Verification - Romulus-N 1.3 Hardware Implementation

Aadam and Hawa Dirie

December 2021

Contents

1 Use of Tools

The use of tools for verification mainly included Python in order to use, as well as modify the existing Romulus-N implementation and Vivado simulation in order to verify if certain entities had the expected behavior. We modified the reference implementation such that it had a more powerful debugging mode and We could see results after each and every operation for each and every single round. This was paramount in the first stage of verification that Skinny-128-384+ was working properly due to the fact that after each round, if there are any mistakes in a permutation, or substitution it would permeate very quickly and was initially very difficult to debug just using Vivado. As for generating test vectors, We mainly used the one test vector specified in the Romulus-N 1.3 specification as well as an additional one that was generated myself. As mentioned in *assumptions*, We aimed to verify just two blocks of associated data and 2 blocks of message with no padding needed in order to at least know majority of circuit is working and only small tweaks in top-level controller would be needed.

2 Strategy of Verification

The strategy We employed was very simple. Even the design methodology itself was predicated on a building block approach. We aimed to design very modular VHDL code with use of generics to help alleviate some of the development costs and issues that small tweaks might cause. Nonetheless, We always strived to verify each of the individual building blocks of other entities and go on from there. That is where the first testbench, $E\_K\_Controller\_TB.vhd$, was used to verify and tweak many of the underlying issues we had with some of the round functions, LFSRs, as well as overall Skinny-128-384 tweakable block cipher. Once we were able to verify the datapath of the E_K function and its other subentities was working as expected, we shifted focus to design a controller for the TBC to reduce complexity of the top level controller having to deal with so many other control signals. This was very difficult to do at first due to some of the updates needing to be done prior to a round messing up results but we got it to work.

3 Software Debugging

We used extensive software debugging to help us with verification. Namely, we modified the open source software implementation provided by designers in python in order to have more detailed information of what vectors we expect after each specific round function in Figure.6. This of course, led to us learning which rounds in the skinny verification test led to the wrong modification of the internal state. We even learned to modify some existing mistakes in our design through the use of this software and adding these debug flags and extending them to include more information was really helpful for verification. We also

extended the debugging of Encryption of Romulus-N in terms of knowing what each ρ function and Skinny TBC call results are for comparison with testbench as seen in Figure.7.

4 Highest Level Entity Verified

The highest level entity that was fully verified was *EK_Skinnyromn.vhd*. This file incorporates the interface between the Skinny datapath and controller. Subsequently we began to attempt to verify the entire Datapath which includes a ρ function as well as some additional registers to load, process, and output data. We were able to verify the following for the entire datapath but not the entirety of the circuit.

- Complete functionality of loading Nonce, Key, and Blocks of PlainText/CipherText (Assuming len8 is supplied to datapath correctly)
- Testbench verifying that the LFSR for entire algorithm was valid, again verifying Skinny function worked, output PISO was working as expected, Initial values of ρ function also looked correct. Main issue looked to be with the Rho function and Tweakey Encoding where after each increment some of the values could have been off which would cause subsequent TBC calls to be off results wise.

we did not do thorough testing at all of the main ASM chart aka Top Level Controller of the CryptoCore because we shifted our attention in trying to verify the entire datapath first. However a sizable portion of the ASM Chart was implemented in VHDL and is synthesizable.

5 Skinny-128-384+ Verification

Here is the successful verification of Skinny-128-384+, the Tweakable Block Cipher (TBC) as utilized in the Romulus-N 1.3 variant. Below are the test vectors that were used to make this happen. We utilized a generic in the design to denote if testing could be done to assign the Tweakey directly and we also manually instantiated the entities and their corresponding datapath, this is clearly described here in Figure.1. We we successful in having the PlainText and CipherText match after a long series of debugging to ensure each round function and Tweakey update was indeed accurate. The Testbench basically initializes Skinny with the Start signal E_start and waits around 40 or so clock cycles for all rounds to complete and when E_done is signaled as seen in Figure.2.

```
E_K_CONTROLLER_INSTANTIATION_TB : entity work.E_K_controller
    port map(clk => clk,
             e_start => e_start,
             e_done => e_done,
             selInitial => selInitial,
             enTK => enTK,
             enRound => enRound,
             enIS => enIS,
             enAC => enAC);

-- Testing entire E_K component
E_K_FUNCTION_INSTANTIATION_TB : entity work.E_K
     generic map (TESTING => 1)
     port map(clk => clk,
              enIS => enIS,
              enAC => enAC,
              enTK => enTK,
              selInitial => selInitial,
              S => S,
              K => K,
              T => T,
              B => B,
              D => D,
              S_E => S_E);
 LFSRD_FUNCTION_INSTANTIATION_TB : entity work.LFSRD
      port map(clk => clk,
               selDD => selInitial,
               enDD => enDD,
               D => D);
```

Figure 1: Instantiating Entities used in Verification of Skinny in TestBench

```
-- Since E_K function was instantiated with Testing = 1, we set Tweakey to a specific value which was shown
-- in the Skinny-128-384+ test vector in Romulus v1.3 description, we also assign S to this same specified value

UUT : process
    begin
        S <= x"A3994B66AD85A3459F44E92B08F550CB";
        -- S <= (others => '0');
        K <= (others => '0');
        T <= (others => '0');
        B <= (others => '0');
        e_start <= '0';
        wait for clk_period;
        e_start <= '1';
        wait for clk_period;
        e_start <= '0';
        for i in 0 to 42 loop
            wait for clk_period;
        end loop;
        wait for clk_period;
        wait;
end process;
```

