ii) $\boxed{\mathcal{C}}$ is a finite set of ciphertexts[b].

(iii) $\boxed{\mathcal{K}}$ is a finite set of possible keys[c].

(iv) $\boxed{\mathcal{E} : \mathcal{K} \times \mathcal{P} \to \mathcal{C}}$ is a deterministic function that maps a key $k \in \mathcal{K}$ and a plaintext $p \in \mathcal{P}$ to a ciphertext $c \in \mathcal{C}$. Formally:

$$\begin{array}{rcccc} \mathcal{E} & : & \mathcal{K} \times \mathcal{P} & \longrightarrow & \mathcal{C} \\ & & (k, p) & \longmapsto & c \end{array}.$$

(v) $\boxed{\mathcal{D} : \mathcal{K} \times \mathcal{C} \to \mathcal{P}}$ is a deterministic function that maps a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ to a ciphertext $p \in \mathcal{P}$. Formally:

$$\begin{array}{rcccc} \mathcal{D} & : & \mathcal{K} \times \mathcal{C} & \longrightarrow & \mathcal{P} \\ & & (k, c) & \longmapsto & p \end{array}.$$

[a] These are the possible inputs to the encryption algorithm and typically represent meaningful data to be protected.

[b] These are the encrypted outputs of the encryption algorithm corresponding to plaintexts in $\mathcal{P}$.

[c] Each key $k \in \mathcal{K}$ determines a specific encryption and decryption function.

Remark 1 (Correctness Property)**.** For every key $k \in \mathcal{K}$ and every plaintext $p \in \mathcal{P}$, the decryption function is the inverse of the encryption function. That is:

$$\mathcal{D}(k, \mathcal{E}(k, p)) = p.$$

Remark 2 (Security)**.** The security of the cryptosystem is defined with respect to a particular adversarial model. Informally, a cryptosystem is secure if an adversary with limited computational resources cannot distinguish between the ciphertexts of any two plaintexts, even if they know the encryption algorithm but do not know the key.

Encryption Scheme

Definition 2. An **encryption scheme** is a three-tuple

$$\Pi := (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}).$$

where

(i) KeyGen is a probabilistic algorithm that ouputs a key $k \in \mathcal{K}$, where $\mathcal{K}$ is the key space. Formally:

$$\boxed{\mathsf{KeyGen} : \{0,1\}^* \longrightarrow \mathcal{K}},$$

where $\{0,1\}^*$ is the set of binary strings of arbitrary length (representing randomness or input seed). The ouput k is uniformly distributed over $\mathcal{K}$.

(ii) Enc is a (possibly probabilistic) algorithm that takes a key $k \in \mathcal{K}$ and a message $p \in \mathcal{M}$ (message space) and outpus a ciphertext $c \in \mathcal{C}$ (ciphertext sapce). Formally:

$$\boxed{\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \times \{0,1\}^* \to \mathcal{C}}.$$

The algorithm may use randomness (from $\{0,1\}^*$) to ensure that repeated encryptions of the same message $m \in \mathcal{M}$ under the same key $k \in \mathcal{K}$ yield different ciphertexts c.

(iii) Dec is a deterministic algorithms that takes a key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$ and ouputs the corresponding message $m \in \mathcal{M}$. Formally:

$$\boxed{\mathsf{Dec} : \mathcal{K} \times \mathcal{C} \longrightarrow \mathcal{M}}.$$

