UNIVERSITÄT
BERN

Implementing RSA signatures on the Internet Computer

Signatures in a distributed system

Bachelor Thesis

Michael Senn

from

Zürich, Switzerland

Faculty of Science, University of Bern

01. May 2021

Prof. Christian Cachin
Jovana Micic
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

The Internet Computer is a distributed computing platform, providing fault-tolerant and scalable application hosting by spreading computations across multiple physical servers. Applications for the Internet Computer can be developed in multiple languages, one of which - Motoko - was created specifically for that purpose.

PKCS#1 is an industry standard which formalises properties, operations and schemes for cryptographic signatures and encryption based on the RSA algorithm.

In this thesis a Motoko library was written which allows generation and verification of cryptographic RSA signatures conformant to PKCS#1. This library allows applications running on the Internet Computer to utilize these signatures when interacting with each other, as well as external applications.

Contents

Chapter 1

Introduction

Distributed systems are systems, components of which are spread across different physical or virtual machines. Their components exchange messages with each other to work towards a shared goal. In doing so they appear as one coherent system to an outside viewer [TS02].

Two of the advantages offered by distributed systems are fault-tolerance and scalability. Fault-tolerance refers to the characteristic that the failure of a single component does not affect applications running on the distributed system. Scalability refers to the characteristic that they can be easily extended by adding additional components, increasing resiliency or processing power [TS02].

A recent project in the field of distributed systems is the Internet Computer ("IC") by the DFINITY foundation. The IC aims to provide a platform for distributed, fault-tolerant and scalable hosting of applications ("canisters"). It does so by distributing the execution of canisters across independent physical servers ("nodes"), and utilising a blockchain-like consensus protocol to achieve consensus between all involved nodes [Wil20].

These canisters can be thought of as a generalisation of smart contracts, as they exist on e.g. the Ethereum platform. Unlike these smart contracts, canisters are general-purpose applications which can be written in many languages. One such language — "Motoko" — has been created specifically for this purpose by the DFINITY foundation.

Cryptographic digital signatures are mathematical schemes which allow establishing authenticity and integrity of messages. Authenticity refers to the the ability to establish the identity of the sender, integrity to the ability to establish that messages were not modified in transit. Cryptographic digital signatures are defined on top of public-key cryptography systems such as RSA [Sch15].

Cryptographic signatures play a vital function in modern IT systems. They are standardised in, and used as part of, multiple technology standards such as TLS and X.509.

PKCS#1 is an industry standard formalising encryption and signature schemes based on the RSA algorithm. Signature schemes defined in PKCS#1 are used in other standards such as X.509, which defines the format for storage of public certificates of a public-key infrastructure.

In this thesis we designed and implemented a Motoko library to create and verify digital signatures conformant to the PKCS#1 standard. The library allows any Motoko application running on the IC to verify PKCS#1 signatures created by external software, as well as to sign messages such that the signatures verify in external software.

As Motoko's functionality and extent of its standard library is still limited, we further implemented a lot of supporting code. This code covers topics like modular arithmetic with big numbers and data encodings such as Base64, along with general-purpose utility methods.

The library does not support key generation and exchange, as those are not part of PKCS#1. Nor does it support the encryption schemes defined in PKCS#1.

This thesis will start with an overview of existing technologies, standards and implementations in Chapter 2. Chapter 3 will cover the topic of efficiently performing modular arithmetic. Chapter 4 will cover various general-purpose algorithms and methods which were implemented. Chapter 5 will focus on the PKCS#1 standard and its implementation in Motoko, as well as compatibility with other implementations conformant to PKCS#1 and an evaluation of this library's performance. Finally we will provide a quick overview of our experience with Motoko and the IC in Chapter 6, and conclude in Chapter 7.

1.1 A note about source code comments & unit tests

All library methods intended for usage by an external developer as well as many internal methods have been documented. Documentation covers the purpose of each method, the parameters they take and potential errors which may arise. Such method comments have been left out wherever source code was embedded in this thesis, as they can be quite verbose.

The vast majority of source code has further been extensively covered with automated unit tests. Care has been taken to cover edge cases which may arise. The collection of unit tests can be found in Appendix A, but is more suited for being run programatically than it makes for interesting reading.

Chapter 2

Background

In this chapter we provide an overview of Motoko and the Internet Computer, to facilitate an understanding of the platform the library was written for. We then introduce the PKCS#1 standard, to ease understanding of the implementation details discussed in later chapters.

2.1 Internet Computer architecture

The IC is a distributed computing platform for general-purpose applications, called "canisters", made out of independent physical nodes. Its goal is to provide an execution environment which is resilient towards both faulty, as well as malicious, nodes [DFI21].

2.1.1 Subnets

To ensure resiliency towards faulty and malicious nodes, the IC groups a set of independent — both geographically and in terms of ownership — nodes into a so-called subnet. Eventually DFINITY plans to add support for different types of subnets with different numbers of nodes, to provide varying levels of resiliency [Wil20].

2.1.2 Canisters

Canisters are compiled WebAssembly modules — making up the actual program logic — along with the memory pages associated with these modules. In this regard they are comparable to application containers the likes of Docker [Wil20].

Each canister, when registered on the IC, is assigned one subnet within which it will run.

2.1.3 Canister communication

Canisters are able to exchange messages with all other canisters running on the IC, independent of the subnet they are on. The platform ensures that such messages are delivered. Canisters can choose to act on messages they receive, leading to an intuitive actor-based programming model suitable for development.

The IC differentiates between **update** and **query** messages. Any code executed as response to an update message will execute on all the nodes of a subnet, and changes made in a canister's memory will persist. Any code executed as response to a query message will execute on only one node, and changes to memory will be discarded [DFI21].

Any data retrieved as the result of a query call is as such susceptible to faulty or malicious nodes, whereas results of update calls are ensured to be reliable by the consensus algorithm.

2.1.4 Orthogonal persistence

The IC offers what is called "orthogonal persistence". This refers to the memory of each canister persisting between function calls as well as between upgrades of the canister's code. This largely eliminates the need for traditional persistent storage such as databases, but requires the use of data structures suitable for in-memory storage.

2.1.5 Consensus

A consensus algorithm is used to ensure that the same messages in the same order are delivered on all nodes. In the case of update calls, consensus also ensures that the changes which are made to a canister's memory are consistent across all nodes it runs on [DFI21].

Details about the consensus algorithm are not public as of the time of writing.

As update calls pass through consensus they are inherently slow, with execution overhead on the order of seconds. Query calls, which only execute on one node, are faster, with execution overhead on the order of milliseconds.

2.1.6 Limitations

Due to the requirement to find consensus between involved nodes, execution must generally be deterministic. Randomness must be network-mediated, and can not be determined by each node on its own.

Support for network-mediated randomness is in the process of being added, allowing generation of random numbers within canisters without breaking consensus [DFI21].

A less obvious limitation is that arbitrary network IO is impossible, as there is no guarantee that it is deterministic and idempotent. As an example it is currently impossible to call an external HTTP API from within an application running on the IC, limiting integration with external systems.

Eventually support may be added through intermediary oracles, but there are no plans for this as of yet [Bre21].

2.1.7 Token economy

The IC supports a token to pay for services. Canisters can be charged with tokens from a user's wallet, with each operation of a canister using up parts of this balance based on the complexity of the execution. Tokens are paid out to both providers of hardware as well as entities participating in network governance [Wil21].

2.2 Motoko

Motoko is a language created by the DFINITY foundation specifically for development of applications on the IC. It is a modern statically-typed language which compiles to WebAssembly [DFI21].

This section intends to provide the reader with sufficient understanding of its concepts to understand the code in later chapters of the thesis. For a full introduction the official guide at `https://sdk.dfinity.org/docs/index.html` should be consulted.

2.2.1 Hello world

As is tradition, we start with the typical 'Hello world' program.

```
import Debug "mo:base/Debug";

module {
  public func greeting() : () {
    Debug.print("Hello world!");
  };
};
```

The first line imports the "Debug" module from the Motoko standard library, which allows accessing STDOUT of a local IC replica. We then define a module, which is equivalent to a class containing exclusively static methods in Java, or a module in Go.

Within we define a public function called "greeting". It takes no parameters and has no return value, indicated by the empty type "()". In this function we call a function defined in the previously imported "Debug" module to print to STDOUT.

2.2.2 Built-in types

The Motoko base language and standard library define a number of types, the most important of which are listed here.

Bool Boolean.

Char Single character. Can be converted to Unicode code point.

Text String, sequence of characters.

Int, Int8, Int16, Int32, Int64 Signed integer with checked arithmetic. Supports values in $[-2^{N-1}, 2^{N-1}-1]$.

Nat, Nat8, Nat16, Nat32, Nat64 Unsigned integer with checked arithmetic. Supports values in $[0, 2^N-1]$.

Word8, Word16, Word32, Word64 Unsigned integer with modular arithmetic. Supports values in $[0, 2^N - 1]$.

Float Double-precision floating point number.

Array<T> Fixed-size array of T.

Buffer<T> Variable-size array of T. Grows if required.

List<T> Linked list of T.

Iter<T> Iterator of T. Repeatedly yields values of type T until it runs empty.

Result<Ok, Err> Variant type, can be either of type Ok or of Type Err. Used for error handling.

2.2.3 Variable assignment, mutable and immutable variables

Motoko differentiates between mutable and immutable variables. Immutable variables are instantiated with the "let" keyword, mutable ones with the "var" keyword. Mutable variables may be reassigned with the ':=' operator.

Types are inferred where possible, and have to be specified otherwise.

```
// Type of x is inferred as Int
let x = -20;
// Type of y is specified as Float
let y : Float = -20;

// Invalid reassignment of immutable variable
// y := -40;

// Valid reassignment of mutable variable
var z : Text = "Helo world";
z := "Hello world";

// Immutable array of Nat
```

```
let a : [Nat] = [0, 1, 2];
a[2]; // => 2
// Invalid reassignment of immutable array member
// a[1] := 99;

// Mutable array of Nat. Mind that the *array itself* is immutable!
let b : [var Nat] = [var 0, 1, 2];
b[1] := 99;
b[1]; // => 99
```

2.2.4 Objects and types

Like in other object-oriented languages, objects are a combination of state and function. Objects can be created directly, but can also be instantiated via a class.

```
class Greeter(initName : Text) {
  var name = initName;

  public func greet() : Text {
    return "Hello " # name;
  };
};

// Object instantiated from class
let g = Greeter("John");
g.greet(); // => "Hello John"

// Object instantiated directly
object counter {
  var count = 0;

  public func inc() : Nat {
    count += 1;
    return count;
  };
};
counter.inc(); // => 1
```

2.2.5 Structural subtyping

Motoko utilises structural subtyping rather than nominal subtyping. This means that whether one type is a subtype of another depends exclusively on its interface, and not on any explicitly established relationship. Readers familiar with Go may recognise this pattern from the way interfaces work in that language.

As an example consider the two following types. Without explicitly being established as such, “EmailGreeter” is a subtype of the more general “Greeter” class.

```
class Greeter(initName : Text) {
  var name = initName;

  public func greet() : Text {
    return "Hello " # name;
  };
};

class EmailGreeter(initName : Text) {
  var name = initName;

  public func greet() : Text {
    return "Hello " # name;
  };
```

```
  public func sendGreetingMail(address : Text) : () {
    // Send mail
  };
};
```

2.2.6 Pattern matching

Pattern matching is a language feature to decompose structured data. It is used to e.g. differentiate between variant types or to extract fields of objects in method signatures. Two examples are encountered below in the sections about variant and optional types, a more thorough explanation can be found in the official documentation.

2.2.7 Variant types

Variant types are special types which can be one of a well-defined set of types. Consider the following type which represents a point on a plane. A value of this type can be either the origin (in which case it is a null type), a polar representation — in which case it is an object with angle and distance — or a cartesian representation in which case it is an object with an x and y coordinate.

```
type point = variant {
  origin : null;
  polar : { angle : Float; distance : Float };
  cartesian : { x : Float; y : Float };
};
```

Another use for variant types is the implementation of enums, where the payload of each type is implicitly “null”.

```
type day = variant { monday; tuesday; wednesday; thursday; friday; saturday; sunday;
    ↳ };
```

A common use is error handling, for which a dedicated “Result” type exists. Consider the following function which performs integer division of two positive integers, gracefully handling the case where the divisor is 0.

```
func div(a : Nat, b : Nat) : Result.Result<Nat, Text> {
  if (b == 0) {
        return { #err("Cannot divide by zero!") };
      } else {
        return { #ok(a / b) };
  };
};

switch (div(27, 3)) {
  case(#ok(result)) {
        // #ok type of this Result is Nat
        Debug.print("The result was: " # result);
      };
      case(#err(error)) {
        // #err type of this Result is Text
        Debug.print("Error while dividing: " # error);
      };
};
```

Error handling could be implemented with custom variant types, but due to being such a common operation the standard library provide the “Result” module with a dedicated type and several utility functions.

2.2.8 Optional types

Motoko supports optional types — a kind of 'typesafe null values'. It allows e.g. having one parameter of a method be optional without violating type safety. As an example consider the following function which has one mandatory and one optional parameter.

```
func greeting(name : Text, title : ?Text) {
  switch (title) {
    case (null) return "Hello " # name;
    case (?t) return "Hello " # t # " " # name;
  };
};
```

While the "title" parameter may be either a null value or a string, the requirement to handle both cases separately via pattern matching ensures that type safety is not violated.

2.3 PKCS#1

PKCS#1 defines two encryption and two signature schemes based on the RSA cryptosystem. On the encryption side it defines "RSAES-OAEP", which utilises the optimal asymmetric padding scheme defined by Bellare and Rogaway [BR95], as well as "RSAES-PKCS1-v1_5", which is based on an earlier version of PKCS#1. Using the latter is only recommended for backwards compatibility due to security concerns [Mor+16].

On the signature side it defines two signature schemes. "RSASSA-PKCS1-v1_5" is a deterministic signature scheme, "RSASSA-PSS" a probabilistic signature scheme with a formal security proof by Bellare and Rogaway [MP00]. Both signature schemes are signature schemes with appendix, meaning that they do not allow retrieving the message from the signature alone. Rather, these signatures are intended to be transmitted in addition to the message — e.g. as an appendix.

The structure of the standard is such that it first introduces a set of operations — called "primitives". These describe tasks such as conversion between byte strings and numerical representations, as well as how to apply the RSA cryptosystem to convert between plain- and ciphertext, respectively messages and signatures.

It then defines a set of encoding methods which generally take arbitrary byte sequences and encode them in a way that they are suitable for encryption respectively signing. This can involve elements such as hashing or introducing probabilistic elements.

Finally it then defines the four schemes mentioned above, using the introduced primitives and encoding methods.

2.3.1 Usage in modern technology standards

The signature schemes defined by PKCS#1 are used as part of other standards. In X.509, which defines the formats of public-key certificates, both signature algorithms are valid for signing certificates [Pol+02] [Sch+05]. In the TLS 1.3 standard they are also designated as valid signature schemes to use [RM18].

Chapter 3

Modular big number arithmetic

In this chapter we will provide an overview of the problem of modular exponentiation with big numbers. We will introduce methods to improve performance of modular exponentiation, with a focus on the way it is used as part of RSA, and describe their implementation in Motoko.

3.1 Introduction

Recall that RSA is based on it being feasible to find integers e, d and N, such that:

$$m \equiv (m^e)^d \equiv (m^d)^e \mod N \tag{3.1}$$

That is

$$e \cdot d \equiv 1 \mod N \tag{3.2}$$

This allows utilising modular exponentiation with one of the exponents as the forward direction, and modular exponentiation with the other exponent as the backward direction of the RSA operation.

A naive implementation of modular exponentiation with an n bit exponent will repeatedly multiply by the base and calculate the remainder of division with the modulus. Such an algorithm will require $O(2^n)$ multiplications and divisions.

As security of RSA is directly linked to the bit size of the modulus N and its exponents e and d, sizes of at least 2048 bits are recommended as of 2015 [BD15].

For exponents of this size, naive implementations of modular exponentiation do not perform well due to their exponential runtime cost. To improve performance, various optimisations for different parts of modular exponentiation exist, some of which will be discussed in detail.

3.2 Chinese Remainder Theorem

The Chinese remainder theorem ("CRT") states that, when the remainder of the division of an integer i by pairwise coprime integers d_i is known, that the remainder of the division of i by the product $d = \prod d_i$ can be determined.

This can be applied to RSA to speed up modular exponentiations of the form $m = c^d \mod n$, as originally shown by Quisquater and Couvreur in Electronic Letters [QC82]. Algorithm 1 shows the algorithm they described.

As the bit length of the two utility exponents d_p and d_q is approximately half the bit length of d, performance of exponentiation is increased by a factor of roughly four.

The CRT was used to speed up modular exponentiation during the signature verification operation as described by PKCS#1 [Mor+16]. Implementation details will be discused in Chapter 5.

Algorithm 1 Speeding up RSA with the CRT

Input

Private key: $K = (n, p, q, d)$
$d_p = d \mod (p-1)$
$d_q = d \mod (q-1)$
$q_{inv} = q^{-1} \mod p$
Message representative: Integer *m*

Algorithm

$m_1 = c^{d_p} \mod p$
$m_2 = c^{d_q} \mod q$
$h = q_{inv}(m_1 - m_2) \mod p$
$m = m_2 + h \cdot q \mod (p \cdot q)$
Return *m*

3.3 Exponentiation by squaring

Exponentiation by squaring is a way to significantly lower the amount of multiplications required for exponentiation. It is based on the observation that any power of a number a can be written as product of factors of a and the squaring operation, as described by Knuth [Knu96]. Consider a^{57}:

$$\begin{aligned} a^{57} &= a^{56} \cdot a = (a^{28})^2 \cdot a = ((a^{14})^2)^2 \cdot a = (((a^7)^2)^2)^2 \cdot a \\ &= (((a^6 \cdot a)^2)^2)^2 \cdot a = ((((a^3)^2 \cdot a)^2)^2)^2 \cdot a = ((((a^2 \cdot a)^2 \cdot a)^2)^2)^2 \cdot a \end{aligned}$$

This allows calculating a^{57} with only eight multiplications instead of the naive 57. The same also applies to modular exponentiation due to the properties of the modulo operation.

$$(a \cdot b) \mod p = ((a \mod p) \cdot (b \mod p)) \mod p$$

As an example:

$$a^7 \mod x = (((((a^2 \mod x) \cdot a) \mod x)^2 \mod x) \cdot a) \mod x$$

Algorithm 2 shows exponentiation by squaring as described by Schneier in Applied Cryptography [Sch15].

Algorithm 2 Modular exponentiation by squaring

Input

Base: Integer *b*
Exponent: Integer *exp*
Modulus: Integer *n*

Algorithm

```
If n == 1
    Return 0
result := 1
base := base mod n
While exp > 0
    If exp mod 2 == 1
        result := (result * base) mod n
    exp := exp / 2
    base := (base * base) mod n
Return result
```

3.3.1 Complexity

Consider a modular exponentiation $a^b \mod c$. Assume for simplicity that the bit-length of a, b and c is k. There will be k iterations of the loop, each iteration performing up to two multiplications and four divisions.

Assuming multiplication and division of k-bit numbers is $O(k^2)$, the expected complexity of this algorithm is hence $O(k^3)$, so polynomial in the bit length of the input.

3.3.2 Implementation in Motoko

The implementation in Motoko is a literal adoption of the algorithm, accounting only for the fact that the parameters passed to the function are immutable.

```
public func modPow(base : Nat, exponent : Nat, modulus : Nat) : Nat {
  if (modulus == 1) {
    return 0;
  };

  // result = b^0 = 1
  var result : Nat = 1;
  // Base residue
  var b : Nat = base % modulus;
  var e : Nat = exponent;

  while (e > 0) {
    if (e % 2 == 1) {
      // Least-significant bit of exponent is 1
      result := result * b % modulus;
    };
    e := e / 2;

    b := b**2 % modulus;
  };

  return result;
};
```

3.4 Montgomery modular multiplication

An attempt was made to implement Montgomery modular multiplication as described by Montgomery [Mon85], but the performance of the implementation was significantly worse than the one of exponentiation-by-squaring. Due to Motoko's lack of profiling tools it turned out to be impossible to reliably investigate the cause, and this method was abandonded. The code remains accessible in the project repository, but will not be elaborated upon further.

Chapter 4

Implementation of supporting methods in Motoko

This chapter will focus on various loosely connected methods and algorithms which were implemented. These cover topics such as manipulation of binary and integer values, Base64 encoding and decoding and iterator-based hash functions.

Many of these topics would normally be covered by a language's standard library, but are not yet covered by Motoko's.

