Rolling CarKey :

State Machine Logic for Secure Car Access using Synchronized Rolling Codes

Nayla Pramesti Adhina
2406368901

Nadira Fayyaza Aisy
2406368933

Hanif Mulia Nugroho
2406368933

Automotive Security in Car

With current interconnected cars being transformed into a data center on wheels, automotive cybersecurity is no longer a matter of interest, but has turned into a matter of concern. The use of cloud integration and interaction with personal devices, which are becoming more complex each day makes vehicles extremely susceptible to attacks. Thus, wireless car states that the strategy of Security-by-design should be the basis of automotive cybersecurity infrastructure.

All connected car services should be heavily laced with security, as opposed to being viewed as an afterthought (can be referred as icing on the cake). **Our Rolling CarKey project aligns with this important principle by applying the Rolling Code algorithm in hardware level (VHDL)** with the objective of offering a high efficiency and secure key to car authentication module. This design was designed in direct response to the desperate demand of a strong, non-replayable security design in the modern car industry.

Introduction

Rolling CarKey: Enhancing Automotive Security

Brief Description

Rolling CarKey is a wireless authentication system utilizing a rolling code mechanism to secure the communication between the car key (Client) and the car (Server). This approach is crucial to prevent replay attacks, where an attacker records and reuses a past valid signal.

The system relies on three primary parameters that must be synchronized and correctly managed by both entities :

1. **Secret Key (K):** A shared secret key used for encrypting/decrypting the codes (K=11).
2. **Synchronization Counter (C):** A counter value that increments after every successful authentication (C=5).
3. **Identification (I):** A unique ID to identify the specific Key/Account (I=2).

System Overview

Components and Communication

Key (Client) → Car (Server)

Using SPI Communication (4 wires)

Key FSM

Key.vhd
Key_FSM.vhd
Key_Database.vhd

Car FSM

Car.vhd
Car_FSM.vhd
Car_Database.vhd

Objective and Tools

What's the purpose of Rolling CarKey?

1. To establish a rolling code authentication scheme based on the One-Time Password (OTP) that is calculated by using a counter and a secret key, which is effective in preventing replay attacks.
2. To establish reliable master-slave communication between the key and car unit using the Serial Peripheral Interface (SPI) protocol and To develop and integrate a synchronized counter system that increments with each transmission and validates counter freshness within a defined window on the receiver side.
3. To create modular VHDL components, including finite state machines (FSMs), an OTP generator, SPI controllers, and a secure database for key and counter storage.
4. To ensure the complete system is synthesizable and capable of real-time authentication, with clear outputs for door lock/unlock status and alarm activation.

Objective and Tools

What's the purpose of Rolling CarKey?

Tools that we use to create the technology:

ModelSim

Quartus Prime

VS Code

Github

Google Slides

Implementations

Rolling CarKey implements six modules.

Module 1: Dataflow Style

Dataflow Style is implemented through concurrent signal assignments that directly wire internal signals to outputs, such as SCK<=sck_int in the SPI master and Tx_Active<=spi_is_busy in the key unit. It also appears in combinational processes like the packet selector MUX, where outputs update immediately when inputs change, emphasizing parallel signal propagation over sequential execution.

Module 2: Behavioral Style

In this project, behavioral styles were mainly implemented through process statements with sensitivity lists that describe sequentially and are driven by clock behavior. It's used extensively in all finite state machines, such as key_fsm, car_fsm for state transitions, in modules like counter_inc and spi_master for sequential counting and shifting, and in the otp generator for algorithmic computation. This enables a procedural description of functionality using if-then-else, case, and synchronous updates on clock edges, and focuses on what the system does rather than its structural wiring.

Module 3: Testbench

Testbench, in this project, was handled in a single VHD. file. Our testbench uses structural instantiation of both key and car units, creates a clock process with controlled timing, and implements stimulus procedures such as press_unlock_button, lock_car to simulate user interactions. It monitors outputs such as car_door, car_siren, and key_tx_active to track authentication success through unlock_count.

