

RNS Generator:

User Guide

This RNS Generator emulates the use of the processor described in [Pedro Thesis] in order to generate the required assembly instructions for two the major asymmetrical cryptographic algorithms: RSA and ECC.

The main window is depicted in Figure 1, where the selected algorithm is RSA. Here, the input fields are in the form of a decimal integer, as described next:

- **Msg** – is the message to cipher (or decipher);
- **Key** – is the private or public key, depending on the operation;
- **Modulo** – is the value of the RSA *modulo*.

The above fields can be randomized by selecting the button **Random**, for demonstration purposes. The following input is always required:

- **Bits** – is the number of bits of the **Modulo**, **Key** and **Modulo**, denoted $\langle M \rangle$;
- **Number of bits per channel** – is the number of bits on each RNS channel, denoted $\langle n \rangle$.

By clicking **Start** the tool then generates the required number of channels per RNS base and displays them in the section bellow. Each white square represents a RNS channel and displays the instruction being computed by such channel. The first line represents the operation, while the other represent the *destination register*, *register A* and *register B*, in this order.

Now the user has the possibility to perform a step by step analysis of the instructions being sent to each RNS channel by clicking on the button **Next**, or, in the other hand, can jump to the end of the computations by clicking the **End** button.

As soon as the result is computed it is displayed in the **Result** field. At the same time, 5 output files are written to the folder where the RNS generator is located. They are:

- “key $\langle M \rangle$ - $n\langle n \rangle$ -assemblyRSA.txt” – containing the pseudo-code assembly instructions;
- “key $\langle M \rangle$ - $n\langle n \rangle$ -codeRSA.txt” – containing the assembly instructions in hexadecimal to be fed to the processor;
- “key $\langle M \rangle$ - $n\langle n \rangle$ -dataRSA.txt” – containing the application specific constants and data to be fed to the processor;
- “key $\langle M \rangle$ - $n\langle n \rangle$ -moduliRSA.txt” – containing information about the RNS moduli set in use, as well as original the **Msg**, **Key** and **Modulo**;
- “key $\langle M \rangle$ - $n\langle n \rangle$ -resultRSA.txt” – containing the expected result, in hexadecimal.

Figure 1 – RNS Generator Interface: RSA

Alternatively, an equivalent for ECC can be selected in the upper left corner of the window. The window is depicted in Figure 2, where the input fields are:

- **Curve ‘a’** – is the parameter a of the elliptic curve, as in $y^2 = x^3 + ax + b$;
- **Curve ‘b’** – is the parameter b of the elliptic curve equation;
- **Point ‘x’ coordinate** – is the ‘x’ coordinate of the curve point to be multiplied by a scalar;
- **Point ‘y’ coordinate** – is the ‘y’ coordinate of the curve point to be multiplied by a scalar;
- **Scalar to multiply** – is the scalar to multiply by the curve point mentioned above;
- **GF(p)** – is the modulo on which the elliptic curve is built.

For demonstration purposes, an example setup for the above mentioned parameters can be selected by activating the **Example** button. Similarly to the RSA tab, the following input is always required:

- **Bits** – is the number of bits of the **Modulo**, **Key** and **Modulo**, denoted $\langle M \rangle$;

- **Number of bits per channel** – is the number of bits on each RNS channel, denoted $\langle n \rangle$.

The rest of the behavior is to the RSA counterpart.

For more information on how these algorithms are implemented please refer to [Juvenal Thesis].

The screenshot shows a software window titled "RNS" with a standard Windows interface (minimize, maximize, close buttons). Inside the window, there are two tabs: "RSA" and "ECC", with "ECC" currently selected. The interface includes several input fields and controls:

- Curve 'a' and Curve 'b' input fields.
- Point 'x' coordinate and Point 'y' coordinate input fields.
- An "Example" radio button and a "Scalar to multiply" input field.
- A "GF(p)" input field.
- A "Bits" input field.
- A "Number of bits per channel" input field.
- A blue "Start" button.
- A large rectangular area below the inputs, divided into three sections labeled "Base 1", "Base 2", and "Redundant".
- At the bottom, there are two input fields for "Result 'x' coordinate:" and "'y' coordinate:", followed by "End" and "Next" buttons.

Figure 2 – RNS Generator Interface: ECC

Developing for the RNS processor

A user intending to develop a different application for the RNS processor can write his algorithm using the provided functions, driving the tool to generate the correspondent assembly instructions for the considered RNS processor.