4.1 Byte & bit manipulation

4.1.1 zeroLeadingBits

This method allows setting the n leading bits of a byte to zero. To do so a binary AND operation is performed with a bit mask where those bits which should be zeroed are 0, while the others are 1. Consider an example of zeroing the first 5 bits of a byte with value $(93)_{10} = (01011101)_2$:

$$\begin{aligned} &(01011101)_2 \\ \text{AND } &(00000111)_2 \\ =&(00000101)_2 \end{aligned}$$

Such a bit mask can be created by performing a logical right-shift of 0xFF by x digits:

$$(11111111)_2 \text{ RSHIFT } 5 = (00000111)_2$$

The implementation in Motoko had to further compensate for Motoko's logical right-shift operation masking the amount of digits by which to shift to 8.

```
public func zeroLeadingBits(b : Word8, n : Nat) : Word8 {
  if (n > 8) {
    Debug.print("zeroLeadingBits: n > 8");
    assert false;
  };

  // Motoko's bit shifts on WordN wrap around after N shifts, eg 0xFF >> 8 =
  // 0xFF.
  // This seems to be implemented akin to how eg the `SHL` command of the x86
  // architecture works, by masking the shift amount to `N - 1` bits. While
  // this does not conform to the intuition of logical shifts, we need to
  // work around it none the less.
  if (n == 8) {
    return 0 : Word8;
```

```
  };

  // To zero the n leftmost bits of a byte, AND it with a mask where the
  // leftmost n bits are 0, while the other bits are 1.
  // Such a mask can be constructed as 0xFF shifted right by however many
  // bits you want to zero. Eg 0xFF >> 3 = 0b 00 01 11 11.
  let zeroMask : Word8 = 0xFF >> Word8.fromNat(n);
  return b & zeroMask;
};
```

4.1.2 getLeadingBits

This method allows accessing the value of the x leading bits of a byte. To do so a binary AND operation is performed with a bit mask where those bits which should be accessed are 1, while the others are 0. Consider an example of accessing the first 3 bits of a byte with value $(57)_{10} = (00111001)_2$.

$$\begin{aligned} &(00111001)_2 \\ \text{AND } &(11100000)_2 \\ =&(00100000)_2 \end{aligned}$$

Such a bit mask can be created as the difference of 0xFF and 0xFF right-shifted by x digits:

$$(11100000)_2 = (11111111)_2 - ((11111111)_2 \text{ RSHIFT } 3)$$

As is the case for zeroLeadingBits, the implementation in Motoko has to further compensate for Motoko's logical right-shift operation masking the amount of digits by which to shift to 8.

```
public func getLeadingBits(b : Word8, n : Nat) : Word8 {
  if (n > 8) {
    Debug.print("getLeadingBits: n > 8");
    assert false;
  };

  // Motoko's bit shifts on WordN wrap around after N shifts, eg 0xFF >> 8 =
  // 0xFF.
  // This seems to be implemented akin to how eg the `SHL` command of the x86
  // architecture works, by masking the shift amount to `N - 1` bits. While
  // this does not conform to the intuition of logical shifts, we need to
  // work around it none the less.
  if (n == 8) {
    // b & 0x00 = 0 forall b
    return 0 : Word8;
  };

  // To extract the n leftmost bits of a byte, AND it with a mask where the
  // leftmost n bits are 1, while the other bits are 0.
  let mask : Word8 = 0xFF - (0xFF >> Word8.fromNat(n));
  return b & mask;
};
```

4.1.3 xor

This method allows performing a byte-wise XOR with two byte arrays of equal size. The implementation ensures that the two arrays are of equal size, then iterates over them.

```
public func xor(a : [Word8], b : [Word8]) : [Word8] {
  if (a.size() != b.size()) {
    Debug.print("xor: a.size() != b.size()");
```

```
    assert false;
  };

  return Array.tabulate<Word8>(
    a.size(),
    func(i : Nat) : Word8 {
      return a[i] ^ b[i];
    },
  );
};
```

4.2 Integer manipulation

4.2.1 ceilDiv

This method implements ceiling division of two natural numbers:

$$z = \left\lceil \frac{x}{y} \right\rceil$$

As Motoko's support for conversions between different numerical types is limited, this method was implemented without the use of floating-point arithmetic.

```
func ceilDiv(x : Nat, y : Nat) : Nat {
  var z = x / y;
  if (x % y != 0) {
    z += 1;
  };

  return z;
};
```

4.3 Hash utilities

While Motoko provides no built-in support for industry-standard hash functions, there exists an open-source community-maintained implementation of SHA-256 which was used [Hau20b].

4.3.1 hashFromIter

While the utilised implementation of SHA-256 supports iteratively building up a hash by repeatedly consuming byte arrays, it has no support for Motoko's iterator class. This iterator class — similar to iterators in Python or Java — repeatedly yields an unspecified number of elements before running empty. They can be used in any place where data should be processed without being first fully loaded into memory, such as might be the case with hashing a big file.

This method repeatedly consumes up to 4096 bytes from the iterator and writes them to the hash function. Once the iterator is empty, it will return the hash digest.

```
func hashFromIter(it : Iter.Iter<Word8>) : [Word8] {
  // We'll feed data to the SHA256 implementation in 4KB batches.
  let batchSize = 4096;
  var batch : [var Word8] = Array.init<Word8>(batchSize, 0);
  let digest = SHA256.Digest();
  var count : Nat = 0;

  label consumeIterator loop {
    while (count < batchSize){
      switch (it.next()) {
        case (?word) {
```

```
          batch[count] := word;
          count += 1;
        };
        case null {
          // No more data left to read in iterator
          break consumeIterator;
        };
      };
    };

    // Full batch could be read from iterator
    digest.write(Array.freeze<Word8>(batch));
    count := 0;
  };

  // `batch` will now contain the last `count` bytes which were read from the
  // iterator before it ran dry, write those to the digest.
  if (count > 0) {
    digest.write(
      Array.tabulate<Word8>(
        count,
        func(i : Nat) : Word8 {
          return batch[i];
        },
      ),
    );
  };

  return digest.sum();
};
```

4.4 Base64 encoding & decoding

The Base64 data encoding represents arbitrary binary data as printable ASCII characters. It is used in any place where binary information has to be exchanged via a channel only supporting printable characters. Examples of such communication channels are email or terminals.

The 64 in its name refers to supporting a total of 64 printable ASCII characters, allowing each character to encode $\log_2(64) = 6$ bits. As each ASCII character is generally encoded as one byte, this leads to an overhead of $33\,\%$.

While not required for creating and verifying RSA signatures, Base64 serves as a convenient way to exchange binary data — such as signatures — with the IC by means of a terminal during testing. As Motoko's standard library offers no support for Base64 encodings, support according to RFC 4648 was implemented [Jos06].

4.4.1 Base64 alphabets

RFC 4648 defines two alphabets for a Base64 encoding, shown in table 4.1. They differ in two encoding characters, with the "base64url" alphabet being chosen such that it is safe for use in a URL or filename.

4.4.2 Encoding & padding

Base64 encodes blocks of 24 bits to $24/6 = 4$ characters each. If the final block of inputs is less than 24 bits, behaviour is as follows. Note how, as inputs must be byte sequences, there are only two cases to consider:

- If the last block is 8 bits, it is zero-padded on the right side to 12 bits. The final encoded output then consists of two encoding characters and two padding characters.

0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	b	45	t	62	+
12	M	29	b	46	u	63	/
13	N	30	b	47	v		
14	O	31	b	48	w	pad	=
15	P	32	b	49	x		
16	Q	33	b	50	y		

(a) 'base64' alphabet

0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	b	45	t	62	-
12	M	29	b	46	u	63	_
13	N	30	b	47	v		
14	O	31	b	48	w	pad	=
15	P	32	b	49	x		
16	Q	33	b	50	y		

(b) 'base64url' alphabet

Table 4.1. Base64 alphabets

- If the last block is 16 bits, it is zero-padded on the right side to 24 bits. The final output then consists of three encoding characters and one padding character.

4.4.3 Implementation in Motoko

Base64 alphabets

The choice of alphabets is implemented as a Motoko variant type, which can be passed into the encoding respective decoding functions.

```
public type Encoding = {
  // Standard encoding according to RFC4648, section 4.
  // Uses `+` and `/` for the 62nd and 63rd characters.
  #standard;
  // Encoding according to RFC4648, section 5.
  // Uses `-` and `_` for the 62nd and 63rd characters.
  #url;
};
```

Encoding & decoding sextets

The following two methods implement the encoding of a sextet — six bits — to one Base64 character as well as the corresponding decoding operation. The sextets are passed in the lower six bits of a Word8 (byte) type, with the encoding method aborting if the higher two bits are non-zero, and the decoding method returning an error if an invalid character is encountered.

```
func sextetToChar(b : Word8, enc : Encoding) : Char {
  return switch(b) {
    case (0) 'A';
    case (1) 'B';
    // Truncated for brevity.
    case (61) '9';
    case (62) {
      switch (enc) {
```

```
        case (#standard) '+';
        case (#url) '-';
      };
    };
    case (63) {
      switch (enc) {
        case (#standard) '/';
        case (#url) '_';
      };
    };
    case (_) {
      Debug.print("Invalid sextet passed to Base64 encoding: " # Word8.toText(b));
      Prelude.unreachable();
    };
  };
};

func charToSextet(c : Char) : Result.Result<Word8, Text> {
  let b : Word8 = switch(c) {
    case ('A') 0;
    case ('B') 1;
    // Truncated for brevity
    case ('9') 61;
    case ('+' or '-') 62;
    case ('/' or '_') 63;
    // Strictly speaking not a zero-byte, but the absence of a byte. Doing it
    // this way allows easier parsing, with the superfluous bytes compensated
    // for in the decoding routine.
    case ('=') 0;
    // Abusing an impossible value as error indicator like it's 1985.
    case (_) 255;
  };

  // While this C-style error handling is nasty, the alternatives are to:
  // - Either have an #ok() respectively #err() in each of the cases above
  // - Have the switch have a multivariate return (ie Word8 or error) which
  //   we could differentiate here - which I don't think is possible.
  if (b == (255 : Word8)) {
    return #err("Invalid Base64 character: '" # Char.toText(c) # "'");
  } else {
    return #ok(b);
  };
};
```

Encoding Base64

The following method accepts a byte array and choice of Base64 alphabet and encodes it as a Base64 string. It repeatedly consumes up to three bytes from the input — optionally padding with zero-bytes if required — and extracts four sextets. These are then encoded to their respective Base64 characters and added to the output buffer.

```
public func encode(m : [Word8], enc : Encoding) : Text {
  // m.length / 3 will perform integer division, so the buffer might be 1 - 3
  // chars too small, but will grow so automatically if needed.
  let buf : Buffer.Buffer<Char> = Buffer.Buffer<Char>(m.size() / 3 * 4);

  var paddingCount = 0;
  var i : Nat = 0;
  while (i < m.size()) {
    // If there is one or two lone bytes at the end, encoding will happen as
    // if there was an appropriate amount of zero-bytes following. In the
    // end, the characters which are too much in the output will be replaced
```

```
  // by padding characters.

  let b1 : Word8 = m[i];
  var b2 : Word8 = 0x00;
  var b3 : Word8 = 0x00;

  if (i + 1 < m.size()) {
    b2 := m[i + 1];
  } else {
    paddingCount += 1;
  };

  if (i + 2 < m.size()) {
    b3 := m[i + 2];
  } else {
    paddingCount += 1;
  };

  // let x_i,j be the j-th bit (starting at 1 for the least-significant
  // bit) of the i-th byte (starting at 1 for the first to 3 for the third)
  // byte.

  // (0b) 0 0 x_1,8 x_1,7 x_1,6, x_1,5 x_1,4 x_1,3
  buf.add(
    sextetToChar(
      (b1 & 0xFC) >> 2,
      enc,
    ),
  );

  // (0b) 0 0 x_1,2 x_1,1 x_2,8, x_2,7 x_2,6 x_2,5
  buf.add(
    sextetToChar(
      ((b1 & 0x03) << 4) | ((b2 & 0xF0) >> 4),
      enc,
    ),
  );

  // (0b) 0 0 x_2,4 x_2,3 x_2,2, x_2,1 x_3,8 x_3,7
  buf.add(
    sextetToChar(
      ((b2 & 0x0F) << 2) | ((b3 & 0xC0) >> 6),
      enc,
    ),
  );

  // (0b) 0 0 x_3,6 x_3,5 x_3,4, x_3,3 x_3,2 x_3,1
  buf.add(
    sextetToChar(
      (b3 & 0x3F),
      enc,
    ),
  );

  i += 3;
};

// Replace `A`s at the end, which are the result of padded 0-bytes, by the
// padding char `=`.
// Mind that Iter.range(x, y) includes both x and y.
for (i in Iter.range(0, paddingCount - 1)) {
  buf.put(buf.size() - i - 1, '=');
};
```

```
  return Text.fromIter(buf.vals());
};
```

Decoding Base64

The following method accepts a Base64 encoded text and decodes it to a byte array. It repeatedly consumes four characters of inputs, keeping track of how many padding characters there were. It then converts each character to a sextet, and combines the four sextets to three bytes which are added to the output.

If the decoding operation encounters a non-padded input the length of which isn't a multiple of four, or an invalid character, it aborts.

The decoding operation is able to handle mixed alphabets, as any possible encoding character over the union of both alphabets can be decoded to only one possible sextet.

```
public func decode(m : Text) : Result.Result<[Word8], Text> {
  // Buffer might be marginally too small due to integer division, but will
  // grown on its own if needed.
  let buf : Buffer.Buffer<Word8> = Buffer.Buffer<Word8>(m.size() / 4 * 3);
  var i : Nat = 0;
  let iter = Text.toIter(m);
  let chars = Iter.toArray<Char>(iter);
  var paddingCount : Nat = 0;

  if (chars.size() % 4 != 0) {
    return { #err("Base64 value did not contain multiple of 4 characters. Only
        ↳padded values are supported." )};
  };

  while (i < chars.size()) {
    // We only accept padded Base64, so are guaranteed that there is a
    // multiple of four chars in the text.
    let c1 : Char = chars[i];
    var c2 : Char = chars[i + 1];
    var c3 : Char = chars[i + 2];
    var c4 : Char = chars[i + 3];

    if (c2 == '=') {
      paddingCount += 1;
    };

    if (c3 == '=') {
      paddingCount += 1;
    };

    if (c4 == '=') {
      paddingCount += 1;
    };

    let s1 = switch(charToSextet(c1)) {
      case (#ok(sextet)) sextet;
      case (#err(e)) return { #err(e) };
    };

    let s2 = switch(charToSextet(c2)) {
      case (#ok(sextet)) sextet;
      case (#err(e)) return { #err(e) };
    };

    let s3 = switch(charToSextet(c3)) {
      case (#ok(sextet)) sextet;
```

```
      case (#err(e)) return { #err(e) };
    };

    let s4 = switch(charToSextet(c4)) {
      case (#ok(sextet)) sextet;
      case (#err(e)) return { #err(e) };
    };

    let b1 = Word32.fromNat(Word8.toNat(s1));
    let b2 = Word32.fromNat(Word8.toNat(s2));
    let b3 = Word32.fromNat(Word8.toNat(s3));
    let b4 = Word32.fromNat(Word8.toNat(s4));

    // Quite simple, as we can rely on the two highest bits being 0.
    let x : Word32 =
      b1 << 18 |
      b2 << 12 |
      b3 << 6  |
      b4
    ;

    buf.add(Word8.fromNat(Word32.toNat(x >> 16)));
    buf.add(Word8.fromNat(Word32.toNat(x >> 8)));
    buf.add(Word8.fromNat(Word32.toNat(x)));

    i += 4;
  };

  // Remove `0x00`s at the end, which are the result of padded `A` chars.
  // Mind that Iter.range(x, y) includes both x and y.
  for (i in Iter.range(0, paddingCount - 1)) {
    let _ = buf.removeLast();
  };

  return { #ok(buf.toArray()) };
};
```

Chapter 5

Implementation of RSA signatures conformant to PKCS#1 in Motoko

This chapter will focus on the implementation of RSA signatures conformant to PKCS#1 in Motoko. It will provide an overview of the building blocks defined by PKCS#1 as well as their implementation in Motoko

Building blocks are covered bottom-up, starting with the most generic ones, finishing with the signature schemes which utilise them.

Additionally a brief overview of the public interface of the library, as it might be used by an external developer, is provided. Lastly compatibility of this implementation with OpenSSL is demonstrated, providing evidence that the implementation is correct.

Lastly performance of this library will be evaluated, analyzing the time it takes for signature creation and verification for various key sizes.

Unless noted otherwise, all of the algorithms discussed in this chapter are based on version 2.2 of PKCS#1, published as RFC 8017 [Mor+16].

5.1 Terminology

The following aims to define terminology and variable identifiers as used throughout PKCS#1.

5.1.1 Variables

RSA modulus n

RSA public exponent e

RSA private exponent d

First CRT exponent dP

Second CRT exponent dQ

CRT coefficient $qInv$

RSA public key (n, e)

RSA private key K $(p, q, dP, dQ, qInv)$

Message representative, m Integer representation of a message. $0 \leqslant m \leqslant n - 1$

Message, M Big-endian byte string representation of a message.

Signature representative, s Integer representation of a signature. $0 \leqslant s \leqslant n - 1$

Signature, S Big-endian byte string representation of a signature.

5.1.2 General terms

Data conversion primitives Functions which convert between integer and byte string representations of values.

Signature and verification primitives Functions which define how to apply the RSA function to integer values, to convert between signature and message representatives.

Mask generation functions Deterministic functions which take a variable-length input string and an output length, and produce a pseudorandom output of the requested length.

Signatures with appendix Signatures which do not contain the message which they signed.

Encoding methods for signatures with appendix Functions which pre-process a message such that it is suitable for signing with the RSA function. This involves operations such as hashing and padding, encoding meta-information, and introducing elements of non-determinism.

5.2 Overview

Figure 5.1 serves as an overview of how the various parts, which will be discussed in this chapter, are combined to produce two signature schemes. Rounded rectangles represent byte strings, regular rectangles represent integers and circles represent booleans. The two signature schemes differ only in an scheme-specific encoding method for signature creation, and a scheme-specific validation method for signature verification.

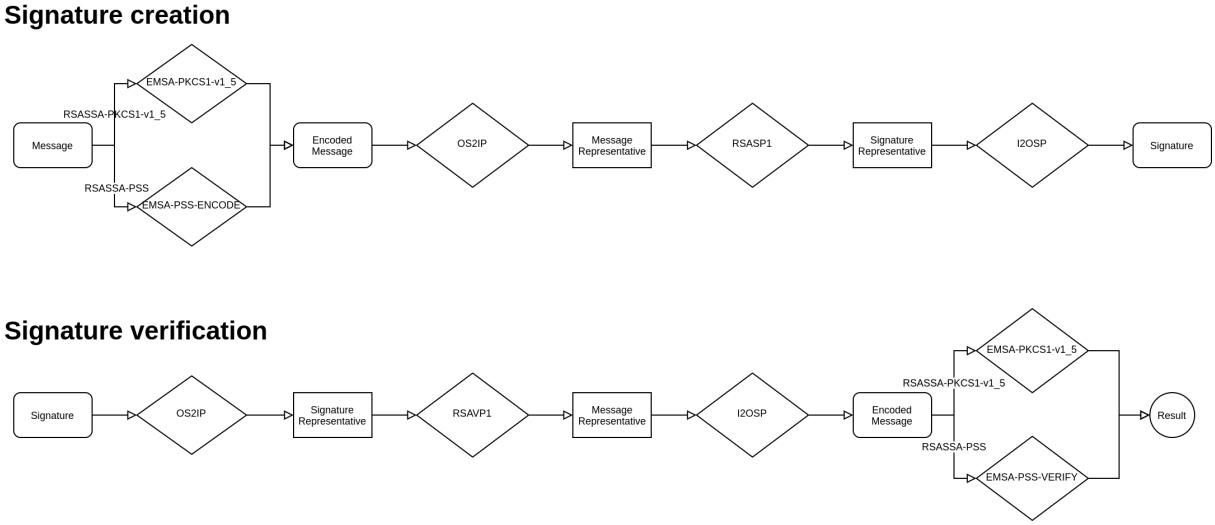

Figure 5.1. PKCS#1 overview

5.3 Public & private key types

The library supports RSA keys based on the description in Section 3 of PKCS#1 [Mor+16]. The public key format is shown in algorithm 3, the private key format in algorithm 4. dP, dQ and $qInv$ allow improving performance of modular exponentiation in private-key operations with the CRT, as described in Section 3.2.

Algorithm 3 Public key

RSA modulus: Integer n
RSA public exponent: Integer e

Algorithm 4 Private key

Variables

First RSA prime: Integer p
Second RSA prime: Integer q
First prime's CRT exponent: Integer dP
Second prime's CRT exponent: Integer dQ
CRT coefficient: Integer $qInv$

Requirements

$e \cdot d \equiv 1 \pmod{\lambda(n)}$
$e \cdot dP \equiv 1 \pmod{p-1}$
$e \cdot dQ \equiv 1 \pmod{q-1}$
$q \cdot qInv \equiv 1 \pmod{p}$

5.3.1 Implementation in Motoko

Implementation is verbatim based on the specification in algorithm 4. In addition to the standard the bit length of the RSA modulus is stored, as it is utilized in various algorithms.

```
public type PublicKey = {
  // RSA modulus
  n : Nat;
  // RSA public exponent
  e : Nat;
  // Bit length of RSA modulus
  modulusBits : Nat;
};

public type PrivateKey = {
  // RSA prime 1
  p : Nat;
  // And its CRT exponent
  dP : Nat;

  // RSA prime 2
  q : Nat;
  // And its CRT exponent
  dQ : Nat;

  // First CRT coefficient
  qInv : Nat;

  // RSA private exponent
  d : Nat;
  // Bit length of RSA modulus
  modulusBits : Nat;
};
```

5.3.2 Support for multi-prime RSA

PKCS#1 supports multi-prime keys since version 2. With such keys the RSA modulus n is the product of multiple primes $n = p_1 \cdot \ldots \cdot p_k$. For a modulus which is a product of k factors, the speedup achievable by application of the CRT over naive modular exponentiation approaches k^2. As such, multi-prime RSA

can significantly improve the speed of private-key operations.

While support for multi-prime RSA was not added to the library, doing so would be fairly straightforward. Changes would have to be made to only the private key type and the "RSASP1" signature creation primitive.

5.3.3 Key generation

Key generation was not implemented in the library due to a lack of support for secure randomness in Motoko. Once available, key generation can be trivially added. Section 3 of PKCS#1 defines the required properties of RSA private and public keys. Based on two randomly chosen primes and a public exponent e, the private exponent d as well as all CRT exponents and the CRT coefficient can be easily derived using the extended euclidean algorithm and basic arithmetic operations.