Remark 3 (Correctness Property)**.** For every $k \in \mathcal{K}$, $m \in \mathcal{M}$, and $c \in \mathcal{C}$, the scheme must satisfy

$$\mathsf{Dec}(k, \mathsf{Enc}(k, m; r)) = m$$

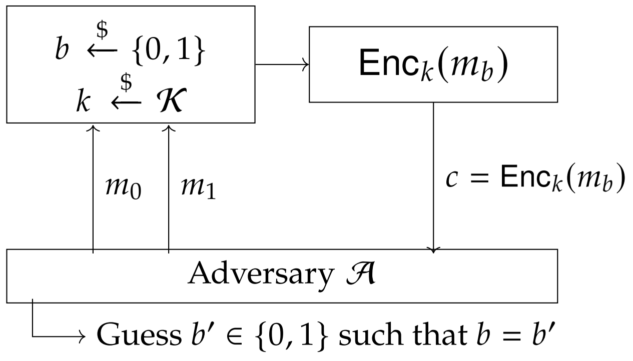
where r represents the random bits used by Enc.

Remark 4 (Security)**.** The security of an encryption scheme depends on the adversarial model. For **semantic security**, an encryption scheme must satisfy the following:

> "Given a ciphertext c, no computationally bounded adversary can distinguish between encryptions of any two messages m_0, m_1, even if they are chosen adaptively by the adversary."

Example 1 (IND-CPA)**.**
The **indistinguishability under chosen plaintext attack (IND-CPA)** model:

1. The adversary chooses two messages m_0, m_1.
2. A random bit $b \in \{0,1\}$ is chosen, and the ciphertext $c = \mathsf{Enc}(k, m_b)$ is provided to the adversary.
3. The adversary outputs a guess $b' \in \{0,1\}$.


$b \overset{\$}{\leftarrow} \{0,1\}$
$k \overset{\$}{\leftarrow} \mathcal{K}$
$\mathsf{Enc}_k(m_b)$
m_0
m_1
$c = \mathsf{Enc}_k(m_b)$
Adversary $\mathcal{A}$
Guess $b' \in \{0,1\}$ such that $b = b'$

The scheme is secure if the adversary's advantage is negligible:

$$\mathrm{Adv}_{\Pi}^{\text{IND-CPA}}(\mathcal{A}) := \left|\Pr[b' = b] - \frac{1}{2}\right| \leq \mathrm{negl}(\lambda),$$

where λ is the security parameter.

1.2 Perfect Security

Perfect Security of an Encryption Scheme

Definition 3. An encryption scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is **perfect security** if, for every $m \in \mathcal{M}$, $c \in \mathcal{C}$, and $k \in \mathcal{K}$ such that $\mathsf{Enc}(k, m) = c$, the following holds:

(i) **Ciphertext Independence**:

$$\Pr[M = m \mid C = c] = \Pr[M = m],$$

where

- M is the random variable representing the plaintext.
- C is the random variable representing the ciphertext.

(ii) **Key Uniformity**: The key K must satisfy:

$$\Pr[\mathsf{Enc}(K, m) = c] = \Pr[C = c],$$

for all $m \in \mathcal{M}$, $c \in \mathcal{C}$, and uniformly random K.

Remark 5.

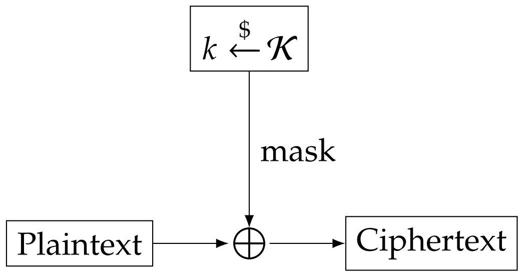
$$\Pr[M = m \mid C = c] = \Pr[M = m] \iff \Pr[C = c \mid M = m] = \Pr[C = c]$$

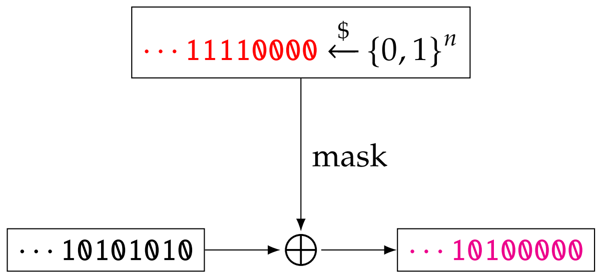
Example 2 (One-Time Pad)**.** The one-time pad encryption scheme is perfect security.