The class `Rns` corresponds to a number represented in RNS format in the registers of the processor. It offers:

- Conversion from binary to RNS, using the constructor `Rns(String value, String name);`
- RNS addition, subtraction and multiplication between two `Rns` objects, using respectively `add(Rns b, Rns destination)`, `sub(Rns b, Rns destination)` and `mult(Rns b, Rns destination)`;
- Modular multiplication based on the RNS Montgomery Multiplication, using `modularMult(Rns b, Rns destination)`.
Note that the modular operation can only be performed in relation to a single modulo, defined in the beginning of the program. Both operands must be represented in the Montgomery Domain (MD);
- Modulo operation, based in the RNS Montgomery Multiplication, using `modulo(Rns destination)`. Note that the modulo is the same as described before. The operand must be represented in the MD;
- Conversion from RNS to binary, using `toIntMRC()`.

An RNS program computing only a multiplication will be detailed as example, defining the generic workflow for developing for the tool:

1. The developer starts configuring the hardware by selecting the operand bit width and the bit width of each RNS channel, as well as the name for the XML file with the hardware description:
`Main.createModuliSet(64, 16, "mult.xml");`
In this case, the operands will be 64 bits long, and each RNS channel will have 16 bits. The hardware configuration will be written to `mult.xml`.

Alternatively, a developer may use

`Main.loadModuliSet("mult.xml");` in order to load the hardware configuration from an existing XML file. The structure for the XML file is described in the next subsection.

2. Now, the developer calls `Main.createFiles("MULT");` in order for the program to create the output files, in this case appended by `MULT`.
3. Next, the user creates the RNS numbers, converting from decimal to RNS:
`Rns A = new Rns("70000", "A");`
`Rns B = new Rns("51000", "B");`

```
BigInteger C_bin;  
Rns C = new Rns("C");
```

Here, the variable C is reserved without setting its value.

4. The RNS multiplication $C = A \times B$ can now be performed:
`A.mult(B, C);`
5. Following, the number must be converted back to binary:
`C_bin = C.toIntMRC();`
6. Finally, the user must close the output files by calling:
`Main.closeFiles();`

This example, and another ones performing Modular Multiplication, can be found in the project package examples.

For more information on how these methods are implemented and the RNS Modular Multiplication please refer to [\[Juvenal Thesis\]](#).

XML file

The XML file containing the hardware configuration must start with the opening tag `<RNS>`. Afterwards, it is expected the information about the number of registers in the co-processor, between the tag `<maxNumberOfRegisters>`, then the information about the size of the operands the RNS is set to represent, between the tag `<dynamicRange_nBits>`, and then the number of bits per modulus, between the tag `<channelLength>`.

Following, the opening tag `<bases>` is expected setting as attribute the `numberOfbases` parameter, indicating the number of bases of the co-processor. In the moment only the value "2" is supported.

Next, in order to specify the moduli set for each base, the opening tag `<moduliSet>` is used with the attribute `numberOfModuli` indicating the number of moduli per base for the first base. This number is then used to read the values between tags `<k>` indicating the value for k as in $2^{\text{bitsPerModulus} - k}$.

After setting the value of k for `numberOfModuli` times, the closing tag `</moduliSet>` is expected. A new `<moduliSet>` following the same rules as before for the second base.

Now that both bases are described, the closing tag `</bases>` is expected.

The document can end now using the closing tag `<RNS>`, or optionally the RNS constants for the co-processor inner working can be set using the opening tag `<constants>`. Each constant is declared opening the `<constant>` tag with attributes `name` and `numberOfBases` representing respectively the name of the constant and the number of bases on which this constant is to be set.

The opening tag `<base>` is used to set the values for the RNS channels, represented between tags `<chi>`, where i is the number of the channel.

Unless generated by the tool, the XML file without constants is the most commonly used for providing the hardware configuration. An example of a XML file without constants is presented next. Refer to a XML file generated by the tool for an example with constants configuration.

```
<RNS>
  <maxNumberOfRegisters>128</maxNumberOfRegisters>
  <dynamicRange_nBits>64</dynamicRange_nBits>
  <channelLength>16</channelLength>
  <bases numberOfbases="2">
    <moduliSet numberOfModuli="5">
      <k>5</k>
      <k>15</k>
      <k>27</k>
      <k>39</k>
      <k>47</k>
    </moduliSet>
    <moduliSet numberOfModuli="5">
      <k>3</k>
      <k>9</k>
      <k>17</k>
      <k>33</k>
      <k>45</k>
    </moduliSet>
  </bases>
</RNS>
```