For testing purposes, keys can be generated in other libraries, such as OpenSSL, and imported. Commands to generate a 1024-bit key with OpenSSL are shown for completeness.

```
# Generate RSA-PKCS1-v1_5 keypair
openssl genpkey -algorithm rsa -pkeyopt rsa_keygen_bits:1024 -out private.key
openssl rsa -in private.key -pubout -out public.key

# Generate RSASSA-PSS keypair
openssl genpkey -algorithm rsa-pss -pkeyopt rsa_keygen_bits:1024 -out private.key
openssl rsa -in private.key -pubout -out public.key
```

Of note is the '-algorithm rsa' vs '-algorithm rsa-pss' parameter. While the keys themselves do not differ between the two schemes, this information is encoded as part of the ASN.1 PrivateKeyInfo meta information. Libraries such as OpenSSL use this to specify that a given key is only intended to be used with one specific signature scheme. This prevents weaknesses in one scheme affecting signatures made with another [Squ19].

5.3.4 Key exchange formats

Standards such as PKCS#8 formalise binary key formats and are supported by libraries like OpenSSL [Tur10]. No support for these formats was implemented in the Motoko library however.

5.4 I2OSP: Integer to big-endian byte-string

The "I2OSP" data conversion primitive, show in Algorithm 5, encodes any positive integer as a zero-padded big-endian byte string of requested length. It is used in e.g. the conversion of signature representatives to signatures.

Algorithm 5 I2OSP

Input

Number: Integer x
Encoded length: Integer n

Algorithm

If $x \geqslant 256^n$:
Return error
$x =: x_{n-1} * 256^{n-1} + x_{n-2} * 256^{n-2} + \ldots + x_1 * 256 + x_0$
Return $\langle x_{n-1}, x_{n-2}, \ldots, x_1, x_0 \rangle$

5.4.1 Implementation in Motoko

An array of desired length is initiated with bytes of value zero, ensuring that outputs are zero-padded. The common division-with-rest algorithm is then used to calculate a base-256 representation of the input. The output array is filled starting at the rightmost index to achieve the required big-endianness.

If the input cannot be represented as a byte array of length $xLen$, an error is returned. Otherwise an immutable copy of the resulting byte array is returned.

```
public func primI2OSP(x : Nat, xLen : Nat) : Result.Result<[Word8], Text> {
  let out : [var Word8] = Array.init<Word8>(xLen, 0x00);
  let base : Nat = 2**8;
  var val = x;
  // Algorithm below generates bytes starting at LSB, while we want a
  // big-endian representation.
  var i : Nat = xLen;

  while (val != 0) {
    if (i == 0) {
      return { #err("Integer too large for byte array of length " # Nat.toText(xLen)
          ↳) };
    };
    i -= 1;

    // WordN employs modular arithemtics, so will lead to the calculation of
    // `out[i] := val mod 256` implicitly.
    out[i] := Word8.fromNat(val);
    val := val / base;
  };

  return { #ok(Array.freeze<Word8>(out)) };
};
```

5.5 OS2IP: Big-endian byte string to integer

The "OS2IP" data conversion primitive, shown in Algorithm 6, decodes a big-endian byte string to its integer representation. It is used in e.g. the conversion of messages to message representatives.

Algorithm 6 OS2IP

Input

Input: Byte string $\langle x_n, x_{n-1}, \ldots, x_1, x_0 \rangle$

Algorithm

Return $x_n * 256^n + x_{n-1} * 256^{n-1} + \ldots + x_1 * 256 + x_0$

5.5.1 Implementation in Motoko

The integer value is calculated by iterating over all bytes in the input, summing up the terms. As the input is big-endian, the first byte has value $256^{length-1}$.

```
public func primOS2IP(x : [Word8]) : Nat {
  var out : Nat = 0;

  // Mind that the input is big-endian, so the first byte has value
  // 256**(x.size() - 1)
  for (idx in Iter.range(0, x.size() - 1)) {
    let power : Nat = x.size() - idx - 1;
    out += Word8.toNat(x[idx]) * 256**power;
```

```
  };

  return out
};
```

5.6 RSASP1: Signature creation with RSA

The "RSASP1" signature primitive, shown in Algorithm 7, describes how to, given a private key, apply the RSA function to a message representative to calculate the corresponding signature representative. As shown in the overview, the message is already padded at this time. This primitive merely implements the forward direction of the RSA function.

The CRT as described in Chapter 3 is applied to improve the performance of the modular exponentiation operations.

Algorithm 7 RSASP1

Input

Private key: $K = (p, q, dP, dQ, qInv)$
Message representative: Integer m

Algorithm

If $m < 0$ **or** $m \geqslant n$
Return error

$s_1 := m^{dP} \mod p$
$s_2 := m^{dQ} \mod q$
If $s_2 \leqslant s_1$
$h := (s_1 - s_2) \cdot qInv \mod p$
Else
$h := ((s_1 + \left\lceil \frac{q}{p} \right\rceil) \cdot p - s_2) \cdot qInv \mod p$

Return $s_2 + q \cdot h$

5.6.1 Implementation in Motoko

The signing direction of the RSA function $s = m^d \mod n$ is implemented using the CRT according to Algorithm 1. An error is returned if the message representative is too big for the RSA modulus.

As Motoko does not support conversion from its natural-number type to its integer type, care must be taken not to cause a natural-number underflow when $s_2 > s_1$. When this is the case, h is calculated as described by Schneier [Sch15] instead, by adding a sufficiently large value such that it will be positive.

```
public func primRSASP1(k : PrivateKey, m : Nat) : Result.Result<Nat, Text> {
  let n = k.p * k.q;
  if (m >= n) {
    return { #err("Message representative out of range, must be smaller than modulus
        ↳") };
  };

  let s1 = Bignum.modPow(m, k.dP, k.p);
  let s2 = Bignum.modPow(m, k.dQ, k.q);

  // As we can't get from a Nat to an Int (Motoko...), we have to handle the
  // case of s2 > s1. In this case add an appropriate value such that the
  // difference is positive, without affecting the final result to the modulo
  // calculation.
  let h = if (s2 <= s1) {
    ((s1 - s2) * k.qInv) % k.p;
  } else {
    (((s1 + ceilDiv(k.q, k.p) * k.p) - s2) * k.qInv) % k.p;
  };

  return #ok(s2 + k.q * h);
};
```

5.7 RSAVP1: Signature verification with RSA

The "RSAVP1" verification primitive, shown in Algorithm 8, describes how to, given a private key, apply the RSA function to a signature representative to calculate the corresponding message representative. As shown in the overview this primitive merely handles the backwards direction of the RSA function, the scheme-specific validation will happen subsequently.

Algorithm 8 RSAVP1

Input

Public key: (n, e)
Signature representative: Integer s

Algorithm

Return $s^e \mod n$

5.7.1 Implementation in Motoko

The verification direction of the RSA function is implemented as $m = s^e \mod n$, using big-number modular arithmetic. An error is returned if the signature representative is too big for the RSA modulus.

```
if (s >= k.n) {
  return { #err("Signature representative out of range, must be smaller than modulus
      ↳") };
};

return { #ok(Bignum.modPow(s, k.e, k.n)) };
```

5.8 MGF1: Mask generation function

The "MGF1" mask generation function, shown in Algorithm 9, is the sole mask generation function defined by the standard. Given a hash function and its corresponding hash length, as well as an input byte string and output length it generates a pseudorandom output of desired length.

Each iteration of the algorithm will cause the output size to grow by $hLen$ bytes, so that for example the generation of a 256 byte output using a 32 byte hash function takes eight iterations.

The hard limitation of output length to $2^{32}hLen$ is due to the internal counter i being represented as a byte string of length four. This limitation has little practical impact, as with a hash length of 32 bytes this already permits generation of outputs with a length exceeding 10^{11} bytes.

Algorithm 9 MGF1

Input

Input: Byte string *mgfSeed*
Output length: Integer *maskLen*
Hash function: *Hash* := SHA-256
Hash length: Integer *hLen* := 32

Algorithm

If $maskLen > 2^{32}hLen$
Return error
T := $\langle\rangle$
For *i* := 0 **to** $\lceil maskLen/hLen \rceil - 1$
C := *I2OSP*(i, 4)
T := *T* || *Hash*(*mgfSeed* || *C*)
Return first *maskLen* bytes of *T*

5.8.1 Implementation in Motoko

The Motoko implementation first checks if the requested output length exceeds the allowed range, and aborts if so.

As we only support the SHA256 hash function, the choice of a hash function as well as its hash length — 32 bytes — is hardcoded. The output array is then built up by repeatedly hashing the concatenation of the seed and the counter as per the algorithm, and finally an output of requested length is returned.

```
public func mask(seed : [Word8], maskLength : Nat) : Result.Result<[Word8], Text> {
  // This prevents an overflow of our 4-byte counter in the loop below.
  if (maskLength > 2**32 * hashLength) {
    return { #err("Mask too long") };
  };

  // Could be made more efficient with mutable array pre-allocated to desired
  // size, but given the usual number of iterations (256 => 2048 bits, so 8)
  // this will work just fine.
  var t : [Word8] = [];

  let lastIteration = ceilDiv(maskLength, hashLength) - 1;
  for (i in Iter.range(0, lastIteration)) {
    switch (primI2OSP(i, 4)) {
      case (#ok(c)) {
        let hash = SHA256.sha256(
          Array.append<Word8>(seed, c),
        );

        t := Array.append<Word8>(t, hash);
      };
      case (#err(c)) {
        // The check above will ensure that lastIteration <= 2**32 - 1, so is
        // guaranteed to fit in 4 bytes.
        Debug.print("ERROR: Counter too large, conversion to 4-byte number failed.")
            ↳;
        Prelude.unreachable();
```

```
      };
    };
  };

  // t now has size ceil(maskLength / hashLength) * hashLength >= maskLength.
  // As such we can simply return the first maskLength bytes.
  t := Array.tabulate<Word8>(
    maskLength,
    func(i : Nat) : Word8 {
      return t[i];
    },
  );

  return { #ok(t) };
};
```

5.9 EMSA-PKCS1-v1_5: Deterministic encoding method

The 'EMSA-PKCS1-v1_5' encoding method is used as part of the 'RSASSA-PKCS-v1_5' signature scheme. Its operation is visualized in Figure 5.2 and described in Algorithm 10.

This encoding method serves three main uses. Firstly the hashing of the message allows to sign messages of arbitrary sizes — or at least up to the limit of the used hash function. Secondly it embeds information about which hash algorithm was used to allow the recipient to verify the signature, and thirdly it pads the message such that a size suitable for RSA can be achieved irrelevant of the size of the hash.

This algorithm deviates from the standard in two points:

- The message hash rather than the message is passed into the function, see Section 5.14.1.
- The choice of hash function is fixed, see Section 5.14.2.

Algorithm 10 EMSA-PKCS1-v1_5

Input

Message hash: Byte string *mHash*
Encoded message length: Integer *emLen*
Hash function: *Hash* := SHA-256
Hash length: Integer *hLen* := 32

Algorithm

If *emLen* < *tLen* + *11*
Return error
T := *DER-Encode*(*Hash*, *mHash*)
PS := Byte string of *emLen* - |*T*| - *3* bytes of value 0xFF
Return 0x00 || 0x01 || *PS* || 0x00 || *T*

5.9.1 DER-Encode

The DER-Encode function encodes the following ASN.1 structure using the DER encoding. The relevant ASN.1 definitions can be found in appendices A.2.4 and B.1. A set of precomputed values for every supported hash algorithm is found in Section 9.2 of PKCS#1. The embedded hash identifier allows the recipient to determine which hash algorithm was used to generate the signature.

```
DigestInfo ::= SEQUENCE {
                   digestAlgorithm AlgorithmIdentifier,
```

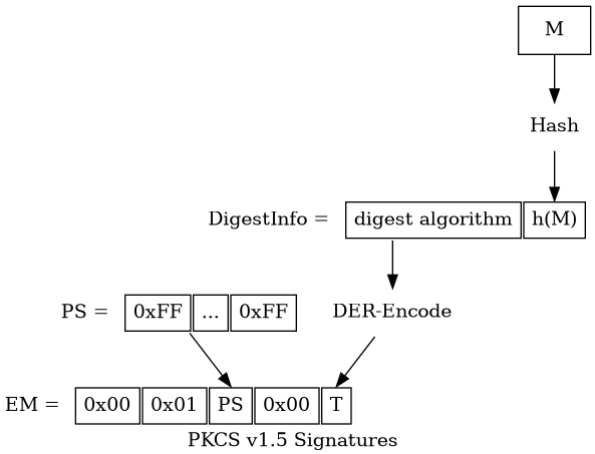

Figure 5.2. EMSA-PKCS1-v1_5 encoding operation

```
        digest OCTET STRING
    }
```

5.9.2 Implementation in Motoko

As we only support the SHA-256 hash algorithm, the implementation of 'EMSA-PKCS1-v1_5' can be simplified a lot. Except for the actual hash value, the whole ASN.1 structure can be hardcoded. This alleviates the need for an ASN.1 parser & DER encoder.

The concatenation of pre-defined DER-encoded ASN.1 structure and hash value is then padded according to the algorithm, and the result returned.

```
public func pkcs15Encode(mHash : [Word8], encodedLength : Nat) : Result.Result<[
    ↳Word8], Text> {
  // DER-encoded ASN.1 `DigestInfo` type. As we only support SHA256, the
  // `digestAlgorithm` parameter is hardcoded according to PKCS #1 v2.2,
  // section 9.2, 1st note.
  let digestInfo : [Word8] = Array.append<Word8>(
    [0x30, 0x31, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0
        ↳x02, 0x01, 0x05, 0x00, 0x04, 0x20],
    mHash,
  );

  if (encodedLength < digestInfo.size() + 11) {
    return { #err("Intended encoded message length too short") };
  };

  // This padding is the combination of the `PS` octet string and the three
  // magical bytes according to the standard.
      // An array of appropriate size is initiated with 0xFF - corresponding to
      // `PS` - and the three magical bytes are then set manually.
  let padding = Array.init<Word8>(encodedLength - digestInfo.size(), 0xFF);
  padding[0] := 0x00;
  padding[1] := 0x01;
  padding[padding.size() - 1] := 0x00;

  // em = 0x00 || 0x01 || PS || 0x00 || T
  let em = Array.append<Word8>(
    Array.freeze<Word8>(padding),
    digestInfo,
  );

  return { #ok(em) };
```

```
};
```

5.10 EMSA-PSS-ENCODE: Probabilistic encoding method

The "EMSA-PSS-ENCODE" encoding is used as part of the signing direction of the "RSASSA-PSS" probabilistic signature scheme. Its operation is visualized in Figure 5.3 and described in algorithm 11.

This encoding method serves three main uses. Like 'EMSA-PKCS-v1_5' the hashing operation allows signing messages of arbitrary sizes, and it supports padding the signature to a desired size. Lastly it introduces an element of non-determinism, and embeds it in the final signature in a way that the recipient can verify the probabilistic signature.

This probabilistic salt ensures that two signatures of the same data with the same private key will be different. This non-determinism forms a part of the security proof of the "RSASSA-PSS" signature scheme as described in the original submission to IEEE [MP00].

This algorithm deviates from the standard in two points:

- The message hash rather than the message is passed into the function, see Section 5.14.1.
- The choice of hash function is fixed, see Section 5.14.2.

Algorithm 11 EMSA-PSS-ENCODE

Input

Message hash: Byte string *mHash*
Encoded message length in bits: Integer *emBits*
Hash function: *Hash* := SHA-256
Hash length: Integer *hLen* := 32
Salt length: Integer *sLen*

Algorithm

If *M* exceeds input limitation of *Hash*
Return error

emLen := $\lceil emBits/8 \rceil$
salt := Randomly chosen byte string of length *sLen*
M' := 0x00 00 00 00 00 00 00 00 || *mHash* || salt
H := *Hash*(M')
PS := Byte string of *emLen - sLen - hLen - 2* bytes of value 0x00
DB := *PS* || 0x01 || *salt*
dbMask := *MGF1*(*H*, *emLen - hLen - 1*)
maskedDB := *DB* $\oplus$ *dbMask*
Set leftmost $8 \cdot emLen - emBits$ bits of *maskedDB* to 0
EM := *maskedDB* || *H* || 0xBC
Return *EM*

5.10.1 Implementation in Motoko

The Motoko implementation follows the algorithm, with the exception of the choice of salt value. Due to Motoko's current lack of support for randomness the salt is chosen deterministically. As per Chapter three of the original PSS paper and Chapter 8.1 of PKCS#1 this does not break provable security, but merely makes the security proof less tight, reducing provable security to a similar level as of a full-domain hash [Mor+16] [MP00].

Support for probabilistic salt values can be added once Motoko supports secure randomness, by adjusting the 'generateSalt()' method.

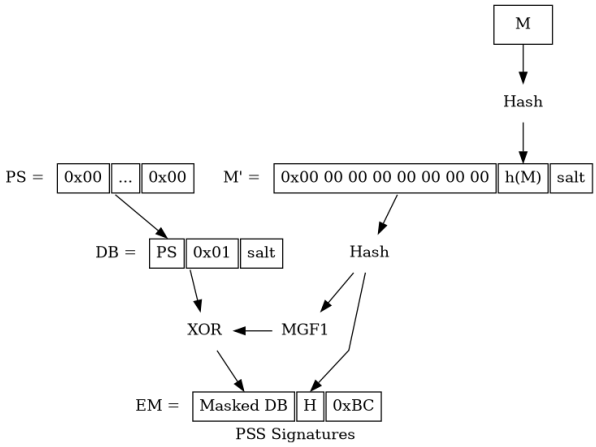

Figure 5.3. EMSA-PSS-ENCODE encoding operation

```
public func emsaPSSEncode(mHash : [Word8], encodedBitLength : Nat, saltLength : Nat)
    ↳ : Result.Result<[Word8], Text> {
  let encodedLength = ceilDiv(encodedBitLength, 8);

  if (encodedLength <  hashLength + saltLength + 2) {
    let msg = "Encoding error: Desired length of encoded message too short. " #
      Nat.toText(encodedLength) #
      " < " #
      Nat.toText(hashLength + saltLength + 2);
    return { #err(msg)};
  };

  // Mind that this salt is deterministic as of now. This makes the PSS security
  // proof less tight, reducing its security to about the one of a
  // full-domain hash.
  let salt : [Word8] = generateSalt(saltLength);

  // M' = 0x 00 00 00 00 00 00 00 00 || mHash || salt
  var mPrime = zeroByteArray(8);
  mPrime := Array.append<Word8>(mPrime, mHash);
  mPrime := Array.append<Word8>(mPrime, salt);
  assert mPrime.size() == 8 + hashLength + saltLength;

  let h = SHA256.sha256(mPrime);

  // Padding might be empty, if the desired encoded length is just sufficient.
  let padding = zeroByteArray(encodedLength - saltLength - hashLength - 2);

  // DB = padding || 0x01 || salt
  var db = Array.make<Word8>(0x01);
  db := Array.append<Word8>(padding, db);
  db := Array.append<Word8>(db, salt);
  assert db.size() == encodedLength - hashLength - 1;

  var dbMask : [Word8] = [];
  switch(mask(h, encodedLength - hashLength - 1)) {
    case (#ok(ary)) {
      dbMask := ary;
    };
    case (#err(e)) {
      return { #err(e) };
    };
  };
```

```
  // maskedDB = DB XOR dMask
  var maskedDB = Array.thaw<Word8>(xor(db, dbMask));

  // Set leftmost (8 * encodedLength - encodedBitLength) bits, of leftmost
  // byte in maskedDB, to zero. This ensures that the output will only have
  // non-zero bits in the encodedBitLength rightmost bits.
  maskedDB[0] := zeroLeadingBits(maskedDB[0], 8 * encodedLength - encodedBitLength);

  // EM = maskedDB || h || 0xbc
  var encoded = Array.append<Word8>(Array.freeze<Word8>(maskedDB), h);
  encoded := Array.append<Word8>(encoded, [0xbc]);

  return { #ok(encoded) };
};
```

5.11 EMSA-PSS-VERIFY: Probabilistic decoding method

The "EMSA-PSS-VERIFY" encoding is used as part of the verification direction of the "RSASSA-PSS" probabilistic signature scheme. Due to the non-determinism of this signature scheme, a dedicated verification encoding is required. Its operation is described in Algorithm 12.

In its first half it retrieves the salt encoded in the signature. Its second half is then identical to the first part of the "EMSA-PSS-ENCODE" encoding to reconstruct a signature with the now-deterministic salt.

This algorithm deviates from the standard in two points:

- The message hash rather than the message is passed into the function, see Section 5.14.1.
- The choice of hash function is fixed, see Section 5.14.2.