- $\mathcal{M} = \mathcal{C} = \mathcal{K} = \{0, 1\}^n$;
- $\mathsf{Enc}(k, m) = m \oplus k$, where $\oplus$ is bitwise XOR;
- $\mathsf{Dec}(k, c) = c \oplus k$.

We must show that $\Pr[C = c \mid M = m] = \Pr[C = c]$. For $m \in \mathcal{M}$ and $c \in \mathcal{C}$,

$$\Pr[C = c \mid M = m] = \sum_{k \in \mathcal{K}} \Pr[K = k]$$





Theorem 1. *The one-time pad encryption scheme is perfectly secret.*

Proof.

$$\begin{aligned}\Pr[C = c \mid M = m] = \Pr[c = \mathsf{Enc}(K, m)] &= \Pr[c = m \oplus K] \\ &= \Pr[K = m \oplus c] \\ &= 2^{-n} \quad \text{if } K \xleftarrow{\$} \mathcal{K} = \{0,1\}^n\end{aligned}$$

Fix any distribution over $\mathcal{M}$. For any $c \in \mathcal{C}$, we have

$$\begin{aligned}\Pr[C = c] &= \sum_{m \in \mathcal{M}} \Pr[C = c \mid M = m] \cdot \Pr[M = m] \\ &= 2^{-n} \cdot \Pr[M = m]\end{aligned}$$

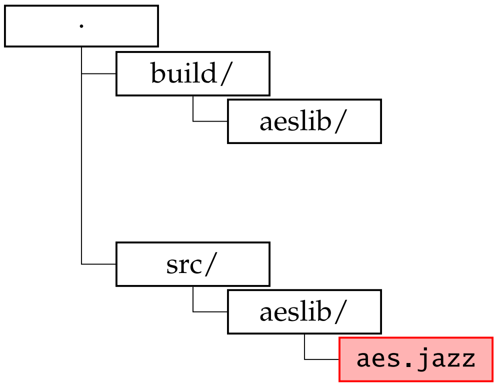
By Bayes' Theorem, we obtain

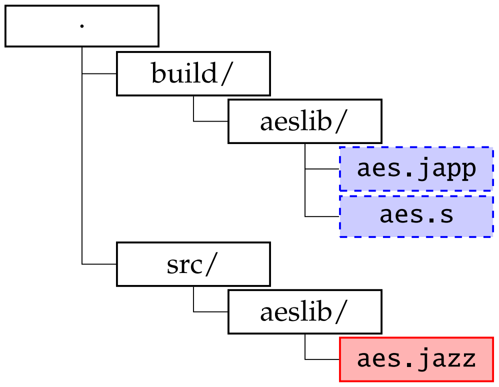
$$\begin{aligned}\Pr[M = m \mid C = c] &= \frac{\Pr[C = c \mid M = m] \cdot \Pr[M = m]}{\Pr[C = c]} \\ &= \frac{2^{-n} \cdot \Pr[M = m]}{2^{-n}} \\ &= \Pr[M = m].\end{aligned}$$

□

2 Machine-checked Proofs for AES

2.1 Build AES Library





Code 1: aes.jazz

```
...
/* Jasmin implementation of
AES using AES-NI */
#ifdef EXPORT_TEST
export
#else
inline
#endif
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;
}
...
```

Code 2: aes.japp

```
...




export



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;
}
...
```

Code 3: aes.s

```
...
        .att_syntax
.text
.p2align        5
.globl  _aes
.globl  aes

_aes:
aes:
...
vmovdqu %xmm0, %xmm12
vpxor   %xmm2, %xmm1, %xmm0
aesenc  %xmm3, %xmm0
...
aesenc  %xmm11, %xmm0
aesenclast      %xmm12, %xmm0
ret
...
```

```
@~$ cpp -nostdinc -DEXPORT_TEST src/aeslib/aes.jazz \
      | grep -v "^#" > build/aeslib/aes.japp
@~$ jasminc  build/aeslib/aes.japp -o build/aeslib/aes.s
```

