This note explores some VHDL code for potential use by an adversary that implements an insertion of malicious circuitry by use of the design entry, with the purpose to activate a denial of service (DOS). DOS, in the context of FPGA security, is defined as an attempt to disable a target device’s functionality, eventually causing it to cease operation. The target device used here is a Xilinx Spartan-6 FPGA, embedded on a Digilent Atlys design board. The function under attack is the code for an implementation of the AES algorithm, which utilizes a matrix block cipher that manipulates, in most cases, the address of signature bits. It rotates the bits, in matrix, in circular fashion.

Due to the design of Digilent's board, a plain text equivalent is taken in from the switches available, a byte in length. When a button is pressed on the board, the switch configuration is stored to be later encrypted. As the store button is pressed, if the switch is in the up position, it is considered "1," in the down position it is considered "0." Figure 1 shows the VHDL code of how the switch array is stored in the plain text variable.

```
SIGNAL plain_text_IN, cipher_text, plain_text_OUT, led_config : std_logic_vector(7 DOWNTO 0);
-- plain_text_IN, a logic vector that takes the value of the INPUT switch_array
-- cipher_text, a logic vector that takes the value of the encrypted plain_text_IN
-- plain_text_OUT, a logic vector that takes the value of the decrypted cipher_text
-- led_config, a logic vector that stores a determined value, dependent on the INPUT event of button_array,
-- and sets the OUTPUT values of the led_array logic vector
switch_array: IN std_logic_vector(7 DOWNTO 0);
-- An array of constrained switches, denoted {switch_array(7), ... , switch_array(0)}, as an
-- INPUT signal
IF (button_array(0) = '1') THEN
    plain_text_IN <= switch_array;
    -- plain_text_IN updates to the value of the INPUT switch_array signal
    led_config <= plain_text_IN;
    -- led_config updates to the value of plain_text_IN
    operation <= "10";
END IF;
```

---

Figure 1. Store Plain Text Module.
Once another button is pressed on the board, informing the program to encrypt the plain text, it is sent to the encryption module. The logic of the encryption module takes an already predefined cipher key given in Figure 2.

```
VARIABLE char_cipher_key : char_word;
-- Key: 4815162342THISISJUSTATESTCIPHERKEYASPARTOFTHEFPGASECURITYREPORT4
```

**Figure 2. Cipher Text**

This cipher key, similar to the steps of the matrix block cipher as described by the AES algorithm, as the 64 character cipher key array is spilt up into 8 parts, shown in Figure 3.

```
char_cipher_key(0) := "48151623"; char_cipher_key(1) := "42THISISIS";
char_cipher_key(2) := "JUSTSTATES"; char_cipher_key(3) := "TCIPHERKER";
char_cipher_key(4) := "EYASPART"; char_cipher_key(5) := "OFTHEFPG";
char_cipher_key(6) := "ASECURITY"; char_cipher_key(7) := "YREPORT4";

bit_cipher_key := char_to_bit_cipher_key_gen(char_cipher_key);
```

**Figure 3. Cipher text broken into parts.**

Each character value is then translated to its ASCII byte binary value. Part of the algorithm is shown in Figure 4. As one may see, each character has an equivalent byte.

```
IF (char(1) = 'A') THEN byte_value := "01000001"; END IF;
IF (char(2) = 'B') THEN byte_value := "01000010"; END IF;
IF (char(3) = 'C') THEN byte_value := "01000011"; END IF;
IF (char(4) = 'D') THEN byte_value := "01000100"; END IF;
IF (char(5) = 'E') THEN byte_value := "01000101"; END IF;
IF (char(6) = 'F') THEN byte_value := "01000110"; END IF;
IF (char(7) = 'G') THEN byte_value := "01000111"; END IF;
IF (char(8) = 'H') THEN byte_value := "01001000"; END IF;
IF (char(9) = 'I') THEN byte_value := "01001001"; END IF;
IF (char(10) = 'J') THEN byte_value := "01001010"; END IF;
IF (char(11) = 'K') THEN byte_value := "01001011"; END IF;
IF (char(12) = 'L') THEN byte_value := "01001100"; END IF;
IF (char(13) = 'M') THEN byte_value := "01001101"; END IF;
IF (char(14) = 'N') THEN byte_value := "01001110"; END IF;
IF (char(15) = 'O') THEN byte_value := "01001111"; END IF;
IF (char(16) = 'P') THEN byte_value := "01010000"; END IF;
IF (char(17) = 'Q') THEN byte_value := "01010001"; END IF;
IF (char(18) = 'R') THEN byte_value := "01010010"; END IF;
IF (char(19) = 'S') THEN byte_value := "01010011"; END IF;
IF (char(20) = 'T') THEN byte_value := "01010100"; END IF;
IF (char(21) = 'U') THEN byte_value := "01010101"; END IF;
IF (char(22) = 'V') THEN byte_value := "01010110"; END IF;
IF (char(23) = 'W') THEN byte_value := "01010111"; END IF;
IF (char(24) = 'X') THEN byte_value := "01011000"; END IF;
IF (char(25) = 'Y') THEN byte_value := "01011001"; END IF;
IF (char(26) = 'Z') THEN byte_value := "01011010"; END IF;
```

**Figure 4. Part of ASCII byte binary values.**
Once the string of 64 characters is translated to its 512-bit binary equivalent, it then undergoes the encryption algorithm shown in Figure 5. This algorithm adds the corresponding bytes of the translated cipher key to the 8-bit plain text consecutively.

```vhdl
VARIABLE sum_cipher : std_logic_vector(0 TO 512);
sum_cipher := plain_text + byte[0];
sum_cipher := sum_cipher + byte[1];
sum_cipher := sum_cipher + byte[2];
sum_cipher := sum_cipher + byte[3];
sum_cipher := sum_cipher + byte[4];
...
sum_cipher := sum_cipher + byte[62];
sum_cipher := sum_cipher + byte[63];
```

Figure 5. VHDL Encryption Algorithm.

After the summation has been calculated, another button may be pushed to decrypt the now cipher text to plain text once again. As shown in Figure 6, this process relies on subtracting, in reverse order, the translated cipher key bytes from the cipher text. After the calculations have been performed, the plain text may then be shown once again.

```vhdl
VARIABLE sum_decipher : std_logic_vector(0 TO 512);
sum_decipher := cipher_text - byte[63];
sum_decipher := sum_decipher - byte[62];
sum_decipher := sum_decipher - byte[61];
sum_decipher := sum_decipher - byte[60];
sum_decipher := sum_decipher - byte[59];
...
sum_decipher := sum_decipher - byte[1];
sum_decipher := sum_decipher - byte[0];
```

Figure 6. VHDL Decryption Algorithm