5.11.1 Implementation in Motoko

The Motoko implementation follows Algorithm 12. In addition it supports optionally detecting the salt length, which is described in more detail in Section 5.11.2.

```
public func emsaPSSVerify(mHash : [Word8], encodedMessage : [Word8],
    ↳encodedBitLength : Nat, saltLengthOrDefault : ?Nat) : Result.Result<Bool, Text>
    ↳ {
  let encodedLength = ceilDiv(encodedBitLength, 8);

  // We can only check this reliably if we know the salt length ahead of
  // time. Otherwise we'll assume a lower bound with a salt length of size.
  let minimalLength = switch(saltLengthOrDefault) {
    case null hashLength + 2;
    case (?saltLength) hashLength + 2 + saltLength;
  };
  if (encodedLength < minimalLength) {
    return { #err("Encoded message too short" ) };
  };

  if (encodedMessage[encodedMessage.size() - 1] != (0xBC : Word8)) {
    return { #err("Rightmost byte != 0xBC") };
  };

  let maskedDB = Array.tabulate<Word8>(
    encodedLength - hashLength - 1,
    func (i : Nat) : Word8 {
      return encodedMessage[i];
    },
  );
```

Algorithm 12 EMSA-PSS-VERIFY

Input

Message hash: Byte string *mHash*
Encoded message (retrieved from signature): Byte string *EM*
Encoded message length in bits: Integer *emBits*
Hash function: *Hash* := SHA-256
Hash length: Integer *hLen* := 32
Salt length: Integer *sLen*

Algorithm

emLen := $\lceil emBits/8 \rceil$
If *M* exceeds input limitation of *Hash*
    Return error
If $emLen < hLen + sLen + 2$
    Return error
If rightmost byte of *EM* != 0xBC
    Return error

mhash := *Hash*(M)
maskedDB := Leftmost $emLen - hLen - 1$ bytes of *EM*
H := The next *hLen* bytes of *EM*
If leftmost $8 \cdot emLen - emBits$ bits of *maskedDB* != 0
    Return error

dbMask := *MGF1*(*H*, *emLen - hLen - 1*)
DB := $maskedDB \oplus dbMask$
Set leftmost $8 \cdot emLen - emBits$ bits of *DB* to 0
If leftmost $emLen - hLen - sLen - 2$ bytes of *DB* != 0x00
or byte at position $emLen - hLen - sLen - 1$!= 0x01
    Return error
salt := The last *sLen* bytes of *DB*
M' := 0x00 00 00 00 00 00 00 00 || *mHash* || salt
H := *Hash*(M')
If *H' == H*
    Return success
Else
    Return error

```
// Hash of message & salt embedded in signature
let h = Array.tabulate<Word8>(
  hashLength,
  func (i : Nat) : Word8 {
    return encodedMessage[encodedLength - hashLength - 1 + i];
  },
);

// Leftmost 8 * encodedLength - encodedBitLength bits of leftmost byte must
// all be 0.
let zeroLeftmostBits : Nat = 8 * encodedLength - encodedBitLength;
if (getLeadingBits(maskedDB[0], zeroLeftmostBits) != (0 : Word8)) {
  return { #err(Nat.toText(zeroLeftmostBits) # " leftmost bits of maskedDB != 0")
     ↳};
};

let dbMask = switch(mask(h, encodedLength - hashLength - 1)) {
  case (#ok(x)) x;
  case (#err(e)) return { #err(e) };
};
var varDB = Array.thaw<Word8>(xor(maskedDB, dbMask));

// Zero leftmost 8 * encodedLength - encodedBitLength bits of leftmost byte
// of DB.
varDB[0] := zeroLeadingBits(varDB[0], zeroLeftmostBits);
let db = Array.freeze<Word8>(varDB);

// Detect salt length if needed
let saltLength = switch(saltLengthOrDefault) {
  case null {
    // Try detecting salt length
    switch (detectPSSSaltLength(db)) {
      case (#ok(x)) {
        x
      };
      case (#err(e)) {
        return { #err(e) };
      };
    };
  };
  case (?x) {
    // Salt length provided, no need to detect.
    x;
  };
};

// Leftmost `encodedLength - hashLength - saltLength - 2` bytes of DB are
// `PS` padding of 0-bytes.
// These are bytes `0` through `encodedLength - hashLength - saltLength - 3`
// as we are 0-indexed.
//
// With a max-size salt of `encodedLength - hashLength - 2` bytes there
// will be no padding, in which case the last padding byte
// (`encodedLength - hashLength - saltLength - 3`) would be `-1`, and would
// underflow.  As such we track the first non-padding byte (ie the magic
// 0x01 byte).
let firstNonPaddingByte = encodedLength - hashLength - saltLength - 2;
for (i in Iter.range(0, firstNonPaddingByte - 1)) {
  if (db[i] != (0 : Word8)) {
    // One-indexed bytes in error messages
    return { #err("Byte " # Nat.toText(i + 1) # " of PS padding of DB != 0x00") };
  };
};
```

```
  // The byte following the last `PS` padding byte must be `0x01`.
  if (db[firstNonPaddingByte] != (0x01 : Word8)) {
      return {
        #err(
          "Byte " #
          Nat.toText(firstNonPaddingByte + 1) # // 1-indexed bytes in error messages
          " of DB != 0x01"
        )
      };
  };

  // - PS is [0 .. firstNonPaddingByte - 1]
  // - Magic byte 0x01 is firstNonPaddingByte
  // - Salt is [firstNonPaddingByte + 1 .. firstNonPaddingByte + 1 + saltLength - 1]
  let salt = Array.tabulate<Word8>(
    saltLength,
    func (i : Nat) : Word8 {
      return db[firstNonPaddingByte + 1 + i];
    },
  );

  // M' = (0x00) * 8 || mHash || salt
  // Ie the reconstructed value of the encoding operation, assuming the
  // signature is for mHash.
  var mPrime = Array.append<Word8>(
    zeroByteArray(8),
    mHash,
  );
  mPrime := Array.append<Word8>(
    mPrime,
    salt,
  );

  let hPrime = SHA256.sha256(mPrime);

  if (Array.equal<Word8>(h, hPrime, Word8.equal)) {
    return { #ok(true) };
  } else {
    return { #err("Hash mismatch") };
  };
};
```

5.11.2 Detecting salt length

Recall the way in which the "EMSA-PSS-ENCODE" probabilistic encoding method encodes the salt, as described in Section 5.10. The relevant "DB" field consists of a variable-length padding of 0x00 bytes followed by a single 0x01 byte, followed by the salt.

While not mandated by the standard, this allows extracting the salt value from a signature without knowing the salt length. Such auto-detection is supported in libraries such as OpenSSL, and was also implemented in Motoko. Algorithm 13 describes how auto-detection is implemented.

Implementation in Motoko

Implementation in Motoko follows Algorithm 13. It iterates through the bytes of the "DB" field, discarding 0x00 bytes until the 0x01 byte is found. If so, the number of remaining bytes in "DB" is returned, otherwise an error is raised.

```
public func detectPSSSaltLength(db : [Word8]) : Result.Result<Nat, Text> {
  // At least the magic byte `0x01` must be present, padding and salt may be
  // empty.
```

Algorithm 13 Detect EMSA-PSS salt length

Input

Padded salt value: Byte string *DB*

Algorithm

If |*DB*| = 0:

Return error

Discard leading 0x00 bytes of *DB*

If |*DB*| < 1:

Return error

If *DB*[0] != 0x01:

Return error

Return |*DB*| - 1

```
  if (db.size() == 0) {
    return { #err("Unable to detect salt length: DB too short") };
  };

  var paddingCount : Nat = 0;
  // Padding can be at most up to the second-last byte, as the last byte has
  // to be the magic byte.
  // We also know that db.size() >= 1, so don't have to worry about an
  // out-of-bounds exception in the first iteration.
  while (db[paddingCount] == (0x00 : Word8)) {
    if (paddingCount == db.size() - 1) {
      return { #err("Unable to detect salt length: Reached end of DB without
          ↳encountering magic byte") };
    };
    paddingCount += 1;
  };

  // The next byte must be the magic byte 0x01 now.
  if (db[paddingCount] != (0x01 : Word8)) {
    return { #err("Unable to detect salt length: Magic byte 0x01 is missing or
        ↳padding corrupted") };
  };

  // First `paddingCount` bytes are `0x00` padding, then the magic `0x01` byte.
  return { #ok(db.size() - paddingCount - 1) };
};
```

5.12 RSASSA-PKCS1-v1_5: Deterministic signature scheme

The 'RSASSA-PKCS1-v1_5' signature scheme combines the 'EMSA-PKCS1-v1_5' encoding scheme with the signature creation and verification primitives to form a full signature scheme. Its operation is described in algorithm 14 for the signing, and algorithm 15 for the verification direction.

5.12.1 Implementation in Motoko

The Motoko implementation of the signing direction supports operation on either a byte array or an iterator. As Motoko has no support for method overloading, two separate methods exist. The one handling plain byte arrays wraps them in an iterator, and delegates processing to the other.

Algorithm 14 RSASSA-PKCS1-v1_5-SIGN

Input

Private key: $K = (p, q, dP, dQ, qInv)$
Message: Byte string *M*

Algorithm

k := Byte length of RSA modulus *N*
EM := *EMSA-PKCS1-V1_5-ENCODE*(*M*, *k*)

m := *OS2IP*(*EM*)
s := *RSASP1*(*K*, *m*)
S := *I2OSP*(*s*, *k*)

Return *S*

Algorithm 15 RSASSA-PKCS1-v1_5-VERIFY

Input

Public key: (n, e)
Message to be verified: Byte string *M*
Signature to be verified: Byte string *S*

Algorithm

k:= Byte length of RSA modulus *N*
If $|S| \neq k$:
    Return error

s := *OS2IP*(*S*)
m := *RSAVP1*((*n*, *e*), *s*)
EM := *I2OSP*(*m*, *k*)

EM' := *EMSA-PKCS1-V1_5-ENCODE*(*M*, *k*)

If *EM* == *EM'*:
    Return success
Else:
    Return error

The implementation is directly based on algorithm 14, with error handling added for when either of the steps fail.

```
public func pkcs15Sign(k : PrivateKey, m : [Word8]) : Result.Result<[Word8], Text> {
  return pkcs15SignIter(k, Iter.fromArray<Word8>(m));
};

public func pkcs15SignIter(k : PrivateKey, m : Iter.Iter<Word8>) : Result.Result<[
    ↳Word8], Text> {
  let mHash = hashFromIter(m);

  var encodedSigBytes : [Word8] = [];
  switch (pkcs15Encode(mHash, ceilDiv(k.modulusBits, 8))) {
    case (#ok(ary)) {
      encodedSigBytes := ary;
    };
    case (#err(e)) {
      return { #err(e) };
    };
  };

  let sigRep = primOS2IP(encodedSigBytes);

  var sig : Nat = 0;
  switch (primRSASP1(k, sigRep)) {
    case (#ok(n)) {
      sig := n;
    };
    case (#err(e)) {
      // The encoded signature has as many bytes as the modulus, but its
      // first byte is guaranteed to be 0x00 (and second 0x01), so it *has*
      // to be smaller than the RSA modulus. Hence the signing primitive may
      // not fail.
      Debug.print("ERROR: Signature representative was too large, RSA signing
          ↳primitive failed.");
      Prelude.unreachable();
    };
  };

  switch (primI2OSP(sig, ceilDiv(k.modulusBits, 8))) {
    case (#ok(sigBytes)) {
      return { #ok(sigBytes) };
    };
    case (#err(e)) {
      // As the signature representative is guaranteed to be smaller than the
      // RSA modulus, it *has* to fit into a modBit bits number.
      Debug.print("ERROR: Encrypted signature representative was too large,
          ↳conversion to octets failed.");
      Prelude.unreachable();
    };
  };
};
```

The implementation of the verification direction is equivalent, supporting operation on both iterators as well as plain byte arrays. It closely follows Algorithm 15, with error handling where steps may fail.

```
public func pkcs15Verify(pubkey : PublicKey, m : [Word8], sigBytes : [Word8]) :
    ↳Result.Result<Bool, Text> {
  return pkcs15VerifyIter(pubkey, Iter.fromArray<Word8>(m), sigBytes);
};

public func pkcs15VerifyIter(pubkey : PublicKey, m : Iter.Iter<Word8>, sigBytes : [
    ↳Word8]) : Result.Result<Bool, Text> {
  let k = ceilDiv(pubkey.modulusBits, 8);
  let mHash = hashFromIter(m);
```

```
  if (sigBytes.size() != k) {
    return { #err("Invalid signature: Signature length not equal to modulus byte
        ↳length.")};
  };

  let sigrep = primOS2IP(sigBytes);
  let encodedSigRep = switch(primRSAVP1(pubkey, sigrep)) {
    case (#ok(x)) x;
    case (#err(e)) return { #err("Invalid signature: " # e) };
  };

  let encodedSig = switch(primI2OSP(encodedSigRep, k)) {
    case (#ok(x)) x;
    case (#err(e)) return { #err("Invalid signature: " # e) };
  };

  let encodedSigFromMessage = switch (pkcs15Encode(mHash, k)) {
    case (#ok(x)) x;
    case (#err(e)) return { #err("Invalid signature: " # e) };
  };

  if (Array.equal<Word8>(encodedSig, encodedSigFromMessage, Word8.equal)) {
    return { #ok(true) };
  } else {
    return { #err("Invalid signature: Encoded signature mismatch.") };
  };
};
```

5.13 RSASSA-PSS: Probabilistic signature scheme

The "RSASSA-PSS" signature scheme combines the "EMSA-PSS-ENCODE" encoding scheme with the signature creation primitive, and the "EMSA-PSS-VERIFY" decoding scheme with the signature verification primitive, to form a full probabilistic signature scheme. It is described in algorithm 16 for the signing, and algorithm 17 for the verification direction.

A formal security proof for RSASSA-PSS is given by Bellare and Rogaway [MP00], but is not elaborated on in this thesis.

For the signing direction, note how the second parameter passed to *EMSA-PSS-ENCODE* ensures that the bit length of the encoded message will be one less than the bit length of the RSA modulus. This ensures the encoded message — interpreted as an integer — is less than the RSA modulus.

Algorithm 16 RSASSA-PSS-SIGN

Input

Private key: $K = (p, q, dP, dQ, qInv)$
Message: Byte string *M*

Algorithm

k := Byte length of RSA modulus *N*
modBits := Bit length of RSA modulus *N*
EM := *EMSA-PSS-ENCODE*(*M*, *modBits* - 1)

m := *OS2IP*(*EM*)
s := *RSASP1*(*K*, *m*)
S := *I2OSP*(*s*, *k*)

Return *S*

Algorithm 17 RSASSA-PSS-VERIFY

Input

Public key: (n, e)
Message to be verified: Byte string *M*
Signature to be verified: Byte string *S*

Algorithm

k := Byte length of RSA modulus *N*
modBits := Bit length of RSA modulus *N*

If $|S| \neq k$:
Return error

s := *OS2IP*(*S*)
m := *RSAVP1*((n, e), *s*)
EM := *I2OSP*(*m*, *ceil*((*modBits* - 1) / 8))

Result := *EMSA-PSS-VERIFY*(*M*, *EM*, *modBits* - 1)
Return *Result*

5.13.1 Implementation in Motoko

Implementation of "RSASSA-PSS-SIGN" closely follows algorithm 16. Two methods exist for signing messages as byte arrays and iterators respectively, with error handling added in case any of the steps fails.

```
public func pssSign(k : PrivateKey, m : [Word8]) : Result.Result<[Word8], Text> {
  return pssSignIter(k, Iter.fromArray<Word8>(m));
};

public func pssSignIter(k : PrivateKey, m : Iter.Iter<Word8>) : Result.Result<[Word8
    ↳], Text> {
  let mHash = hashFromIter(m);

  // We use a salt with the same length as the output of our hash function,
  // see section 9.1, fourth note, of the standard.
  var encodedSigBytes : [Word8] = [];
  switch (emsaPSSEncode(mHash, k.modulusBits - 1, hashLength)) {
    case (#ok(ary)) {
      encodedSigBytes := ary;
    };
    case (#err(e)) {
      return { #err(e) };
    };
  };

  let sigRep = primOS2IP(encodedSigBytes);

  var sig : Nat = 0;
  switch (primRSASP1(k, sigRep)) {
    case (#ok(n)) {
      sig := n;
    };
    case (#err(e)) {
      // As the encoded signature has modBits - 1 bits the signature
      // representative *has* to be smaller than the RSA modulus, so the
      // signing primitive may not fail.
      Debug.print("ERROR: Signature representative was too large, RSA signing
          ↳primitive failed.");
```

```
      Prelude.unreachable();
    };
  };

  switch (primI2OSP(sig, ceilDiv(k.modulusBits, 8))) {
    case (#ok(sigBytes)) {
      return { #ok(sigBytes) };
    };
    case (#err(e)) {
      // As the signature representative is guaranteed to be smaller than the
      // RSA modulus, it *has* to fit into a modBit bits number.
      Debug.print("ERROR: Encrypted signature representative was too large,
          ↳conversion to octets failed.");
      Prelude.unreachable();
    };
  };
};
```

The verification direction equivalently follows Algorithm 17.

```
public func pssVerify(pubkey : PublicKey, m : [Word8], sigBytes : [Word8],
    ↳saltLengthOrDefault : ?Nat) : Result.Result<Bool, Text> {
  return pssVerifyIter(pubkey, Iter.fromArray<Word8>(m), sigBytes,
      ↳saltLengthOrDefault);
};

// Same as `pssVerify`, but takes an iterator for the message.
public func pssVerifyIter(pubkey : PublicKey, m : Iter.Iter<Word8>, sigBytes : [
    ↳Word8], saltLengthOrDefault : ?Nat) : Result.Result<Bool, Text> {
  let emLen = ceilDiv(pubkey.modulusBits - 1, 8);
  let mHash = hashFromIter(m);

  if (sigBytes.size() != emLen) {
    return { #err("Invalid signature: Signature length != modulus size") };
  };

  let sigrep = primOS2IP(sigBytes);
  let encodedSigRep = switch(primRSAVP1(pubkey, sigrep)) {
    case (#ok(x)) x;
    case (#err(e)) return { #err("Invalid signature: " # e) };
  };

  let encodedSig = switch(primI2OSP(encodedSigRep, emLen)) {
    case (#ok(x)) x;
    case (#err(e)) return { #err("Invalid signature: " # e) };
  };

  switch (emsaPSSVerify(mHash, encodedSig, pubkey.modulusBits - 1,
      ↳saltLengthOrDefault)) {
    case (#ok(b)) {
      return { #ok(true) };
    };
    case (#err(e)) {
      return { #err("Invalid signature, verification failed: " # e) };
    };
  };
};
```

5.14 Deviations from PKCS#1

This section aims to discuss deviations of the Motoko implementation from the standard.

5.14.1 Calculation of message hash outside of encoding functions

As per the standard, calculation of the message hash is to take place within the encoding methods — that is "EMSA-PSS-ENCODE" and "EMSA-PKCS1-v1_5". In our implementation this instead takes place in the signing and verification methods, that is "RSASSA-PSS-SIGN", "RSASSA-PSS-VERIFY", "RSASSA-PKCS1-v1_5-SIGN" and "RSASSA-PKCS1-v1_5-VERIFY". The reason for doing this is that we support not only byte arrays but also iterators as messages. Hashing messages as early as possible therefore allows the inner methods to work on byte arrays rather than needing to support iterators. This simplifies the code as well as unit tests for encoding methods.

This is a software-engineering decision with no impact on security, as hashing still takes place within the library's code. For RSASSA-PSS specifically, the security proof even holds if the hash was controlled by the attacker, as per Section 9.1 of PKCS#1.

5.14.2 Fixed hash function

The standard allows free choice of hash functions for both the hashing of messages as well as for use within the "MGF1" mask-generation function. A list of recommended hash functions is given in Appendix B of PKCS#1.

As the Motoko standard library has no support for standard hash functions, a community-maintained implementation of SHA-256 is used for all operations [Hau20b].

5.15 Library API for developers

The following serves as an overview of the public API of the library. These types and methods offer high-level access to the signature and verification operations implemented in the library. Method signatures are left out for brevity, and can be found in the referenced sections.

Name	Description	Reference
PublicKey	Public key type	Section 5.3
PrivateKey	Private key type	Section 5.3
pkcs15Sign()	Signing direction of 'RSASSA-PKCS1-v1_5'	Section 5.12
pkcs15SignIter()	Signing direction of 'RSASSA-PKCS1-v1_5' for iterables	Section 5.12
pkcs15Verify()	Verifying direction of 'RSASSA-PKCS1-v1_5'	Section 5.12
pkcs15VerifyIter()	Verifying direction of 'RSASSA-PKCS1-v1_5' for iterables	Section 5.12
pssSign()	Signing direction of "RSASSA-PSS"	Section 5.13
pssSignIter()	Signing direction of "RSASSA-PSS" for iterables	Section 5.13
pssVerify()	Verifying direction of "RSASSA-PSS"	Section 5.13
pssVerifyIter()	Verifying direction of "RSASSA-PSS" for iterables	Section 5.13

The library also exposes other operations and primitives defined by the standard, but this is only done for additional flexibility, and the ability to have automated unit-tests for each of the operations. Direct usage of these other methods should not be required.

5.15.1 Example code

The following code snippet shows how to use the library to sign a message using the "RSASSA-PSS" signature scheme. The factors of the key are truncated for brevity. Note that the import statement has to be adjusted based on how the library is imported, see the Motoko documentation on imports for details.

```
import RSA "mo:crypto/RSA";

let privkey = {
  p = 11...;
```

```
  dP = 93...;
  q = 10...;
  dQ = 43...;
  qInv = 98...;
  d = 41...;
  modulusBits = 1024;
};
// 'hello' encoded as ASCII
let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];

switch (RSA.pssSign(privkey, message)) {
  case (#ok(sig)) {
        Debug.print("Signature: " # debug_show(sig));
  };
  case (#err(e)) {
    return "Error signing message: " # e;
  };
};
```

5.16 Compatibility with other implementations

In order to test compatibility with other implementations, OpenSSL was chosen as a wide-spread and stable candidate to test against. Ensuring that our implementation is compatible with other implementations of PKCS#1serves as evidence that the standard was implemented correctly.

Tests covered both directions — using OpenSSL to verify signatures generated by the Motoko library as well as using OpenSSL to create signatures which were then verified in the library. Additionally, compatibility with itself was tested too.

5.16.1 Test application

A small application was written which allowes creating and verifying signatures with a set of hardcoded keys. As signatures are arbitary byte sequences, Base64 encoding was used to exchange data with the application by means of the terminal. Its source code can be found in Appendix B.

5.16.2 Key generation

Two keypairs — one for the deterministic, one for the probabilistic signature scheme — were created as described in Section 5.3.3. Keys were stored on disk for use by OpenSSL, and hard-coded into the test application.