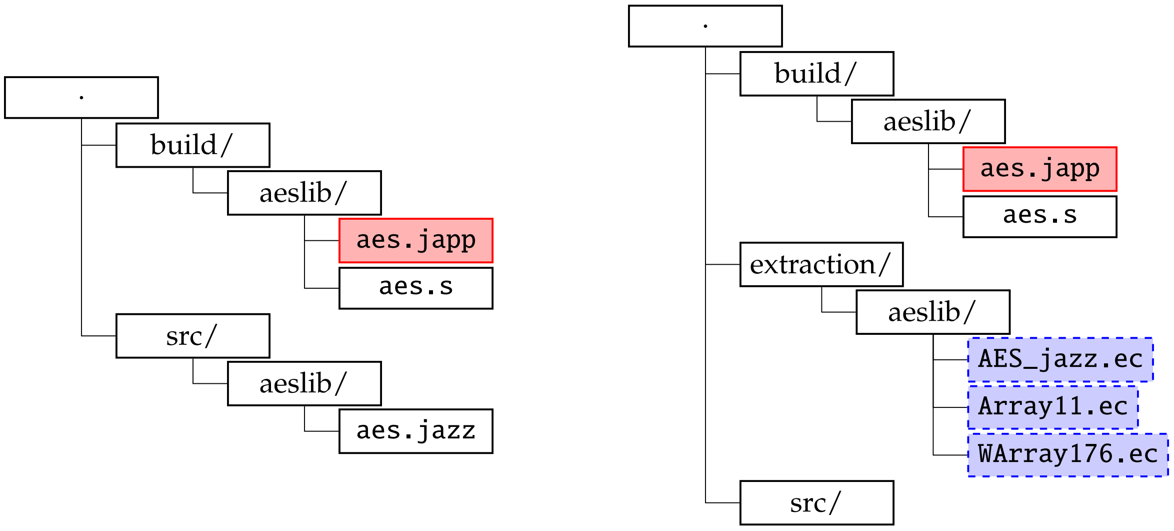
Command 1 :

`[cpp]`	*C Preprocessor*	Preprocesses source code files by expanding macros, processing conditional compilation directives, and handling file inclusions.
`[-nostdinc]`	*No Standard Include*	Prevents the preprocessor from searching in standard system directories for include files, limiting the scope of header file processing to explicitly specified paths or local includes.
`[-DEXPORT_TEST]`	*Define Macro* `EXPORT_TEST`	Defines a preprocessor macro (`EXPORT_TEST`) that can be used for conditional compilation within the source code.
`[../aes.jazz]`	*Source File Path*	Specifies the input Jasmin source code file to be preprocessed.
`[|]`	*Pipe*	Passes the output of the cpp command as input to the next command (`grep`).
`[grep -v "^#"]`	*Global Regular Expression Print (with* `-v` *for inverse matching)*	Filters out lines starting with #, such as preprocessor directives or comments, from the preprocessed output.
`[>]`	*Output Redirection*	Redirects the filtered output from the preprocessing pipeline to a specified file.
`[../aes.japp]`	*Output File Path*	Specifies the destination file for the preprocessed Jasmin code.

Command 2 :

`[jasminc]`	*Jasmin Compiler*	Compiles Jasmin source code into optimized assembly code.
`[../aes.japp]`	*Input File Path*	Specifies the preprocessed Jasmin source file to be compiled.
`[-o]`	*Output File Option*	Indicates the output file name or path for the generated assembly code.
`[../aes.s]`	*Output File Path*	Specifies the destination file for the generated assembly code.