Figure 2: UUT used for Skinny-128-384+ Verification

6 CipherCore_Datapath_TB

As mentioned previously, we were able to verify that some of the functionality was working but not everything. This TestBench was designed in order to do for loops through multi-dimensional arrays or std_logic_vectors which consist of 32 bit words that would be fed in through bdi or key in order to load inputs manually into the circuit to begin attempted encryption of multiple blocks. To make this easier, we designed the testbenches such that they would have arrays of std_logic_vectors as seen in Figure.3. The amazing thing about this is that we can simply do a for loop through the length of the array in Figure.5 so that for an arbitrary length of even blocks the testbench can verify. We designed this testbench to be handled with the use of for loops so that for generic even sizes of message or associated data could be processed without having to repeat to much code. An example of loading Key and Nonce is seen in Figure.5. The logic for this also helped in attempted design of the main ASM chart. After processing all AD blocks, the TB would process the Message blocks and in every ρ operation it would load into the PISO the newest Ciphertext block. The TB did not necessarily take the 4 clock cycles to output this through bdo but we just wanted to see if the blocks themselves were accurate and unfortunately they were not. We think after a couple more days of debugging, it could have been fully verified but we are happy with the progress.

```
constant ADarray : test_array := ((x"00010203", x"04050607", x"08090A0B", x"0C0D0E0F"),
                                  (x"00010203", x"04050607", x"08090A0B", x"0C0D0E0F"));

constant Marray : test_array := ((x"00010203", x"04050607", x"08090A0B", x"0C0D0E0F"),
                                 (x"00010203", x"04050607", x"08090A0B", x"0C0D0E0F"));

constant KeyArray : input_array := (x"00010203", x"04050607", x"08090A0B", x"0C0D0E0F");

constant NonceArray : input_array := (x"00010203", x"04050607", x"08090A0B", x"0C0D0E0F");
```

Figure 3: Input Data Types

```
-- Loading Key
for a in 0 to 3 loop
    enKey <= '1';
    key <= KeyArray(a);
    wait for clk_period;
end loop;
enKey <= '0';
-- Loading Nonce
for b in 0 to 3 loop
    enN <= '1';
    bdi <= NonceArray(b);
    wait for clk_period;
end loop;
enN <= '0';
```

Figure 4: Loading Key and Nonce

```
-- Load and Process Associated Data Blocks
for c in 0 to ADarray'length-1 loop
                    (00010203,04050607,0809
```

Figure 5: Example of processing input blocks using length of array

7 Software Figures

```
for i in range(1,math.floor(a/2)+1):
    S, _ = rho(S, A_parsed[2*i-1])
    stringS = "".join("{:02X}".format(x) for x in S)
    print(f"RHO Output {stringS}")
    # print(f"Rho Call AD Block {2*i +1} ".join(str(stringS)))
    counter = increase_counter(counter)
    stringS = "".join("{:02X}".format(x) for x in tk_encoding(counter, 8, A_parsed[2*i], K))
    print(f"Counter Output {stringS}")
    # print("".join("{:02X}".format(_) for _ in tk_encoding(counter, 8, A_parsed[2*i], K)))
    S = skinny_enc(S, tk_encoding(counter, 8, A_parsed[2*i], K))
    stringS = "".join("{:02X}".format(_) for _ in S)
    print(f"TBC Output {stringS}")
    counter = increase_counter(counter)
```

Figure 6: Extension of Debugging of Round Functions in Skinny-128-384+

```
for i in range(NB_ROUNDS):
    for j in range(16): s[j] = S8[s[j]]

    if DEBUG==1: print(f"Internal State for Round {i} after SC = " + "".join("{:02X}".format(_) for _ in s))

    s[0] ^= c[i] & 0xf
    s[4] ^= (c[i]>>4) & 0xf
    s[8] ^= 0x2

    if DEBUG==1: print(f"Internal State for Round {i} after AC = " + "".join("{:02X}".format(_) for _ in s))

    for j in range(8): s[j] ^= tk[i][j] ^ tk[i][j+16] ^ tk[i][j+32]

    if DEBUG==1: print(f"Internal State for Round {i} after ART = " + "".join("{:02X}".format(_) for _ in s))

    s[4], s[5], s[6], s[7] = s[7], s[4], s[5], s[6]
    s[8], s[9], s[10], s[11] = s[10], s[11], s[8], s[9]
    s[12], s[13], s[14], s[15] = s[13], s[14], s[15], s[12]

    if DEBUG==1: print(f"Internal State for Round {i} after SR = " + "".join("{:02X}".format(_) for _ in s))

    for j in range(4): s[j], s[4+j], s[8+j], s[12+j] = s[j] ^ s[8+j] ^ s[12+j] , s[j], s[4+j] ^ s[8+j] , s[0+j] ^ s[8+j]

    if DEBUG==1: print(f"Internal State for Round {i} after MC = " + "".join("{:02X}".format(_) for _ in s))
```

Figure 7: Extension of Debugging of Romulus N encryption