5.16.3 Signature creation in OpenSSL

Signature creation with OpenSSL was done as follows.

```
openssl dgst -sha256 -sign private.key -out signature_file input
```

5.16.4 Signature verification in OpenSSL

Signature verification with OpenSSL was done as follows.

```
openssl dgst -sha256 -prverify private.key -signature signature_file input
```

5.16.5 Results

Random 100-byte message were chosen for each test. Results are shown in table 5.1. All tests passed.

Signer	Verifier	Result
Motoko	Motoko	Pass
Motoko	OpenSSL	Pass
OpenSSL	Motoko	Pass
OpenSSL	OpenSSL	Pass

Table 5.1. Compatibility with other implementations

5.17 Performance

Performance of the signing and verification operations of both the "RSASSA-PKCS1-v1_5" as well as the "RSASSA-PSS" signature scheme was evaluated for RSA key sizes ranging from $512\,\mathrm{bit}$ to $4096\,\mathrm{bit}$. Evaluation took place on a local replica as well as the current public version of the IC.

5.17.1 Methodology

Framework

A set of RSA keys with sizes between $512\,\mathrm{bit}$ to $4096\,\mathrm{bit}$ was generated as described in Section 5.3.3. An application was written which utilized the library to sign respectively verify signatures of a hardcoded message ('Hello world') using the earlier generated keys.

This application exposes five endpoints, two each for the signing and verification operation of the two schemes, and one no-op endpoint to estimate network latency and overhead of the Internet Computer.

All endpoints were defined as query functions, meaning they execute on a single node and are not subject to delays due to consensus.

Measuring time

As the clock exposed by Motoko is constant during the processing of a request, time had to be measured client-sided. The time which elapsed between a command being issued to the application, and the result being returned to the client, was measured. These measurements therefore contain overhead such as network latency and processing by IC components outside the library code.

Compensating for overhead variance

As the signing and verification operations are fast, their actual execution time is likely to be lost in the variance of the overhead. To compensate for this, each endpoint was set up to execute the respective operation many times in sequence. This factor was chosen such that each call to an endpoint took at least $1\,\mathrm{s}$, so that the execution time of the algorithm dominates. This factor was compensated for during analysis of the data.

Each endpoint was further called four times with a fixed keypair, to compensate for intermediary differences in IC performance and potential network delays. The average per-operation execution time was used for analysis.

Compensating for overhead

As the application was hosted on the IC, it suffered overhead due to network latency as well as any preprocessing the IC has to do. The overhead was estimated by repeatedly calling a no-op endpoint, and subtracting the estimated value from the time it took other calls to complete.

Compensating for signature generation in verification profiling

The verification operation requires a signature to operate on, which is cumbersome to hardcode into the application due to its size. As such the performance measurement of the verification operation starts by creating a valid signature. The average time of one signature generation operation was then subtracted from each call to the signature verification endpoint.

Raw data and analysis code

The source code of the application used to profile the library can be found in appendix C.1. Raw measurements in Appendix C.2 and the analysis (in R) in Appendix C.3.

5.17.2 Results

Aggregated results are shown in table 5.3 and visualized in Figure 5.4. Both schemes show a sub-exponential growth of execution time with regard to the key size. This is expected as to modular exponentiation — which will dominate the execution time — is in $O(k^3)$ for k bit keys as discussed earlier.

The verification operation is around an order of magnitude faster than the signing operation, likely due to the public RSA exponent being chosen as a small prime.

Barely any difference is visible between the two schemes, as the modular exponentiation they have in common dominates any scheme-specific processing. A small difference can be seen for the verification operation with small key sizes, where the more complex verification operation of "RSASSA-PSS" is slightly slower.

Local execution times are approximately 1.5 times faster than corresponding times on the online version of the IC. This might hint at these nodes having a lower single-core performance than our local development environment, or the online IC being under sufficient load that applications are being throttled.

No-op results as shown in Table 5.5 further imply that the overhead of a query function on the IC is noticeable, but negligible for normal applications, being at roughly $50\,\mathrm{ms}$ on the local replica. On the hosted version network latency dominates, adding an additional half a second.

5.17.3 Caveats

Two caveats apply. In a real-world application it might be desireable to have signature generation happen in an update function, to benefit from the resiliency of the consensus protocol. This will add an overhead due to consensus of $2\,\mathrm{s}$ to $5\,\mathrm{s}$ per call which has to be accommodated for.

Secondly the library does not currently support randomness for the probabilistic signature scheme. Once supported by Motoko it is likely that the act of requesting randomness will have to go through consensus, incurring a delay of multiple seconds. A naive implementation where the library requests its own randomness will thus have this task severly dominate the total execution time.

A possible workaround is to allow users of this library to pass previously requested randomness into the signature operation. This allows retrieving this randomness during a time where it is less disruptive to operations, but puts the burden of securely sourcing randomness on the developer.

Key size	Cluster	Scheme	Operation	Time (s)
512	Local replica	RSASSA-PSS	Sign	0.015 s
512	Local replica	RSASSA-PSS	Verify	0.002 s
512	Local replica	RSASSA-PKCS1-v1_5	Sign	0.015 s
512	Local replica	RSASSA-PKCS1-v1_5	Verify	0.001 s
512	IC	RSASSA-PSS	Sign	0.025 s
512	IC	RSASSA-PSS	Verify	0.003 s
512	IC	RSASSA-PKCS1-v1_5	Sign	0.025 s
512	IC	RSASSA-PKCS1-v1_5	Verify	0.002 s
1024	Local replica	RSASSA-PSS	Sign	0.05 s
1024	Local replica	RSASSA-PSS	Verify	0.004 s
1024	Local replica	RSASSA-PKCS1-v1_5	Sign	0.05 s
1024	Local replica	RSASSA-PKCS1-v1_5	Verify	0.004 s
1024	IC	RSASSA-PSS	Sign	0.072 s
1024	IC	RSASSA-PSS	Verify	0.006 s
1024	IC	RSASSA-PKCS1-v1_5	Sign	0.071 s
1024	IC	RSASSA-PKCS1-v1_5	Verify	0.005 s
2048	Local replica	RSASSA-PSS	Sign	0.234 s
2048	Local replica	RSASSA-PSS	Verify	0.014 s
2048	Local replica	RSASSA-PKCS1-v1_5	Sign	0.235 s
2048	Local replica	RSASSA-PKCS1-v1_5	Verify	0.013 s
2048	IC	RSASSA-PSS	Sign	0.339 s
2048	IC	RSASSA-PSS	Verify	0.019 s
2048	IC	RSASSA-PKCS1-v1_5	Sign	0.343 s
2048	IC	RSASSA-PKCS1-v1_5	Verify	0.018 s
4096	Local replica	RSASSA-PSS	Sign	1.458 s
4096	Local replica	RSASSA-PSS	Verify	0.07 s
4096	Local replica	RSASSA-PKCS1-v1_5	Sign	1.354 s
4096	Local replica	RSASSA-PKCS1-v1_5	Verify	0.066 s
4096	IC	RSASSA-PSS	Sign	1.786 s
4096	IC	RSASSA-PSS	Verify	0.09 s
4096	IC	RSASSA-PKCS1-v1_5	Sign	1.741 s
4096	IC	RSASSA-PKCS1-v1_5	Verify	0.085 s

Table 5.3. Library performance

Cluster	Time (s)
Local replica	0.05 s
IC	0.62 s

Table 5.5. No-Op performance

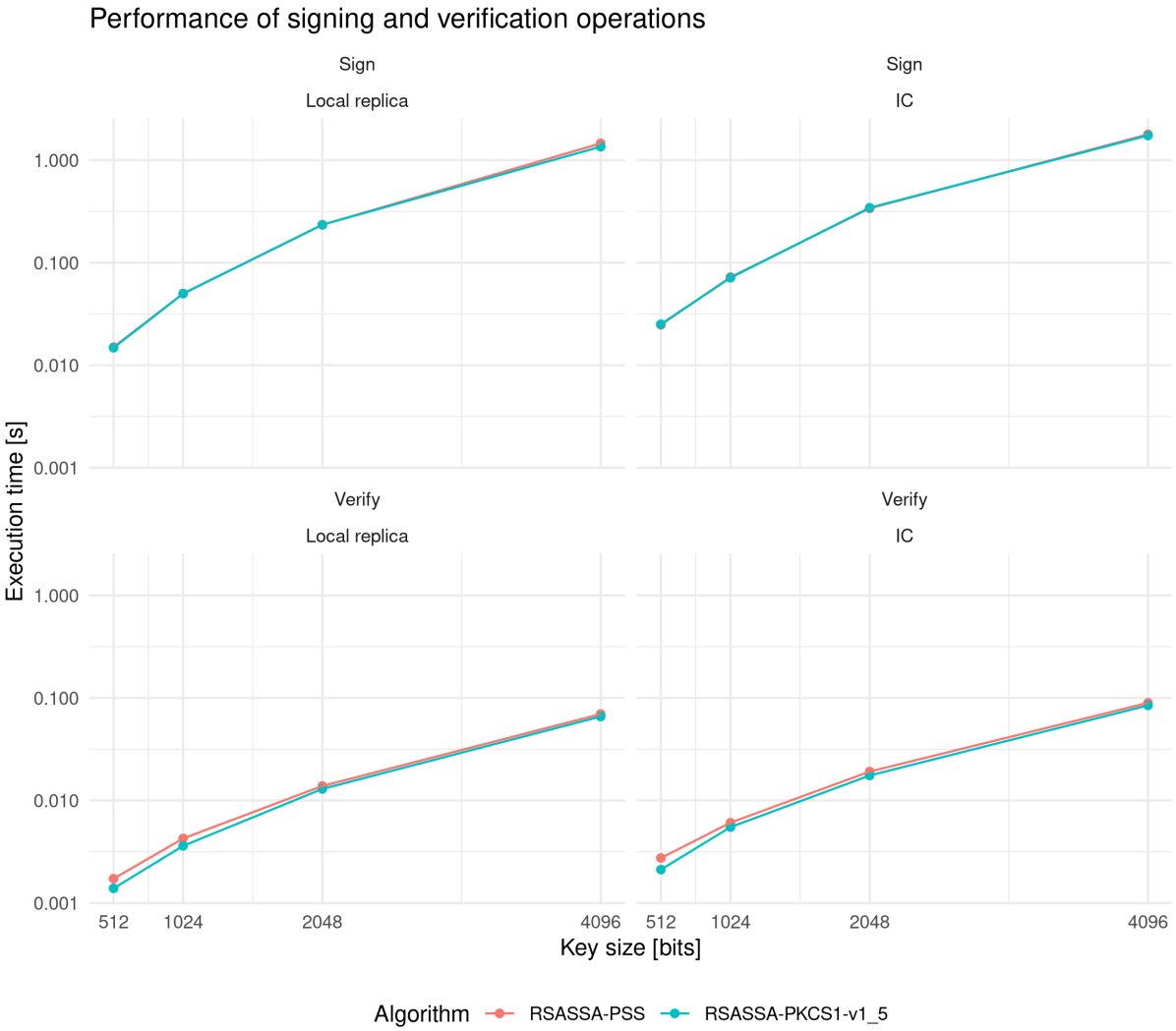

Figure 5.4. Library performance

Chapter 6

Experience with Motoko and the IC

As Motoko and the IC are relatively new and unknown, this chapter aims to provide an overview of experience gathered during this work. The goal is to stay as objective as possible, although opinions about programming languages and frameworks are often inherently subjective.

6.1 Motoko programming language

Motoko seems to be a modern language with good and interesting design fudamentals. It provides all the features one expects from a modern statically-typed language.

Its type system supporting both generics as well as variant types allows for concise and elegant code. As an example consider type-safe error handling as used throughout the library. Support for structural rather than nominal subtyping is different from most statically-typed mainstream programming languages but provides a lot of flexibility in that not the specific type, but rather its interface, is relevant.

Motoko is however clearly still in early and active development. This manifests itself e.g. in documentation of core language features being incomplete or incorrect. Examples we encountered are e.g. an implicit wrap-around of bitshift operations [Mor20b], garbage collection only happening once per message [Mor20a] or missing and inelegant type conversions [Mor21]. Documentation of its tooling is also limited, with usage and features of its compiler being minimally documented.

It should be noted that documentation has improved significantly in the last few months, and will likely continue to do so in the near future.

6.2 Motoko stdlib and third-party ecosystem

A shortcoming when it comes to the development of real-world applications is the bareness of Motoko's stdlib as well as the absence of a third-party ecosystem. As an example its standard library does not support common tasks such as conversion of string representations of numbers into numerical types. It lacks support for most non-trivial data structures, has no support for cryptography or even standard hash algorithms and up to recently had no support for cryptographically secure randomness.

The third-party ecosystem is still very limited, lacking both experienced developers willing and able to contribute open-source code, as well as stable and well-documented tooling for e.g. dependency management.

DFINITY is also struggling to release libraries they developed internally, with e.g. the release of "BigMap" — a library for a sharded key-value store — having been postponed for months [Hau20a].

These limitations are to be expected from a language so early in development, but quickly prove to be an obstacle when it comes to developing non-trivial real-world applications.

6.3 Internet Computer

From an academic point of view the IC is an interesting topic, promising fault-tolerance and scalability which is transparent to both users and application developers alike. DFINITY did promise to eventually release in-depth information, but is only doing so slowly, such as a recent paper on non-interactive key-sharing [Gro21].

From a practical point of view the IC is extremely easy to get started on, with beginner's tutorials guiding the developer through every step required to deploy their first application [DFI21]. However there is a steep learning curve when it comes to writing real-world applications, where the combined lack of documentation and missing features of the Motoko standard library become substantial roadblocks.

As an example consider the memory limitation of $4\,\mathrm{GB}$ of each canister. Storing real-world volumes of data then requires both data structures which are able to be sharded across multiple canisters, as well as the ability to spawn new canisters on demand. The later, while possible, is cumbersome and badly documented. For the former there exists no public implementations of such a data structure yet.

Further there are architectural limitations such as the inability for HTTP calls from within the IC to outside systems — let alone the ability to do arbitrary networking — which severly limit what can be done.

6.4 Summary

In summary, Motoko as a language seems promising even if still early in development. Its ecosystem is lacking however, requiring a lot of what is usually taken for granted to be implemented from scratch.

Lastly, what DFINITY staff communicates as vision seems to, at times, be in conflict with the current and immediate technical reality.

Chapter 7

Conclusion

In this thesis we have successfully implemented a Motoko library supporting the signature schemes defined in PKCS#1. These schemes allow applications written in Motoko to interface with other applications conformant to the standard. While doing so we have also created a host of supporting functionality for e.g. data encoding and big number arithmetic, which might find use independently. The performance of this library is sufficient for real-world applications, although still orders of magnitudes lower than existing libraries in established programming languages.

We have shown that certain real-world applications can be realised on the IC, albeit requiring more effort than would be required on mature plattforms. At the same time we have encountered various limitations which are hard to impossible to circumvent, limiting what can be realised until more features are added to the plattform.

Future work might consist of adding support for encryption schemes as specified by PKCS#1, utilizing the implemented signature schemes for higher-level applications such as certificate management, or analyzing and optimizing the performance of this library.

Appendix A

Unit tests

This chapter lists all unit tests for the library. Assuming a working local Internet Computer & Motoko setup, tests can be run by navigating to the "test" directory and executing "make". See the "Makefile" for commands which will be run.

A.1 Test framework

This section lists methods of the basic hand-made test framework which was utilized.

```
import Array "mo:base/Array";
import Debug "mo:base/Debug";
import Int "mo:base/Int";
import Nat "mo:base/Nat";
import Word8 "mo:base/Word8";
import Word16 "mo:base/Word16";

module {
  // Returning a boolean ensures that the stacktrace from the failed assertion
  // points to the line with the actual test, while this function can print
  // additional details.
  public func eq(x : Int, y : Int) : Bool {
    if (x == y) {
      return true;
    } else {
      return printAssertionError(Int.toText(x), Int.toText(y));
    };
  };

  public func natEq(x : Nat, y : Nat) : Bool {
    if (x == y) {
      return true;
    } else {
      return printAssertionError(Nat.toText(x), Nat.toText(y));
    };
  };

  public func byteEq(x : Word8, y : Word8) : Bool {
    if (x == y) {
      return true;
    } else {
      return printAssertionError(Word8.toText(x), Word8.toText(y));
    };
  };

  public func textEq(x : Text, y : Text) : Bool {
    if (x == y) {
      return true;
```

```
    } else {
      return printAssertionError(x, y);
    };
  };

  public func w16ArrayEq(x : [Word16], y : [Word16]) : Bool {
    if (Array.equal(x, y, Word16.equal)) {
      return true;
    } else {
      return printAssertionError(debug_show(x), debug_show(y));
    }
  };

  public func byteArrayEq(x : [Word8], y : [Word8]) : Bool {
    if (Array.equal(x, y, Word8.equal)) {
      return true;
    } else {
      return printAssertionError(debug_show(x), debug_show(y));
    }
  };

  func printAssertionError(x : Text, y : Text) : Bool {
    Debug.print("Assertion error: Expected " # x # " == " # y);
    return false;
  }
}
```

A.2 Base64 tests

This section lists tests for the Base64 module.

```
import Result "mo:base/Result";

import Base64 "mo:crypto/Base64";

import T "../../rsa/test/Test";

debug {
  func encode() {
    var input : [Word8] = [0x00, 0x00, 0x00];
    var out = Base64.encode(input, #standard);
    assert T.textEq(out, "AAAA");

    input := [0xFA, 0x47, 0xD8];
    out := Base64.encode(input, #standard);
    assert T.textEq(out, "+kfY");

    input := [0x0, 0x10, 0x83, 0x10, 0x51, 0x87, 0x20, 0x92, 0x8B, 0x30, 0xD3, 0x8F,
        ↳ 0x41, 0x14, 0x93, 0x51, 0x55, 0x97, 0x61, 0x96, 0x9B, 0x71, 0xD7, 0x9F, 0
        ↳x82, 0x18, 0xA3, 0x92, 0x59, 0xA7, 0xA2, 0x9A, 0xAB, 0xB2, 0xDB, 0xAF, 0xC3
        ↳, 0x1C, 0xB3, 0xD3, 0x5D, 0xB7, 0xE3, 0x9E, 0xBB, 0xF3, 0xDF, 0xBF];
    out := Base64.encode(input, #standard);
    assert T.textEq(out, "
        ↳ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/");

    // One byte left over
    input := [0xFA, 0x47, 0xD8, 0x25];
    out := Base64.encode(input, #standard);
    assert T.textEq(out, "+kfYJQ==");

    // Two bytes left over
    input := [0xFA, 0x47, 0xD8, 0x25, 0xA7];
```

```
    out := Base64.encode(input, #standard);
    assert T.textEq(out, "+kfYJac=");

    // URL encoding
    input := [0xFB, 0xF0, 0x00];
    out := Base64.encode(input, #url);
    assert T.textEq(out, "-_AA");
  };

  func decode() {
    var input = "AAAA";
    //var out = Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    //assert T.byteArrayEq(out, [0x00, 0x00, 0x00]);

    input := "+kfY";
    var out = Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    assert T.byteArrayEq(out, [0xFA, 0x47, 0xD8]);

    input := "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    out := Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    assert T.byteArrayEq(out, [0x0, 0x10, 0x83, 0x10, 0x51, 0x87, 0x20, 0x92, 0x8B,
        ↳0x30, 0xD3, 0x8F, 0x41, 0x14, 0x93, 0x51, 0x55, 0x97, 0x61, 0x96, 0x9B, 0
        ↳x71, 0xD7, 0x9F, 0x82, 0x18, 0xA3, 0x92, 0x59, 0xA7, 0xA2, 0x9A, 0xAB, 0xB2
        ↳, 0xDB, 0xAF, 0xC3, 0x1C, 0xB3, 0xD3, 0x5D, 0xB7, 0xE3, 0x9E, 0xBB, 0xF3, 0
        ↳xDF, 0xBF]);

    // One byte left over
    input := "+kfYJQ==";
    out := Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    assert T.byteArrayEq(out, [0xFA, 0x47, 0xD8, 0x25]);

    // Two bytes left over
    input := "+kfYJac=";
    out := Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    assert T.byteArrayEq(out, [0xFA, 0x47, 0xD8, 0x25, 0xA7]);
    // URL encoding
    input := "-_AA";
    out := Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    assert T.byteArrayEq(out, [0xFB, 0xF0, 0x00]);

    // Mixed encoding
    input := "-/AA";
    out := Result.unwrapOk<[Word8], Text>(Base64.decode(input));
    assert T.byteArrayEq(out, [0xFB, 0xF0, 0x00]);

    // Error if unpadded input
    input := "+kfYJac";
    var err = Result.unwrapErr<[Word8], Text>(Base64.decode(input));
    assert T.textEq(err, "Base64 value did not contain multiple of 4 characters.
        ↳Only padded values are supported.");

    // Error if invalid Base64 char
    input := "ABC;";
    err := Result.unwrapErr<[Word8], Text>(Base64.decode(input));
    assert T.textEq(err, "Invalid Base64 character: ';'");
  };

  encode();
  decode();
};
```