2.2 Extract AES Library for Correctness and Security



```
@~$ jasminc build/aeslib/aes.japp -ec aes -ec invaes -oec AES_jazz.ec
@~$ mv AES_jazz.ec Array11.ec WArray176.ec extraction/aeslib
```

[jasminc]	The 'jasminc' command in this context extracts verification conditions specifically designed to interface with EasyCrypt, a tool used for formal reasoning about cryptographic security.
[-ec aes -ec invaes]	The '-ec' flag specifies functions or modules to extract formal verification constraints from the Jasmin source file.
[-oec AES_jazz.ec]	The '-oec' flag specifies the output file where the extracted constraints for the specified modules (aes and invaes) will be saved.

Code 4: Array11.ec

```
from Jasmin require import JArray.

clone export PolyArray as Array11  with op size <- 11.
```

Code 5: WArray176.ec

```
from Jasmin require import JArray.

clone export PolyArray as Array11  with op size <- 11.
```

Code 6: AES_jazz.ec (lines 127-134)

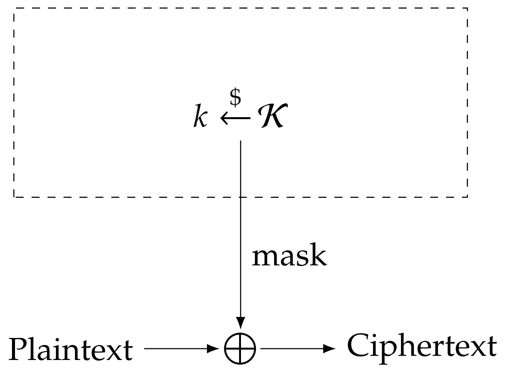
```
proc aes (key:W128.t, in_0:W128.t) : W128.t = {
  var out:W128.t;
  var rkeys:W128.t Array11.t;
  rkeys <- witness;
  rkeys <@ keys_expand (key);
  out <@ aes_rounds (rkeys, in_0);
  return out;
}
```

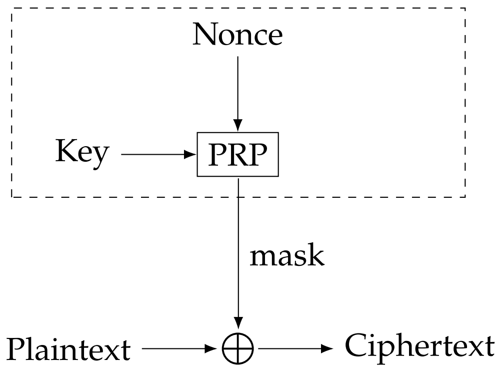
2.3 Extract AES Library for Constant-time

```
@~$ jasminc build/aeslib/aes.japp -CT -ec aes -ec invaes -oec AES_jazz_ct.ec
@~$ mv AES_jazz_ct.ec Array11.ec WArray176.ec extraction/aeslib
```

2.4 Build Nonce-based AES Encryption with register calling convension

```
@~$ cpp -nostdinc src/example/NbAESEnc.jazz  \
     | grep -v "^#" > build/example/NbAESEnc.japp
@~$ jasminc build/example/NbAESEnc.japp -o build/example/NbAESEnc.s
```





- The **one-time pad (OTP) encryption scheme** is a cryptographic construct that achieves perfect security.

$$\Pi_{\mathrm{OTP}} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}),$$

  where

  (i)
  (ii) $\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C} : (k, m) \mapsto c = k \oplus m;$

$$\begin{array}{rcccl} \mathsf{Enc} & : & \mathcal{K} & \longrightarrow & \mathcal{C}^{\mathcal{M}} \\ & & k & \longmapsto & c = \mathsf{Enc}_k(m) \end{array} \quad \text{where} \quad \begin{array}{rcccl} \mathsf{Enc}_k & : & \mathcal{M} & \longrightarrow & \mathcal{C} \\ & & m & \longmapsto & c = k \oplus m \end{array}$$

- A **nonce-based PRP encryption scheme** is a cryptographic construct where a nonce (number used once) is incorporated to ensure unique ciphertexts for the same plaintext under the same key.

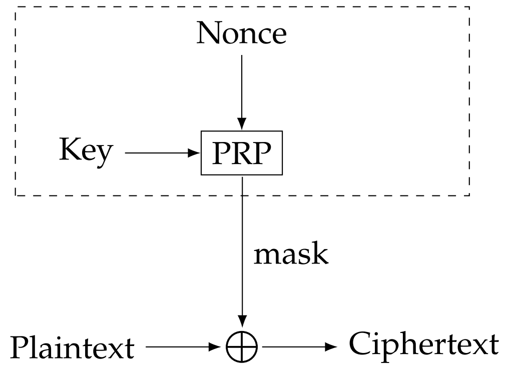
$$\Pi_{\mathcal{N}-\mathrm{PRP}} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}),$$

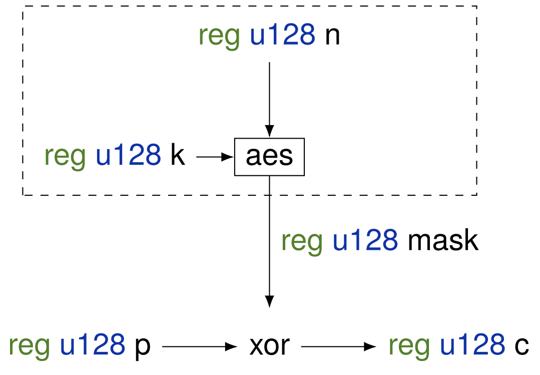
  where

  (i)
  (ii) $\mathsf{Enc} : \mathcal{K} \times \mathcal{N} \times \mathcal{M} \to \mathcal{C} : (k, n, m) \mapsto c = \mathsf{Enc}_k(n) \oplus m;$

$$\begin{array}{rcccl} \mathsf{Enc} & : & \mathcal{K} & \longrightarrow & [\mathcal{N} \to [\mathcal{M} \to \mathcal{C}]] \\ & & k & \longmapsto & c = \mathsf{Enc}_k(n) \oplus m \end{array} \quad \text{where} \quad \begin{array}{rcccl} \mathsf{Enc}_k & : & \mathcal{N} & \longrightarrow & [\mathcal{M} \to \mathcal{C}] \\ & & n & \longmapsto & c = k \oplus m \end{array}$$

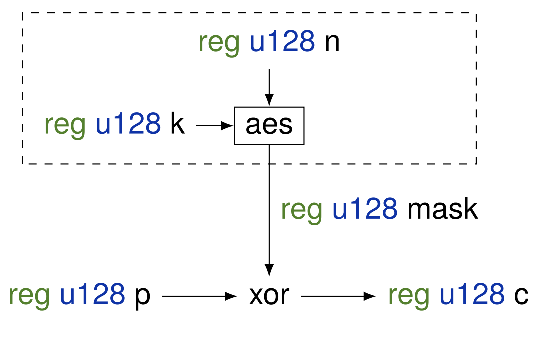
2.5 Implementation of Nonce-based AES Encryption Scheme