A.3 RSA tests

This section lists tests for the RSA module.

```
import Array "mo:base/Array";
import Result "mo:base/Result";
import Iter "mo:base/Iter";
import Word8 "mo:base/Word8";

import RSA "mo:crypto/RSA";

import SHA256 "mo:sha/SHA256";

import T "../../rsa/test/Test";

import Debug "mo:base/Debug";

debug {
  func pkcs15Encode() {
    let input : [Word8] = SHA256.sha256([0x68, 0x65, 0x6c, 0x6c, 0x6f]);
    let encodedLength = 65;

    var expected : [Word8] = [
      0x00, 0x01, // magic bytes
      0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, // PS
      0x00, // magic byte
0x30, 0x31, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02,
    ↳0x01, 0x05, 0x00, 0x04, 0x20, // DER-encoded ASN.1 DigestInfo magic bytes
      0x2c, 0xf2, 0x4d, 0xba, 0x5f, 0xb0, 0xa3, 0x0e, 0x26, 0xe8, 0x3b, 0x2a, 0xc5,
        ↳0xb9, 0xe2, 0x9e, 0x1b, 0x16, 0x1e, 0x5c, 0x1f, 0xa7, 0x42, 0x5e, 0x73, 0
        ↳x04, 0x33, 0x62, 0x93, 0x8b, 0x98, 0x24 // SHA-256, hash of input
    ];
    var result = RSA.pkcs15Encode(input, encodedLength);
    assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), expected);

    // Error if encoded length too short
    result := RSA.pkcs15Encode(input, 61);
    assert T.textEq(Result.unwrapErr<[Word8], Text>(result), "Intended encoded
       ↳message length too short");
  };

  func pkcs15Sign() {
    let privkey = {
      p = 11020296296279217962509830717061206754434665075238241629751
            81130962265023244294017243250165619921549708942140499878156811 3
            86029269896824799823012860617 7723;
      dP = 937087556645937804011089683995890764012901132662277384596 1
            95951390284956823748676212396741225063523233897303901265085 71
                928713000523078951499972322041758 41;
      q = 109047354674244945420320277592673369368652730738518818961 55
            15446648481293965633098109140463376116936652779985364003342 89
                282495414136399778494110939625415 33;
      dQ = 433047965691426056961602689263609428443254340326943677967 1
            54508361282545077583151675976771897200350844946395152105696 05
                152299092853104973298865241233060 09;
      qInv = 983084269951644067382156171540879339546754836296918950 72
            35549352286075218685831726299506526920724712753245135872259 00
                558916440694744504497653275069209356 1;
      d = 411091110470969689154080766462380469360133538679015960993 16
            24089411039457575169535245813930381082823508623217573761813 90
                472542163140762520705450581581483767720672804251506334497075 9
                782544383621147544271543975200749511476695123943075361286536 1
                380327243145577148214098060431321093673800351415187716044012 8
                06921;
```

```
    modulusBits = 1024;
  };
  let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];

  let result = RSA.pkcs15Sign(privkey, message);
  let sig = Result.unwrapOk<[Word8], Text>(result);
  assert T.byteArrayEq(sig, [0xA, 0xD5, 0xA5, 0x97, 0xFA, 0xD1, 0xD0, 0xD, 0xCA, 0
      ↳x4B, 0xE5, 0x88, 0x27, 0x4A, 0x82, 0x1A, 0xD0, 0x84, 0xB, 0x1A, 0xF9, 0x16,
      ↳ 0x79, 0xA7, 0x4F, 0xD4, 0x91, 0x8, 0x4E, 0xE4, 0x5E, 0x6E, 0xEC, 0x4D, 0
      ↳x2B, 0x51, 0xD1, 0xAC, 0x2, 0x3D, 0x26, 0xD7, 0x86, 0xB2, 0x55, 0x4B, 0x95,
      ↳ 0xEE, 0xC6, 0x28, 0xAF, 0xAC, 0xD8, 0x17, 0x3D, 0x38, 0x5F, 0x75, 0x4A, 0
      ↳xC7, 0xDB, 0x93, 0x93, 0x48, 0x76, 0xC0, 0x26, 0xB8, 0x9, 0x5D, 0xDD, 0x84,
      ↳ 0x18, 0x7B, 0x9B, 0x7E, 0x52, 0xA6, 0xE, 0xEF, 0xC1, 0x7B, 0x3, 0x63, 0x89
      ↳, 0x45, 0xDE, 0xC8, 0x6F, 0xB3, 0x17, 0xF1, 0xE1, 0x9B, 0xB, 0x6A, 0x1A, 0
      ↳x0, 0xB1, 0x3, 0x64, 0x0, 0x9A, 0xFB, 0xC7, 0x4E, 0xCD, 0x18, 0x84, 0xD3, 0
      ↳x4, 0xC3, 0x5F, 0x51, 0x1D, 0x3B, 0x1E, 0x9F, 0xA8, 0xC6, 0xF6, 0x29, 0xE7,
      ↳ 0xD3, 0x31, 0x24, 0x6A, 0x76]);
};

func pkcs15Verify() {
  let pubkey = {
    n = 11919453465862970171090893627251153344045511531627172701842
          659381614664098109095666114861345818082584787981226439468016 2
              078154311655130646822909856191624639428448719998085881859149 7
              526772354333456430079060616813713160544073238916599071662043 5
              126141956413991538921217646806999789874635673801247662683071 5
              125899;
    e = 65537;
    modulusBits = 1024;
  };
  let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];
  var sig : [Word8] = [0x82, 0x3e, 0xc7, 0x53, 0xa7, 0x91, 0x9e, 0x39, 0x22, 0x6f,
      ↳ 0xe0, 0x67, 0xcb, 0xff, 0x6a, 0x06, 0x4b, 0x38, 0x40, 0x00, 0x12, 0x54, 0
      ↳xd6, 0x17, 0x0b, 0x10, 0x46, 0x26, 0xa0, 0x8c, 0x9b, 0x4c, 0x04, 0xeb, 0x9b
      ↳, 0xb5, 0xf9, 0x43, 0x54, 0x96, 0xc5, 0x2e, 0x08, 0x01, 0x6a, 0x2a, 0xc2, 0
      ↳xc1, 0xe2, 0x0a, 0xfc, 0x92, 0x9c, 0xef, 0x75, 0xa7, 0xa7, 0x54, 0x0e, 0xd9
      ↳, 0x1d, 0xb5, 0xc7, 0x34, 0x40, 0xbe, 0x99, 0x78, 0x53, 0x0f, 0x3b, 0xc2, 0
      ↳x83, 0x19, 0xad, 0x9e, 0x6f, 0x80, 0x77, 0x92, 0x58, 0xba, 0xaf, 0x51, 0x2a
      ↳, 0xc2, 0x80, 0x69, 0x46, 0xfb, 0x36, 0x3d, 0xd0, 0x6e, 0xee, 0x63, 0xc4, 0
      ↳x6c, 0x71, 0x60, 0x0c, 0x2b, 0x17, 0x1b, 0x35, 0x91, 0x40, 0x02, 0x34, 0xf1
      ↳, 0x3c, 0x67, 0x3e, 0x85, 0xf8, 0xc4, 0x60, 0xdc, 0x2b, 0xcc, 0xd7, 0xc4, 0
      ↳xc7, 0xa2, 0x63, 0x7f, 0x17, 0x88];

  var result = RSA.pkcs15Verify(pubkey, message, sig);
  assert Result.unwrapOk<Bool, Text>(result);

  // Signature / message mismatch
  result := RSA.pkcs15Verify(pubkey, [0x67, 0x64, 0x6b, 0x6e], sig);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Invalid signature:
      ↳Encoded signature mismatch.");

  // Signature too short
  result := RSA.pkcs15Verify(pubkey, message, [0x00, 0x01]);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Invalid signature:
      ↳Signature length not equal to modulus byte length.");
};

// Test whether:
// - Our own signatures verify
// - External signatures verify
func pkcs15Integration() {
  let privkey = {
    p = 11020296296279217962509830717061206754434665075238241629751
```

```
          8113096226502324429401724325016561992154970894214049987815681
              1386029269896824799823012860617723;
    dP = 9370875566459378040110896839958907640129011326622773845961
          9595139028495682374867621239674122506352323389730390126508571
              9287130005230789514999723220417584l;
    q = 1090473546742449454203202775926733693686527307385188l896155
          1544664848129396563309810914046337611693665277998536400334289
              2824954141363997784941109396254153 3;
    dQ = 4330479656914260569616026892636094284432543403269436779671
          5450836128254507758315167597677189720035084494639515210569605
              1522990928531049732988652412330600 9;
    qInv = 9830842699516440673821561715408793395467548362969189507 2
          3554935228607521868583172629950652692072471275324513587225900
              5589164406947445044976532750692093561;
    d = 4110911104709696891540807664623804693601335386790159609931 6
          2408941103945757516953524581393038108282350862321757376181390
              4725421631407625207054505815814837677206728042515063344970759
              7825443836211475442715439752007495114766951239430753612865361
              3803272431455771482140980604313210936738003514151877160440128
              06921;
    modulusBits = 1024;
  };
  let pubkey = {
    n = 1201734158835627838801507256864491227104468162023581294241 8
          8789842424413636247774580224322028362114726029100232004829830
              8907578203566446019870338318618390945201953908602804631659490
              6193388347870309913259704957199801387533051876684360852478744
              1183689210147157502382120887635763306858951076966521058697766
              869359;
    e = 65537;
    modulusBits = 1024;
  };
  let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];

  // Test own signature
  var sig = Result.unwrapOk<[Word8], Text>(RSA.pkcs15Sign(privkey, message));
  assert Result.unwrapOk<Bool, Text>(RSA.pkcs15Verify(pubkey, message, sig));

  // Test external (OpenSSL-generated) signature
  sig := [0xa, 0xd5, 0xa5, 0x97, 0xfa, 0xd1, 0xd0, 0xd, 0xca, 0x4b, 0xe5, 0x88, 0
      ↳x27, 0x4a, 0x82, 0x1a, 0xd0, 0x84, 0xb, 0x1a, 0xf9, 0x16, 0x79, 0xa7, 0x4f,
      ↳ 0xd4, 0x91, 0x8, 0x4e, 0xe4, 0x5e, 0x6e, 0xec, 0x4d, 0x2b, 0x51, 0xd1, 0
      ↳xac, 0x2, 0x3d, 0x26, 0xd7, 0x86, 0xb2, 0x55, 0x4b, 0x95, 0xee, 0xc6, 0x28,
      ↳ 0xaf, 0xac, 0xd8, 0x17, 0x3d, 0x38, 0x5f, 0x75, 0x4a, 0xc7, 0xdb, 0x93, 0
      ↳x93, 0x48, 0x76, 0xc0, 0x26, 0xb8, 0x9, 0x5d, 0xdd, 0x84, 0x18, 0x7b, 0x9b,
      ↳ 0x7e, 0x52, 0xa6, 0xe, 0xef, 0xc1, 0x7b, 0x3, 0x63, 0x89, 0x45, 0xde, 0xc8
      ↳, 0x6f, 0xb3, 0x17, 0xf1, 0xe1, 0x9b, 0xb, 0x6a, 0x1a, 0x0, 0xb1, 0x3, 0x64
      ↳, 0x0, 0x9a, 0xfb, 0xc7, 0x4e, 0xcd, 0x18, 0x84, 0xd3, 0x4, 0xc3, 0x5f, 0
      ↳x51, 0x1d, 0x3b, 0x1e, 0x9f, 0xa8, 0xc6, 0xf6, 0x29, 0xe7, 0xd3, 0x31, 0x24
      ↳, 0x6a, 0x76];
  assert Result.unwrapOk<Bool, Text>(RSA.pkcs15Verify(pubkey, message, sig));
};

func mask() {
  var input : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];
  var expected : [Word8] = [0xda, 0x75, 0x44, 0x7e, 0x22, 0xf9, 0xf9, 0x9e, 0x1b,
      ↳0xe0, 0x9a, 0x0, 0xcf, 0x1a, 0x7, 0x3, 0x6e, 0x39, 0x57, 0x57, 0x46, 0xba,
      ↳0x2e, 0x2a, 0xcb, 0x26, 0x5c, 0x2b, 0xde, 0x84, 0xc3, 0x72, 0x1f, 0x54, 0
      ↳x39, 0xc8, 0x35, 0xc0, 0xbd, 0xe3, 0x0, 0xa6, 0x23, 0xaf, 0x79, 0xb0, 0x89,
      ↳ 0xe, 0xff, 0x16, 0x1, 0x80, 0xfb, 0x7c, 0xbc, 0x8a, 0xa4, 0x5f, 0xbc, 0xc2
      ↳, 0xa4, 0x22, 0x6a, 0x36, 0x6a, 0xa4, 0x57, 0x76, 0xd5, 0xbf, 0x72, 0x55, 0
      ↳xc7, 0x19, 0x83, 0x12, 0x80, 0x27, 0x5d, 0xe3, 0x48, 0x8b, 0x11, 0x60, 0xb0
      ↳, 0x52, 0xdb, 0xf8, 0x76, 0x27, 0x7f, 0xf7, 0x40, 0x38, 0xae, 0x5c, 0x2a, 0
```

```
      ↳x17, 0xa1, 0x75, 0x73, 0x4, 0xa2, 0x4e, 0x8b, 0x96, 0xf2, 0xf9, 0xd7, 0x9e,
      ↳ 0x91, 0x3c, 0x98, 0xf, 0xe2, 0xb0, 0x64, 0xcf, 0x57, 0x4b, 0xff, 0x19, 0
      ↳xe1, 0xc2, 0xc, 0x1c, 0x3b, 0x72];
  var result = RSA.mask(input, 128);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), expected);

  // Same input, but we only care about a 120 byte output now.
  expected := [0xda, 0x75, 0x44, 0x7e, 0x22, 0xf9, 0xf9, 0x9e, 0x1b, 0xe0, 0x9a, 0
      ↳x0, 0xcf, 0x1a, 0x7, 0x3, 0x6e, 0x39, 0x57, 0x57, 0x46, 0xba, 0x2e, 0x2a, 0
      ↳xcb, 0x26, 0x5c, 0x2b, 0xde, 0x84, 0xc3, 0x72, 0x1f, 0x54, 0x39, 0xc8, 0x35
      ↳, 0xc0, 0xbd, 0xe3, 0x0, 0xa6, 0x23, 0xaf, 0x79, 0xb0, 0x89, 0xe, 0xff, 0
      ↳x16, 0x1, 0x80, 0xfb, 0x7c, 0xbc, 0x8a, 0xa4, 0x5f, 0xbc, 0xc2, 0xa4, 0x22,
      ↳ 0x6a, 0x36, 0x6a, 0xa4, 0x57, 0x76, 0xd5, 0xbf, 0x72, 0x55, 0xc7, 0x19, 0
      ↳x83, 0x12, 0x80, 0x27, 0x5d, 0xe3, 0x48, 0x8b, 0x11, 0x60, 0xb0, 0x52, 0xdb
      ↳, 0xf8, 0x76, 0x27, 0x7f, 0xf7, 0x40, 0x38, 0xae, 0x5c, 0x2a, 0x17, 0xa1, 0
      ↳x75, 0x73, 0x4, 0xa2, 0x4e, 0x8b, 0x96, 0xf2, 0xf9, 0xd7, 0x9e, 0x91, 0x3c,
      ↳ 0x98, 0xf, 0xe2, 0xb0, 0x64, 0xcf, 0x57, 0x4b];
  result := RSA.mask(input, 120);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), expected);

  // Same input, now we want an output of size less than the hash size
  expected := [0xda, 0x75, 0x44, 0x7e, 0x22, 0xf9, 0xf9, 0x9e, 0x1b];
  result := RSA.mask(input, 9);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), expected);

  // Size over 2**32 * hashLength should cause an error, as the counter would
  // not fit into 4 bytes.
  result := RSA.mask(input, 2**32 * 32 + 1);
  assert T.textEq(Result.unwrapErr(result), "Mask too long");

  // And a second test for success
  input := [0x00];
  expected := [0x88, 0x55, 0x50, 0x8a, 0xad, 0xe1, 0x6e, 0xc5, 0x73, 0xd2, 0x1e, 0
      ↳x6a, 0x48, 0x5d, 0xfd, 0x0a, 0x76, 0x24, 0x08, 0x5c, 0x1a, 0x14, 0xb5, 0xec
      ↳, 0xdd, 0x64, 0x85, 0xde, 0x0c, 0x68, 0x39, 0xa4, 0x15, 0xf2, 0xf1, 0xa4, 0
      ↳x33, 0x9f, 0x5f, 0x2a, 0x31, 0x3b, 0x95, 0x01, 0x5c, 0xad, 0x81, 0x24, 0xd0
      ↳, 0x54, 0xa1, 0x71, 0xac, 0x2f, 0x31, 0xcf, 0x52, 0x9d, 0xda, 0x7c, 0xfb, 0
      ↳x6a, 0x38, 0xb4, 0x89, 0xee, 0xfc, 0x18, 0xfa, 0x4b, 0x81, 0x5b, 0xd1, 0xad
      ↳, 0xed, 0x2f, 0x24, 0xeb, 0x28, 0x88, 0x59, 0x93, 0xaa, 0x00, 0xb6, 0xd0, 0
      ↳x17, 0x1b, 0xf5, 0x00, 0x5f, 0x9d, 0x39, 0xaa, 0xea, 0x10, 0x01, 0x6a, 0x68
      ↳, 0x2d, 0x1d, 0xf4, 0xf8, 0x69, 0xb3, 0x2c, 0x48, 0xb0, 0xa9, 0xb4, 0x42, 0
      ↳xa1, 0x49, 0x39, 0x49, 0xfb, 0x85, 0xd9, 0x51, 0xd1, 0x21, 0xc1, 0x14, 0x3b
      ↳, 0xd3, 0xd5, 0xc1, 0xaf];
  result := RSA.mask(input, 128);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), expected);
};

func detectPSSSaltLength() {
  // No padding
  var input : [Word8] = [0x01, 0xAA, 0xAA];
  var result = RSA.detectPSSSaltLength(input);
  assert T.natEq(Result.unwrapOk<Nat, Text>(result), 2);

  // No padding, no salt
  input := [0x01];
  result := RSA.detectPSSSaltLength(input);
  assert T.natEq(Result.unwrapOk<Nat, Text>(result), 0);

  // No salt
  input := [0x00, 0x00, 0x00, 0x01];
  result := RSA.detectPSSSaltLength(input);
  assert T.natEq(Result.unwrapOk<Nat, Text>(result), 0);
```

```
  // Padding and salt
  input := [0x00, 0x00, 0x00, 0x01, 0xAA, 0xAA, 0xAA, 0xAA];
  result := RSA.detectPSSSaltLength(input);
  assert T.natEq(Result.unwrapOk<Nat, Text>(result), 4);

  // Short padding and salt
  input := [0x00, 0x01, 0xAA];
  result := RSA.detectPSSSaltLength(input);
  assert T.natEq(Result.unwrapOk<Nat, Text>(result), 1);

  // DB too short
  input := [];
  result := RSA.detectPSSSaltLength(input);
  assert T.textEq(Result.unwrapErr<Nat, Text>(result), "Unable to detect salt
      ↳length: DB too short");

  // Corrupt padding
  input := [0x00, 0xAA, 0x01, 0xAA];
  result := RSA.detectPSSSaltLength(input);
  assert T.textEq(Result.unwrapErr<Nat, Text>(result), "Unable to detect salt
      ↳length: Magic byte 0x01 is missing or padding corrupted");

  // Missing magic byte
  input := [0x00, 0x00, 0xAA, 0xAA, 0xAA];
  result := RSA.detectPSSSaltLength(input);
  assert T.textEq(Result.unwrapErr<Nat, Text>(result), "Unable to detect salt
      ↳length: Magic byte 0x01 is missing or padding corrupted");

  // Missing magic byte no salt
  input := [0x00, 0x00, 0x00];
  result := RSA.detectPSSSaltLength(input);
  assert T.textEq(Result.unwrapErr<Nat, Text>(result), "Unable to detect salt
      ↳length: Reached end of DB without encountering magic byte");
};

func emsaPSSEncode() {
  let saltLength = 4;
  let encodedBitlength = 256 + 8 * saltLength + 2 * 8 + 3 * 8;
  let input : [Word8] = SHA256.sha256([0x68, 0x65, 0x6c, 0x6c, 0x6f]);

  var expected : [Word8] = [
    0xAB, 0x63, 0xCF, 0xB1, 0x59, 0xE3, 0x2, 0x55, // maskedDB
    0x61, 0xBA, 0xD8, 0x13, 0x22, 0x69, 0x41, 0xD6, 0x4F, 0x79, 0x68, 0x8F, 0xD4,
        ↳0x42, 0x26, 0x18, 0x84, 0xF5, 0xA3, 0x37, 0x8B, 0xE7, 0x4D, 0x20, 0x3, 0
        ↳x79, 0xC, 0xC6, 0xEC, 0xC4, 0x8D, 0xFB, // h
    0xbc, // magic byte
  ];
  var result = RSA.emsaPSSEncode(input, encodedBitlength, saltLength);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), expected);

  // TODO Test with encodedBitLength not a multiple of 8

  // Error if desired length of encoded message is too short.
  result := RSA.emsaPSSEncode(input, 8 * 32 + 8 * saltLength + 8, saltLength);
  assert T.textEq(Result.unwrapErr<[Word8], Text>(result), "Encoding error:
      ↳Desired length of encoded message too short. 37 < 38");

  // TODO Test with another message
};

func emsaPSSVerify() {
  let saltLength = 4;
  let encodedBitlength = 256 + 8 * saltLength + 2 * 8 + 3 * 8;
```

```
  // `hello`
  let input : [Word8] = SHA256.sha256([0x68, 0x65, 0x6c, 0x6c, 0x6f]);
  // Valid encoding of input `hello`
  var encoding : [var Word8] = [var
    0xAB, 0x63, 0xCF, 0xB1, 0x59, 0xE3, 0x2, 0x55, // maskedDB
    0x61, 0xBA, 0xD8, 0x13, 0x22, 0x69, 0x41, 0xD6, 0x4F, 0x79, 0x68, 0x8F, 0xD4,
        ↳0x42, 0x26, 0x18, 0x84, 0xF5, 0xA3, 0x37, 0x8B, 0xE7, 0x4D, 0x20, 0x3, 0
        ↳x79, 0xC, 0xC6, 0xEC, 0xC4, 0x8D, 0xFB, // h
    0xbc, // magic byte
  ];

  var result = RSA.emsaPSSVerify(input, Array.freeze<Word8>(encoding),
      ↳encodedBitlength, ?saltLength);
  assert Result.unwrapOk<Bool, Text>(result);

  // Error if hash does not match message
  result := RSA.emsaPSSVerify([0x00, 0x01, 0x02, 0x03], Array.freeze<Word8>(
      ↳encoding), encodedBitlength, ?saltLength);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Hash mismatch");

  // Error if magic byte of DB != 0x01
  encoding[3] := 0xAA;
  result := RSA.emsaPSSVerify(input, Array.freeze<Word8>(encoding),
      ↳encodedBitlength, ?saltLength);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Byte 4 of DB != 0x01");
  encoding[3] := 0xB1;

  // Error if PS padding of DB != 0x00
  encoding[1] := 0xAA;
  result := RSA.emsaPSSVerify(input, Array.freeze<Word8>(encoding),
      ↳encodedBitlength, ?saltLength);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Byte 2 of PS padding of
      ↳DB != 0x00");
  encoding[1] := 0x63;

  // Error if rightmost byte of signature != 0xBC
  encoding[encoding.size() - 1] := 0xAA;
  result := RSA.emsaPSSVerify(input, Array.freeze<Word8>(encoding),
      ↳encodedBitlength, ?saltLength);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Rightmost byte != 0xBC");
  encoding[encoding.size() - 1] := 0xBC;

  // Error if encoded message too short
  result := RSA.emsaPSSVerify(input, [0x00, 0x01, 0x02] : [Word8], 3 * 8, ?
      ↳saltLength);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Encoded message too short
      ↳");

  // TODO: Error checking of leftmost x bits of maskedDB != 0. Requires a
  // modulus which is not a power of two though.
};

func pssSign() {
  let privkey = {
    p = 123962340477758208615763079097722555840558580158227198148980
          590820030107954455870834150213400900385065330064467800488622 0
              298208769407030621592782805913495 67;
    dP = 946102554031613872583708966444754035215054033741911626137 0
          105653708378012022646798806163298589138358221226095410419158 4
              190550443825171501006292676945651 15;
    q = 113260015705603436531325559584502180152445924227427279533 88
          118829320340011247336476724320332946680024284558963094235176 9
```

```
            29635221057635961534148319083251451;
    dQ = 65054042315120163592301875878349872094824632279369594023 13
          02862645689547955175682428755649927692564740747445027627591 68
            89120018100807032669025820151225023;
    qInv = 90831800773216386159733573757443140161967060856210494581
          09458363227680628762240416075315574538631116793296136394141 65
            57153394425324904269258098568318157 65;
    d = 27406414852693374468560631691199018678861189081106247615557
          28218931407267020507946685571231928457550464143509759215928 91
            41769470523134963407871005690938400553383280198857223890065 70
            19691153860750474854636287067449074262807108341455521257615 35
            95194577898099833323462286999803215294797095197862046747715 54
            65773;
    modulusBits = 1024;
  };
  let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];

  let result = RSA.pssSign(privkey, message);
  let sig = Result.unwrapOk<[Word8], Text>(result);
  assert T.byteArrayEq(sig, [0x74, 0xD7, 0xBE, 0x57, 0x16, 0x8D, 0xEE, 0x34, 0xE9,
      ↳ 0x6F, 0xFA, 0x61, 0x68, 0x78, 0x2F, 0x93, 0xB8, 0xF9, 0xA4, 0x9A, 0xF5, 0
      ↳xC8, 0xD, 0x9C, 0x4D, 0xCC, 0x74, 0x2A, 0xBD, 0x2, 0x3D, 0x71, 0x7E, 0x3, 0
      ↳x40, 0x35, 0x1B, 0xF3, 0xF1, 0x64, 0x17, 0xAF, 0x85, 0xFB, 0xFC, 0x9B, 0x6E
      ↳, 0x42, 0x7F, 0xBC, 0x23, 0x8E, 0xEF, 0x9B, 0x41, 0x39, 0x9, 0x93, 0x75, 0
      ↳x4B, 0x7D, 0x2B, 0xE8, 0x75, 0x8D, 0x88, 0xD0, 0xA6, 0x39, 0xBF, 0x30, 0x4B
      ↳, 0xFC, 0x5D, 0xE6, 0x0, 0x66, 0x77, 0x43, 0xAA, 0xE4, 0x3A, 0xD0, 0xF6, 0
      ↳x2F, 0x40, 0x9F, 0x81, 0x47, 0x88, 0x1D, 0x5C, 0xC8, 0x48, 0x54, 0x9, 0x8B,
      ↳ 0x42, 0xB3, 0x57, 0xD0, 0xE1, 0xC7, 0x83, 0xE, 0x41, 0x90, 0xE8, 0x3F, 0
      ↳xDF, 0x21, 0xBD, 0xCE, 0xA5, 0xBA, 0x13, 0x53, 0x3E, 0x19, 0x15, 0x39, 0x7C
      ↳, 0xE2, 0xD6, 0x99, 0xB2, 0xCE, 0xCA]);
};

func pssVerify() {
  let pubkey = {
    n = 11919453465862970171090893627251153344045511531627172701842 65
          93816146640981090956661148613458180825847879812264394680162 07
            81543116551306468229098561916246394284487199980858818591497 52
            67723543334564300790606168137131605440732389165990716620435 12
            61419564139915389212176468069997898746356738012476626830715 12
            5899;
    e = 65537;
    modulusBits = 1024;
  };

  let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];
  var sig : [var Word8] = [var 0x23, 0xAD, 0x17, 0x6C, 0x12, 0x63, 0xA6, 0xFF, 0
      ↳x9D, 0x58, 0x5D, 0xEF, 0x1E, 0xA1, 0x3E, 0x74, 0xFD, 0x94, 0x9F, 0x38, 0xB4
      ↳, 0xEA, 0xB9, 0xC0, 0xD1, 0x8D, 0xEA, 0x9B, 0x76, 0x9A, 0xE5, 0x5E, 0x5D, 0
      ↳xDC, 0xC8, 0xF4, 0x91, 0x7D, 0xB0, 0xCE, 0x61, 0x3B, 0x93, 0x2, 0xB5, 0x16,
      ↳ 0x89, 0xF3, 0xD5, 0x48, 0x1, 0xF7, 0x4E, 0xB1, 0xD8, 0xFE, 0xD9, 0xFC, 0
      ↳x5B, 0x0, 0xAD, 0x61, 0xCE, 0x1F, 0x5B, 0xFB, 0xEC, 0x59, 0xBC, 0x2A, 0xFE,
      ↳ 0xE7, 0x74, 0xDF, 0x32, 0x8C, 0xF7, 0x9A, 0x7C, 0x3E, 0xD8, 0x1B, 0xD1, 0
      ↳x92, 0x7A, 0xC6, 0xB9, 0x2F, 0x65, 0xB8, 0xA1, 0x64, 0x6C, 0xC1, 0x1B, 0xAD
      ↳, 0xF, 0x31, 0x76, 0xF1, 0x72, 0xAA, 0x5A, 0x1C, 0x79, 0x51, 0x45, 0x45, 0
      ↳x3F, 0x9F, 0xD5, 0x5F, 0x95, 0xB1, 0x77, 0x26, 0xC1, 0x11, 0x64, 0xB6, 0x39
      ↳, 0x2E, 0xD0, 0xFF, 0x56, 0xB1, 0x2D, 0x5A];

  var result = RSA.pssVerify(pubkey, message, Array.freeze<Word8>(sig), null);
  assert Result.unwrapOk<Bool, Text>(result);

  // Error if signature too short
  result := RSA.pssVerify(pubkey, message, [0x00, 0x01] : [Word8], null);
```

```
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Invalid signature:
      ↳Signature length != modulus size");

  // Error if message mismatch
  result := RSA.pssVerify(pubkey, [0x00, 0x01, 0x02, 0x03, 0x04], Array.freeze<
      ↳Word8>(sig), null);
  assert T.textEq(Result.unwrapErr<Bool, Text>(result), "Invalid signature,
      ↳verification failed: Hash mismatch");

  // TODO: Error if RSAVP1, OS2IP or I2OSP fail
};

func primI2OSP() {
  var result = RSA.primI2OSP(0, 0);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), ([] : [Word8]));

  result := RSA.primI2OSP(0, 1);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), ([0x00] : [Word8]))
      ↳;

  result := RSA.primI2OSP(3825184, 3);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), ([0x3A, 0x5E, 0x20]
      ↳ : [Word8]));

  // Test padding
  result := RSA.primI2OSP(6857394, 5);
  assert T.byteArrayEq(Result.unwrapOk<[Word8], Text>(result), ([0x00, 0x00, 0x68,
      ↳ 0xA2, 0xB2] : [Word8]));

  // Test error if input too big
  result := RSA.primI2OSP(256, 1);
  assert T.textEq(Result.unwrapErr(result), "Integer too large for byte array of
      ↳length 1");
};

func primOS2IP() {
  assert T.natEq(RSA.primOS2IP([]), 0);
  assert T.natEq(RSA.primOS2IP([0x00, 0x00, 0x00]), 0);
  assert T.natEq(RSA.primOS2IP([0x0F, 0x02]), 3842);
  assert T.natEq(RSA.primOS2IP([0x3A, 0x5E, 0x20]), 3825184);
  assert T.natEq(RSA.primOS2IP([0x09, 0x4F, 0xDA, 0xD1, 0x1D, 0xF4, 0xD9, 0xD1, 0
      ↳x7E, 0x84, 0x3C, 0xA3, 0x1C, 0x55, 0x10, 0xFD, 0x33, 0x58, 0xDE, 0x41, 0x01
      ↳, 0x08, 0x87, 0x28, 0x9B, 0xE4, 0x09, 0xAA, 0x52, 0x58, 0xBF, 0xFD, 0x04, 0
      ↳x3A, 0x7C, 0x8C, 0xB1, 0xA2, 0x4B, 0x61, 0x9F, 0x6C, 0x58, 0xF8, 0x42, 0xC9
      ↳, 0x4D, 0x63, 0x9B, 0x56, 0x8D, 0x07, 0x72, 0x1A, 0x80, 0xB0, 0x46, 0xD1, 0
      ↳xED, 0x91, 0xC8, 0xDF, 0x65, 0x22, 0x7B, 0x83, 0x8C, 0x52, 0x21, 0x39, 0xC3
      ↳, 0xCF, 0x2B, 0x81, 0x83, 0x47, 0xBA, 0x69, 0x99, 0x7D, 0x55, 0x10, 0xE2, 0
      ↳x42, 0x20, 0x5A, 0x1E, 0x4F, 0xF9, 0x89, 0x79, 0x79, 0xE8, 0x3E, 0xF0, 0x0E
      ↳, 0xAE, 0xC9, 0x1F, 0xD1, 0xA5, 0x6E, 0x76, 0x51, 0xA7, 0x1D, 0xAC, 0x4C, 0
      ↳x95, 0xCE, 0xAD, 0x62, 0x2C, 0x16, 0x42, 0x7A, 0xA9, 0xE5, 0xDE, 0x5B, 0x0F
      ↳, 0x1B, 0x86, 0x7E, 0x13, 0x14, 0xFA, 0xBC]),
        6539061469492441016689354886032563203708691591184200689622243916
        7064531780919514380281632846921748327401376785645856747098501116
        4731320141560872159788072341379891165468814121832966253456807056
        5670957357005535910249364520453474079932757719733461179300737446
        2187453855107399844686208466258487055584784880097360572
  );
};

// Test whether:
// - Our own signatures verify
// - External signatures verify
func pssIntegration() {
```

```
  let privkey = {
    p = 123962340477758208615763079097722555840558580158227198148980
          590820030107954455870834150213400900385065330064467800488622 0
              298208769407030621592782805913495 67;
    dP = 946102554031613872583708966444754035215054033741911626137 0
          105653708378012022646798806163298589138358221226095410419158 4
              190550443825171501006292676945651 15;
    q = 113260015705603436531325559584502180152445924227427279533 88
          118829320340011247336476724320332946680024284558963094235176 9
              296352210576359615341483190832514 51;
    dQ = 650540423151201635923018758783498720948246322793695940231 3
          028626456895479551756824287556499276925647407474450276275916 8
              891200181008070326690258201512250 23;
    qInv = 908318007732163861597335737574431401619670608562104945 81
          094583632276806287622404160753155745386311167932961323941416 5
              571533944253249042692580985683181576 5;
    d = 27406414852693374468560631691199018678861189081106247615557
          282189314072670205079466855712319284575504641435097592159289 1
              417694705231349634078710056909384005533832801988572238900657 0
              196911538607504748546362870674490742628071083414555212576153 5
              951945778980998333234622869998032152947970951978620467477155 4
              65773;
    modulusBits = 1024;
  };
  let pubkey = {
    n = 140399766294142553157668890732909410392912197983933412802 37
          454880333591656274761924640997563268609589992071695387300424 7
              047928225566692417483126874069180822367360298582588800027650 3
              850637082633511333944655364538132474431563532214991152031620 1
              764556992267405046843968695635278054636229696169412644286250 0
              971717;
    e = 65537;
    modulusBits = 1024;
  };
  let message : [Word8] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];

  // Test own signature
  var sig = Result.unwrapOk<[Word8], Text>(RSA.pssSign(privkey, message));
  assert Result.unwrapOk<Bool, Text>(RSA.pssVerify(pubkey, message, sig, null));

  // Test external (OpenSSL-generated) signature
  sig := [0x8b, 0x87, 0x36, 0x22, 0x3d, 0xdd, 0x16, 0x30, 0x72, 0x4d, 0xb9, 0x2a,
      ↳0x88, 0x72, 0x53, 0x93, 0x6d, 0x83, 0x29, 0xf1, 0x41, 0x16, 0xd5, 0xae, 0
      ↳x40, 0x23, 0x3, 0x2a, 0x38, 0xae, 0xfd, 0x92, 0x7a, 0xe0, 0xec, 0x74, 0x6d,
      ↳ 0xd7, 0x33, 0xe1, 0xd1, 0x17, 0xb3, 0xf, 0xed, 0xbc, 0x76, 0x1d, 0x75, 0
      ↳x30, 0xc6, 0x12, 0x54, 0x74, 0x81, 0x96, 0x47, 0x48, 0xd5, 0xdb, 0xf1, 0x28
      ↳, 0x66, 0x26, 0x45, 0xaf, 0xe1, 0xa0, 0xe3, 0xc5, 0xe1, 0x7, 0xf5, 0xf6, 0
      ↳xd7, 0x3d, 0x7a, 0x7e, 0x4f, 0x4c, 0x68, 0x34, 0xf6, 0xb, 0x8d, 0xab, 0x50,
      ↳ 0xf4, 0x73, 0x60, 0x5a, 0xcb, 0xae, 0xe5, 0x79, 0x93, 0x3c, 0x5d, 0xb8, 0
      ↳xd1, 0x8a, 0x8e, 0xba, 0x15, 0x60, 0x3b, 0x62, 0x58, 0xaf, 0xcc, 0x1b, 0xfb
      ↳, 0x68, 0xcb, 0xee, 0xc6, 0xb5, 0xb1, 0x4f, 0x3d, 0xec, 0x6b, 0xf5, 0xf7, 0
      ↳x50, 0xa8, 0x9f, 0x63];
  // OpenSSL uses (by default) a salt as big as possible, but we'll rely on
      ↳autodetection.
  assert Result.unwrapOk<Bool, Text>(RSA.pssVerify(pubkey, message, sig, null));
};

func primRSASP1() {
  // N = 23 * 17 = 391
  // lambda = lcm(p - 1, q - 1) = 176
  // e = 3 (3, 176 coprime)
  // d = modinv(e, lambda) = 59
```

```
  let privkey = { p = 23; dP = 15; q = 17; dQ = 11; qInv = 19; d = 59; modulusBits
      ↳ = 9; };

  assert T.natEq(Result.unwrapOk<Nat, Text>(RSA.primRSASP1(privkey, 139)), 24);
  assert T.natEq(Result.unwrapOk<Nat, Text>(RSA.primRSASP1(privkey, 176)), 90);

  // Error if m >= N
  assert T.textEq(Result.unwrapErr<Nat, Text>(RSA.primRSASP1(privkey, 391)), "
      ↳Message representative out of range, must be smaller than modulus");
};

func primRSAVP1() {
  // N = 23 * 17 = 391
  let pubkey = { n = 391; e = 3; modulusBits = 9; };

  assert T.natEq(Result.unwrapOk<Nat, Text>(RSA.primRSAVP1(pubkey, 24)), 139);
  assert T.natEq(Result.unwrapOk<Nat, Text>(RSA.primRSAVP1(pubkey, 90)), 176);

  // Error if s >= N
  assert T.textEq(Result.unwrapErr<Nat, Text>(RSA.primRSAVP1(pubkey, 391)), "
      ↳Signature representative out of range, must be smaller than modulus");
};

func zeroLeadingBits() {
  assert T.byteEq(RSA.zeroLeadingBits(0xFF, 0), 0xFF);
  assert T.byteEq(RSA.zeroLeadingBits(0xFF, 5), 0x07);
  assert T.byteEq(RSA.zeroLeadingBits(0xFF, 8), 0x00);
  assert T.byteEq(RSA.zeroLeadingBits(0xAD, 5), 0x05);
};

func getLeadingBits() {
  assert T.byteEq(RSA.getLeadingBits(0xFF, 3), 0xE0);
  assert T.byteEq(RSA.getLeadingBits(0x5B, 4), 0x50);
  assert T.byteEq(RSA.getLeadingBits(0x27, 8), 0x27);
  assert T.byteEq(RSA.getLeadingBits(0x27, 0), 0x00);
};

func ceilDiv() {
  assert T.natEq(RSA.ceilDiv(14, 3), 5);
  assert T.natEq(RSA.ceilDiv(15, 3), 5);
  assert T.natEq(RSA.ceilDiv(16, 3), 6);
  assert T.natEq(RSA.ceilDiv(24, 24), 1);
};

func xor() {
  let a : [Word8] = [0x25, 0xA4, 0xFF];
  let b : [Word8] = [0xFF, 0x91, 0x00];
  let expected : [Word8] = [0xDA, 0x35, 0xFF];

  let result : [Word8] = RSA.xor(a, b);
  assert T.byteArrayEq(result, expected);
};

pkcs15Encode();
pkcs15Sign();
pkcs15Verify();
pkcs15Integration();

mask();
detectPSSSaltLength();
emsaPSSEncode();
emsaPSSVerify();
pssSign();
```

```
  pssVerify();
  pssIntegration();

  primI2OSP();
  primOS2IP();
  primRSASP1();
  primRSAVP1();

  zeroLeadingBits();
  getLeadingBits();
  ceilDiv();
};
```