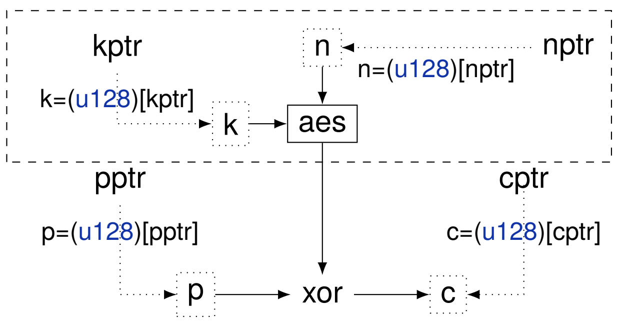




Code 7: src/example/NbAESEnc.jazz

```
/* Nonce-based symmetric encryption for 16-byte messages using AES as a PRF */

/* We use cpp to manage modules in Jasmin */
#include "../aeslib/aes.jazz"

/* We make xor into a function, but this costs nothing because
   Jasmin compiler does not include inlining moves (warning issued o/w). */
inline fn xor(reg u128 a, reg u128 b) → reg u128 {
   reg u128 r;
   r = a^b;
   return r;
}

/* These functions can be called from C for testing. */

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);
}
```





Code 8: src/example/NbAESEnc_mem.jazz

```
/* Nonce-based symmetric encryption for 16-byte messages using AES as a PRF */

#include "../aeslib/aes.jazz"
inline fn xor(reg u128 a, reg u128 b) → reg u128 {
  reg u128 r;
  r = a^b;
  return r;
}

/* These functions can be called from C for testing.
  They receive pointers to byte arrays of length 16.
  Convention is that first pointer is writable for output. */

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;
}
```

Code 9: test/test_NbAESEnc.c

```c
#include <smmintrin.h>
#include <immintrin.h>
#include <stdalign.h>
#include <stdio.h>

int8_t kb[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
int8_t nb[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x1f };
int8_t pb[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x2f };

extern __m128i enc(__m128i k, __m128i n,__m128i p);
extern __m128i dec(__m128i k, __m128i n,__m128i c);

int main() {
    __m128i ct, dt, pt, k, n;

    k = _mm_loadu_si128((__m128i *) kb);
    n = _mm_loadu_si128((__m128i *) nb);
    pt = _mm_loadu_si128((__m128i *) pb);

    ct = enc(k,n,p);
    dt = dec(k,n,c);

    __m128i neq = _mm_xor_si128(pt,dt);
    if(_mm_test_all_zeros(neq,neq)) printf("Verify output: OK!\n");
    else printf("Verify output: Not OK!\n");

    return 0;
}
```

```
@~$ cpp -nostdinc src/example/NbAESEnc.jazz  \
   | grep -v "^#" > build/example/NbAESEnc.japp
@~$ jasminc  build/example/NbAESEnc.japp -o build/example/NbAESEnc.s
@~$ gcc -msse4.1 -Wall build/example/NbAESEnc.s test/test_NbAESEnc.c \
-o build/example/test_NbAESEnc
```

```
*************************************************
***  Testing encryption scheme (reg cc)       ***
*************************************************
build/example/test_NbAESEnc
Verify output: OK!
```

Code 10: test/test_NbAESEnc_mem.c

```c
#include <stdio.h>

byte k[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
byte n[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x1f };
byte pt[] = {
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x2f };

extern void enc(byte *cp, byte *kp, byte *np, byte *pp);
extern void dec(byte *pp, byte *kp, byte *np, byte *cp);

int crypto_verify(const byte *x,const byte *y) {
    unsigned int differentbits = 0;
#define F(i) differentbits |= x[i] ^ y[i];
    F(0) F(1) F(2) F(3)
    F(4) F(5) F(6) F(7)
    F(8) F(9) F(10) F(11)
    F(12) F(13) F(14) F(15)
    return (1 & ((differentbits - 1) >> 8)) - 1;
}

int main() {
    byte ct[16], dt[16];
    enc(ct,k,n,pt);
    dec(dt,k,n,ct);

    if (crypto_verify(pt,dt) == 0) printf("Verify output: OK!\n");
    else printf("Verify output: Not OK!\n");

    return 0;
}
```

References

[1] Jonathan, Katz. *Introduction to Modern Cryptography, Second Edition*., n.d.

[2] Smart, Nigel P. *Cryptography Made Simple. Information Security and Cryptography*. Cham: Springer International Publishing, 2016. https://doi.org/10.1007/978-3-319-21936-3.