Appendix B

Sample application

This chapter lists the source code of a small sample application which was hosted on the IC, to test compatibility with OpenSSL. It can also serve as an example of how to use the libary.

The hardcoded RSA keypairs were truncated for brevity.

```
import Char "mo:base/Char";
import Iter "mo:base/Iter";
import Result "mo:base/Result";
import Word32 "mo:base/Word32";

import RSA "RSA";
import Base64 "Base64";

import SHA256 "mo:sha/SHA256";

import Debug "mo:base/Debug";

actor {
  public query func signPSS(encMsg : Text) : async Text {
    var msg : [Word8] = [];
    switch(Base64.decode(encMsg)) {
      case (#ok(m)) {
        msg := m;
      };
      case (#err(e)) {
        return "Error decoding message: " # e;
      };
    };

    Debug.print("Message bytes " # debug_show(msg));

    let privkey = {
      p = 12...;
      q = 11...;
      d = 27...;
      modulusBits = 1024;
    };


    switch (RSA.pssSign(privkey, msg)) {
      case (#ok(sig)) {
        Debug.print("Bytes: " # debug_show(sig));
        return Base64.encode(sig, #standard);
      };
      case (#err(e)) {
        return "Error signing message: " # e;
      };
    };
```

```
};

public query func signPKCS15(encMsg : Text) : async Text {
  var msg : [Word8] = [];
  switch(Base64.decode(encMsg)) {
    case (#ok(m)) {
      msg := m;
    };
    case (#err(e)) {
      return "Error decoding message: " # e;
    };
  };

  let privkey = {
    p = 11...;
    q = 10...;
    d = 41...;
    modulusBits = 1024;
  };

  switch (RSA.pkcs15Sign(privkey, msg)) {
    case (#ok(sig)) {
      Debug.print("Bytes: " # debug_show(sig));
      return Base64.encode(sig, #standard);
    };
    case (#err(e)) {
      return "Error: " # e;
    };
  };
};

public query func verifyPKCS15(encMsg : Text, encSig : Text) : async Text {
  var msg : [Word8] = [];
  var sig : [Word8] = [];

  switch(Base64.decode(encMsg)) {
    case (#ok(m)) {
      msg := m;
    };
    case (#err(e)) {
      return "Error decoding message: " # e;
    };
  };

  switch(Base64.decode(encSig)) {
    case (#ok(s)) {
      sig := s;
    };
    case (#err(e)) {
      return "Error decoding signature: " # e;
    };
  };

  let pubkey = {
    n = 12...;
    e = 65537;
    modulusBits = 1024;
  };

  switch (RSA.pkcs15Verify(pubkey, msg, sig)) {
    case (#ok(Bool)) {
      return "Signature verified";
    };
```

```
      case (#err(e)) {
        return "Signature failed to verify: " # e;
      };
    };
  };

  public query func verifyPSS(encMsg : Text, encSig : Text) : async Text {
    var msg : [Word8] = [];
    var sig : [Word8] = [];

    switch(Base64.decode(encMsg)) {
      case (#ok(m)) {
        msg := m;
      };
      case (#err(e)) {
        return "Error decoding message: " # e;
      };
    };

    switch(Base64.decode(encSig)) {
      case (#ok(s)) {
        sig := s;
      };
      case (#err(e)) {
        return "Error decoding signature: " # e;
      };
    };

    let pubkey = {
      n = 14...;
      e = 65537;
      modulusBits = 1024;
    };

    switch (RSA.pssVerify(pubkey, msg, sig, null)) {
      case (#ok(Bool)) {
        return "Signature verified";
      };
      case (#err(e)) {
        return "Signature failed to verify: " # e;
      };
    };
  };
};
```

Appendix C

Library performance

This chapter lists the raw data measurements collected during the performance evaluation as well as the the code used to analyze it, to help with reproducability.

C.1 Profiling application

```
import Char "mo:base/Char";
import Iter "mo:base/Iter";
import Result "mo:base/Result";
import Random "mo:base/Random";
import Word32 "mo:base/Word32";

import RSA "RSA";
import Base64 "Base64";

import SHA256 "mo:sha/SHA256";

import Debug "mo:base/Debug";

actor {
  var privkey = {
    d = 23...;
    p = 27...;
    q = 24...;
    dP = 10...;
    dQ = 13...;
    qInv = 21...;
    modulusBits = 4096;
  };

  var pubkey = {
    n = 66...;
    e = 65537;
    modulusBits = 4096;
  };

  var iterCountSign : Nat = 20;
  var iterCountVerify : Nat = 100;
  var msg : [Word8] = [0x65, 0x68, 0x6c, 0x6c, 0x20, 0x6f, 0x77, 0x6c, 0x72, 0x0, 0
      ↳x64];

  // No-op function to test latency
  public query func noop() : async (Nat) {
    return 42
  };
```

```
public func profileRandom() : async () {
  let bytes = await Random.blob();
  Debug.print("Got randomness: " # Base64.encode(Iter.toArray<Word8>(bytes.bytes()
      ↳), #standard));
};

// Profile PSS signing
public query func profilePSSSign() : async () {
  for (i in Iter.range(0, iterCountSign)) {
    switch (RSA.pssSign(privkey, msg)) {
      case (#ok(sig)) {
        // Pass
      };
      case (#err(e)) {
        Debug.print("Error signing message: " # e);
      };
    };
  };
};

// Profile PSS verification. Contains one PSS signing operation.
public query func profilePSSVerify() : async () {
  var sig : [Word8] = [];
  switch (RSA.pssSign(privkey, msg)) {
    case (#ok(sig2)) {
      sig := sig2;
    };
    case (#err(e)) {
      Debug.print("Error signing message: " # e);
      return;
    };
  };


  for (i in Iter.range(0, iterCountVerify)) {
    switch (RSA.pssVerify(pubkey, msg, sig, null)) {
      case (#ok(Bool)) {
        // Pass
      };
      case (#err(e)) {
        Debug.print("Signature failed to verify: " # e);
      };
    };
  };
};

// Profile PKCS15 signing
public query func profilePKCS15Sign() : async () {
  for (i in Iter.range(0, iterCountSign)) {
    switch (RSA.pkcs15Sign(privkey, msg)) {
      case (#ok(sig)) {
        // Pass
      };
      case (#err(e)) {
        Debug.print("Error signing message: " # e);
      };
    };
  };
};

// Profile PKCS15 verification. Contains one PKCS15 signing operation.
public query func profilePKCS15Verify() : async () {
  var sig : [Word8] = [];
```

```
  switch (RSA.pkcs15Sign(privkey, msg)) {
    case (#ok(sig2)) {
      sig := sig2;
    };
    case (#err(e)) {
      Debug.print("Error signing message: " # e);
      return;
    };
  };


  for (i in Iter.range(0, iterCountVerify)) {
    switch (RSA.pkcs15Verify(pubkey, msg, sig)) {
      case (#ok(Bool)) {
        // Pass
      };
      case (#err(e)) {
        Debug.print("Signature failed to verify: " # e);
      };
    };
  };
};

public query func signPSS(encMsg : Text) : async Text {
  var msg : [Word8] = [];
  switch(Base64.decode(encMsg)) {
    case (#ok(m)) {
      msg := m;
    };
    case (#err(e)) {
      return "Error decoding message: " # e;
    };
  };

  Debug.print("Message bytes " # debug_show(msg));

  switch (RSA.pssSign(privkey, msg)) {
    case (#ok(sig)) {
      Debug.print("Bytes: " # debug_show(sig));
      return Base64.encode(sig, #standard);
    };
    case (#err(e)) {
      return "Error signing message: " # e;
    };
  };
};

public query func signPKCS15(encMsg : Text) : async Text {
  var msg : [Word8] = [];
  switch(Base64.decode(encMsg)) {
    case (#ok(m)) {
      msg := m;
    };
    case (#err(e)) {
      return "Error decoding message: " # e;
    };
  };

  switch (RSA.pkcs15Sign(privkey, msg)) {
    case (#ok(sig)) {
      Debug.print("Bytes: " # debug_show(sig));
      return Base64.encode(sig, #standard);
    };
```

```
    case (#err(e)) {
      return "Error: " # e;
    };
  };
};

public query func verifyPKCS15(encMsg : Text, encSig : Text) : async Text {
  var msg : [Word8] = [];
  var sig : [Word8] = [];

  switch(Base64.decode(encMsg)) {
    case (#ok(m)) {
      msg := m;
    };
    case (#err(e)) {
      return "Error decoding message: " # e;
    };
  };

  switch(Base64.decode(encSig)) {
    case (#ok(s)) {
      sig := s;
    };
    case (#err(e)) {
      return "Error decoding signature: " # e;
    };
  };

  switch (RSA.pkcs15Verify(pubkey, msg, sig)) {
    case (#ok(Bool)) {
      return "Signature verified";
    };
    case (#err(e)) {
      return "Signature failed to verify: " # e;
    };
  };
};

public query func verifyPSS(encMsg : Text, encSig : Text) : async Text {
  var msg : [Word8] = [];
  var sig : [Word8] = [];

  switch(Base64.decode(encMsg)) {
    case (#ok(m)) {
      msg := m;
    };
    case (#err(e)) {
      return "Error decoding message: " # e;
    };
  };

  switch(Base64.decode(encSig)) {
    case (#ok(s)) {
      sig := s;
    };
    case (#err(e)) {
      return "Error decoding signature: " # e;
    };
  };

  switch (RSA.pssVerify(pubkey, msg, sig, null)) {
    case (#ok(Bool)) {
      return "Signature verified";
```

```
      };
      case (#err(e)) {
        return "Signature failed to verify: " # e;
      };
    };
  };
};
```

C.2 Measurements

Key size	Algorithm	Cluster	Iter count	Operation	Time(s)
512	pss	local	500	sign	7.526
512	pss	local	500	sign	7.672
512	pss	local	500	sign	7.534
512	pss	local	500	sign	7.514
512	pss	local	2500	verify	4.430
512	pss	local	2500	verify	4.392
512	pss	local	2500	verify	4.399
512	pss	local	2500	verify	4.238
512	pkcs15	local	500	sign	7.289
512	pkcs15	local	500	sign	7.539
512	pkcs15	local	500	sign	7.423
512	pkcs15	local	500	sign	7.509
512	pkcs15	local	2500	verify	3.747
512	pkcs15	local	2500	verify	3.499
512	pkcs15	local	2500	verify	3.410
512	pkcs15	local	2500	verify	3.438
1024	pss	local	100	sign	4.78
1024	pss	local	100	sign	4.94
1024	pss	local	100	sign	5.29
1024	pss	local	100	sign	5.14
1024	pss	local	500	verify	2.32
1024	pss	local	500	verify	2.1
1024	pss	local	500	verify	2.13
1024	pss	local	500	verify	2.22
1024	pkcs15	local	100	sign	4.91
1024	pkcs15	local	100	sign	5
1024	pkcs15	local	100	sign	5.19
1024	pkcs15	local	100	sign	5.07
1024	pkcs15	local	500	verify	1.91
1024	pkcs15	local	500	verify	1.86
1024	pkcs15	local	500	verify	1.83
1024	pkcs15	local	500	verify	1.86
2048	pss	local	100	sign	22.914
2048	pss	local	100	sign	23.721
2048	pss	local	100	sign	23.59
2048	pss	local	100	sign	23.578
2048	pss	local	500	verify	7.178
2048	pss	local	500	verify	6.938
2048	pss	local	500	verify	6.998
2048	pss	local	500	verify	6.987

Key size	Algorithm	Cluster	Iter count	Operation	Time(s)
2048	pkcs15	local	100	sign	22.813
2048	pkcs15	local	100	sign	23.507
2048	pkcs15	local	100	sign	23.787
2048	pkcs15	local	100	sign	23.9
2048	pkcs15	local	500	verify	6.854
2048	pkcs15	local	500	verify	6.415
2048	pkcs15	local	500	verify	6.575
2048	pkcs15	local	500	verify	6.502
4096	pss	local	20	sign	30.411
4096	pss	local	20	sign	31.223
4096	pss	local	20	sign	27.672
4096	pss	local	20	sign	27.531
4096	pss	local	100	verify	7.169
4096	pss	local	100	verify	6.943
4096	pss	local	100	verify	7.287
4096	pss	local	100	verify	8.131
4096	pkcs15	local	20	sign	27.077
4096	pkcs15	local	20	sign	27.51
4096	pkcs15	local	20	sign	26.716
4096	pkcs15	local	20	sign	27.217
4096	pkcs15	local	100	verify	7.127
4096	pkcs15	local	100	verify	6.940
4096	pkcs15	local	100	verify	6.862
4096	pkcs15	local	100	verify	6.997
512	pss	remote	500	sign	12.363
512	pss	remote	500	sign	14.181
512	pss	remote	500	sign	13.091
512	pss	remote	500	sign	12.397
512	pss	remote	2500	verify	7.582
512	pss	remote	2500	verify	7.967
512	pss	remote	2500	verify	7.364
512	pss	remote	2500	verify	7.054
512	pkcs15	remote	500	sign	13.915
512	pkcs15	remote	500	sign	12.719
512	pkcs15	remote	500	sign	12.295
512	pkcs15	remote	500	sign	13.763
512	pkcs15	remote	2500	verify	5.633
512	pkcs15	remote	2500	verify	6.2
512	pkcs15	remote	2500	verify	5.867
512	pkcs15	remote	2500	verify	5.958
1024	pss	remote	100	sign	7.863
1024	pss	remote	100	sign	7.705
1024	pss	remote	100	sign	7.880
1024	pss	remote	100	sign	7.930
1024	pss	remote	500	verify	4.017
1024	pss	remote	500	verify	3.458
1024	pss	remote	500	verify	3.554
1024	pss	remote	500	verify	3.669
1024	pkcs15	remote	100	sign	7.529
1024	pkcs15	remote	100	sign	8.041

Key size	Algorithm	Cluster	Iter count	Operation	Time(s)
1024	pkcs15	remote	100	sign	7.634
1024	pkcs15	remote	100	sign	7.798
1024	pkcs15	remote	500	verify	3.603
1024	pkcs15	remote	500	verify	3.677
1024	pkcs15	remote	500	verify	3.145
1024	pkcs15	remote	500	verify	3.124
2048	pss	remote	100	sign	34.016
2048	pss	remote	100	sign	35.783
2048	pss	remote	100	sign	34.929
2048	pss	remote	100	sign	33.207
2048	pss	remote	500	verify	10.305
2048	pss	remote	500	verify	10.042
2048	pss	remote	500	verify	10.303
2048	pss	remote	500	verify	10.417
2048	pkcs15	remote	100	sign	37.707
2048	pkcs15	remote	100	sign	35.174
2048	pkcs15	remote	100	sign	34.103
2048	pkcs15	remote	100	sign	32.804
2048	pkcs15	remote	500	verify	9.245
2048	pkcs15	remote	500	verify	10.006
2048	pkcs15	remote	500	verify	9.627
2048	pkcs15	remote	500	verify	9.007
4096	pss	remote	20	sign	37.017
4096	pss	remote	20	sign	35.293
4096	pss	remote	20	sign	35.966
4096	pss	remote	20	sign	37.074
4096	pss	remote	100	verify	9.955
4096	pss	remote	100	verify	10.122
4096	pss	remote	100	verify	9.919
4096	pss	remote	100	verify	10.124
4096	pkcs15	remote	20	sign	38.803
4096	pkcs15	remote	20	sign	34.788
4096	pkcs15	remote	20	sign	33.711
4096	pkcs15	remote	20	sign	34.474
4096	pkcs15	remote	100	verify	9.926
4096	pkcs15	remote	100	verify	9.270
4096	pkcs15	remote	100	verify	9.358
4096	pkcs15	remote	100	verify	9.513
0	noop	remote	0	noop	0.833
0	noop	remote	0	noop	0.543
0	noop	remote	0	noop	0.557
0	noop	remote	0	noop	0.556
0	noop	remote	0	noop	0.520
0	noop	remote	0	noop	0.549
0	noop	remote	0	noop	0.887
0	noop	remote	0	noop	1.007
0	noop	remote	0	noop	0.457
0	noop	remote	0	noop	0.619
0	noop	remote	0	noop	0.158
0	noop	remote	0	noop	1.007

Key size	Algorithm	Cluster	Iter count	Operation	Time(s)
0	noop	remote	0	noop	0.214
0	noop	remote	0	noop	0.914
0	noop	remote	0	noop	0.496
0	noop	local	0	noop	0.042
0	noop	local	0	noop	0.17
0	noop	local	0	noop	0.019
0	noop	local	0	noop	0.029
0	noop	local	0	noop	0.050
0	noop	local	0	noop	0.042
0	noop	local	0	noop	0.038
0	noop	local	0	noop	0.049
0	noop	local	0	noop	0.038
0	noop	local	0	noop	0.041
0	noop	local	0	noop	0.040
0	noop	local	0	noop	0.039
0	noop	local	0	noop	0.038
0	noop	local	0	noop	0.08
0	noop	local	0	noop	0.038

C.3 Analysis

```
library(ggplot2)
library(dplyr)
library(viridis)

df = read.csv("../data/performance.csv")

df$modulus_bitlength = as.numeric(df$modulus_bitlength)

df$algorithm = factor(df$algorithm)
levels(df$algorithm) = list("RSASSA-PSS" = "pss", "RSASSA-PKCS1-v1_5" = "pkcs15", "
    ↳noop" = "noop")

df$cluster = factor(df$cluster)
levels(df$cluster) = list("Local replica" = "local", "IC" = "remote")

df$operation = factor(df$operation)
levels(df$operation) = list("Sign" = "sign", "Verify" = "verify")

df_noop = df %>% filter(algorithm == "noop")
df_library = df %>% filter(algorithm != "noop")

noop_stats = df_noop %>% group_by(cluster) %>% summarise(avg_time = mean(time), sd =
    ↳ sd(time), n = n())
latency_local = (noop_stats %>% filter(cluster == 'Local replica'))$avg_time
latency_remote = (noop_stats %>% filter(cluster == 'IC'))$avg_time

# Compensate measured time for no-op latency
df_library = df_library %>% mutate(time = case_when(cluster == 'Local replica' ~
    ↳time - latency_local, cluster == 'IC' ~ time - latency_remote))

# And normalize to one operation
df_library$time = df_library$time / df_library$iter_count

library_stats = df_library %>% group_by(modulus_bitlength, cluster, algorithm,
    ↳operation) %>% summarise(avg_time = mean(time), sd = sd(time), n = n())
```

```
ggplot(library_stats, aes(x = modulus_bitlength, y = avg_time, color = algorithm)) +
  geom_point() +
  geom_line() +
  scale_x_continuous(breaks = c(512, 1024, 2048, 4096)) +
  scale_y_log10() +
  facet_wrap(vars(operation, cluster)) +
  theme_minimal() +
  theme(legend.position = "bottom") +
  labs(title = "Performance of signing and verification operations", y = "Execution
      ↳time [s]", x = "Key size [bits]", color = "Algorithm") +
  ggsave("../../resources/library_performance.png", width = 20, units = 'cm', dpi =
      ↳'print')
```

Bibliography

[BD15] Elaine B. Barker and Quynh H. Dang. *Recommendation for Key Management Part 3: Application-Specific Key Management Guidance*. NIST SP 800-57Pt3r1. National Institute of Standards and Technology, Jan. 2015, NIST SP 800–57Pt3r1. DOI: 10.6028/NIST.SP.800-57Pt3r1. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf (visited on 11/20/2020).

[BR95] Mihir Bellare and Phillip Rogaway. "Optimal Asymmetric Encryption". In: *Advances in Cryptology — EUROCRYPT'94*. Ed. by Alfredo De Santis. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 950. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 92–111. ISBN: 978-3-540-60176-0 978-3-540-44717-7. DOI: 10.1007/BFb0053428. URL: http://link.springer.com/10.1007/BFb0053428 (visited on 03/07/2021).

[Bre21] Joachim Breitner. *Can We Somehow Achieve or Build a Canister with a Interface Bahaving like a Classic Web Server*. DFINITY Developer Forum. Mar. 3, 2021. URL: https://forum.dfinity.org/t/can-we-somehow-achieve-or-build-a-canister-with-a-interface-bahaving-like-a-classic-web-server/1257/29 (visited on 03/07/2021).

[DFI21] DFINITY Foundation. *DFINITY Developer's Guide*. DFINITY Foundation, 2021. URL: https://sdk.dfinity.org/docs/developers-guide (visited on 03/07/2021).

[Gro21] Jens Groth. *Non-Interactive Distributed Key Generation Andkey Resharing*. Draft. DFINITY Foundation, Mar. 16, 2021. URL: https://eprint.iacr.org/2021/339.pdf (visited on 04/27/2021).

[Hau20a] Enzo Haussecker. *Where Is the CanCan App Source?* Dec. 7, 2020. URL: https://forum.dfinity.org/t/where-is-the-cancan-app-source/1628/2.

[Hau20b] Enzoh Haussecker. *Motoko-Sha*. Version e9962f3. July 11, 2020. URL: https://github.com/enzoh/motoko-sha (visited on 02/07/2021).

[Jos06] S Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Internet Engineering Task Force (IETF), Oct. 2006. URL: https://tools.ietf.org/html/rfc4648.

[Knu96] Donald Ervin Knuth. *Seminumerical Algorithms*. 2. ed., 25. print. The Art of Computer Programming Donald E. Knuth ; Vol. 2. Reading, Mass.: Addison-Wesley, 1996. 688 pp. ISBN: 978-0-201-03822-4.

[MP00] Bellare Mihir and Rogaway Phillip. *PSS: Provably Secure Encoding Method for Digital Signatures*. Apr. 2000. URL: https://web.archive.org/web/20170810025803/http://grouper.ieee.org/groups/1363/P1363a/contributions/pss-submission.pdf (visited on 01/17/2021).

[Mon85] Peter L. Montgomery. "Modular Multiplication without Trial Division". In: *Mathematics of Computation* 44.170 (May 1, 1985), pp. 519–519. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1985-0777282-X. URL: http://www.ams.org/jourcgi/jour-getitem?pii=S0025-5718-1985-0777282-X (visited on 11/20/2020).

[Mor+16] K Moriarty et al. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Internet Engineering Task Force (IETF), Nov. 2016. URL: https://tools.ietf.org/html/rfc8017.

[Mor20a] Morrolan. *Garbage Collection in Motoko*. Sept. 14, 2020. URL: https://forum.dfinity.org/t/garbage-collection-in-motoko/1263.

[Mor20b] Morrolan. *Logical Bitshift on 'WordN' Wraps Around*. Oct. 7, 2020. URL: https://forum.dfinity.org/t/logical-bitshift-on-wordn-wraps-around/1385.

[Mor21] Morrolan. *Motoko's Type System & Converting between Types*. Feb. 7, 2021. URL: https://forum.dfinity.org/t/motokos-type-system-converting-between-types/1957.

[Pol+02] W Polk et al. *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 3279. Internet Engineering Task Force (IETF), Apr. 2002. URL: https://tools.ietf.org/html/rfc3279.

[QC82] J.-J. Quisquater and C. Couvreur. "Fast Decipherment Algorithm for RSA Public-Key Cryptosystem". In: *Electronics Letters* 18.21 (1982), p. 905. ISSN: 00135194. DOI: 10.1049/el:19820617. URL: https://digital-library.theiet.org/content/journals/10.1049/el_19820617 (visited on 01/17/2021).

[RM18] E Rescorla and Mozilla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force (IETF), Aug. 2018.

[Sch+05] J Schaad et al. *Additional Algorithms and Identifiers for RSA Cryptography for Use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 4055. Internet Engineering Task Force (IETF), June 2005. URL: https://tools.ietf.org/html/rfc4055.

[Sch15] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 20th anniversary edition. Indianapolis, IN: Wiley, 2015. 758 pp. ISBN: 978-1-119-09672-6.

[Squ19] Squeamish Ossifrage. *Why Does OpenSSL Differentiate between PSS and Non-PSS for Private Key Generation?* crypto.stackexchange.com. May 9, 2019. URL: https://crypto.stackexchange.com/questions/70413/why-does-openssl-differentiate-between-pss-and-non-pss-for-private-key-generatio (visited on 03/14/2021).

[TS02] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Upper Saddle River, N.J: Prentice Hall, 2002. 803 pp. ISBN: 978-0-13-088893-8.

[Tur10] S Turner. *Asymmetric Key Packages*. RFC 5958. Internet Engineering Task Force (IETF), Aug. 2010. URL: https://tools.ietf.org/html/rfc5958.

[Wil20] Dominic Williams. "The Internet Computer in 10 Minutes: Sodium Launch Event for China". 2020. URL: https://drive.google.com/file/d/11HDTsay6-Jke-fncieIGzhEE7Ip0UME
view.

[Wil21] Dominic Williams. "The Internet Computer: A Primer for Entrepreneurs". Feb. 15, 2021. URL: https://dfinity.org/deck/ (visited on 03/07/2021).

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Lyss, 4.5.21

Ort/Datum

Unterschrift