Module 4: Testbench

The design uses a hybrid approach: behavioral code for packet selection, signal assignment, control logic, and constants, combined with structural programming through component instantiation for flexibility and easier debugging. In key.vhd, six components: button_debouncer, counter_inc, otp_generator, key_fsm, SPI_master, plus internal signals are interconnected via port maps, while car.vhd similarly instantiates SPI_slave, Car_Database, OTP_Generator, and Car_FSM. The testbench also applies structural programming by instantiating both top-level entities and directly linking their SPI ports to simulate full system communication. Arrays are used in the Car_Database and Key_Database to efficiently store and retrieve authentication data.

Implementations

Rolling CarKey implements six modules.

Module 5: Looping Construct

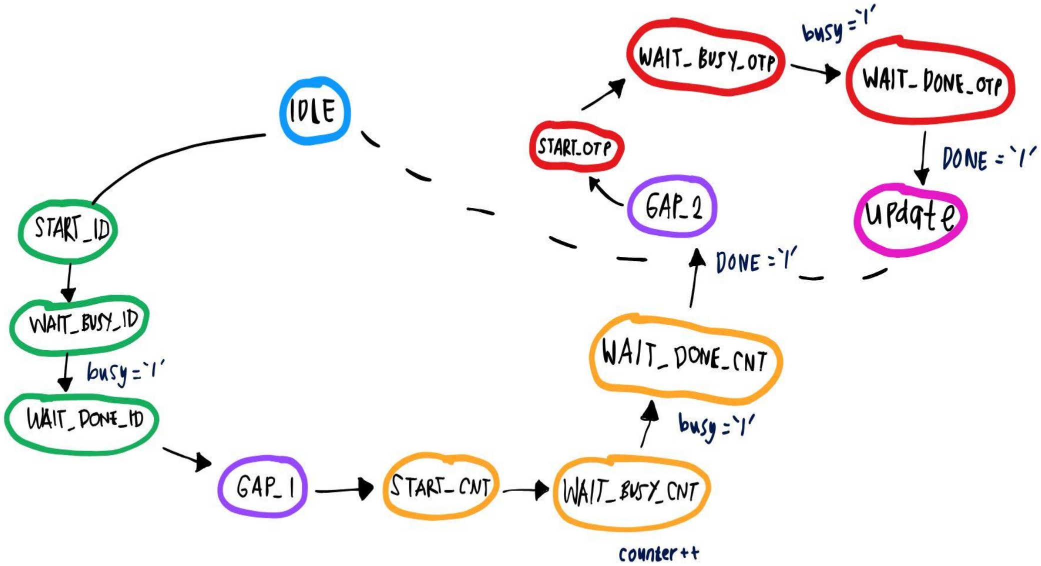
This module is implemented in the Car_Database using a for loop to perform a sequential search operation through the database array. It iterates through all 4 database entries (0 to 3 for loops) to execute two functions, the first one being to search for a matching user ID during the authentication lookup, and the second is to update the corresponding counter value after successful verification. The choice made for a for-loop over something like while or for-generate is due to a fixed number of iterations aligning with the known size of the database, ensuring it was gonna be able to be synthesized and readable.

Module 6: Finite State Machine

The core of this project's control logic lies in two behaviorally implemented Finite State Machines (FSMs): the Key_FSM (Transmitter) and the Car_FSM (Receiver). The Key_FSM (13 states) manages the sequential transmission of three authentication packets (User ID, Counter, and OTP) via SPI; the FSM starts in IDLE, proceeds through packet transmission and SPI synchronization (WAIT_DONE), includes the crucial WAIT_BUSY_CNT state for incrementing the counter value, and concludes in the UPDATE state to permanently record the new counter if the transaction is successful, before returning to IDLE. The Car_FSM (5 states), conversely, is responsible for managing the car's response to authentication attempts; upon receiving the initial trigger, the FSM enters CHECK to validate all three packets, looking up the User ID, calculating the Expected OTP, and comparing the two OTP values. The success path (Code_Is_Correct = '1') leads to UPDATE (incrementing the counter in the database), then to OPEN (setting Door_Open = '1'), and back to IDLE; whereas the failure path (Code_Is_Correct = '0') triggers ALARM (activating Alarm_Siren = '1'), followed by ALARM_WAIT (enforcing a lockout timer), before finally returning to IDLE.

FSM Diagram

Finite State Machine: Key Controller (Client FSM)



Our Command Sequence for Key FSM,

User presses the Key

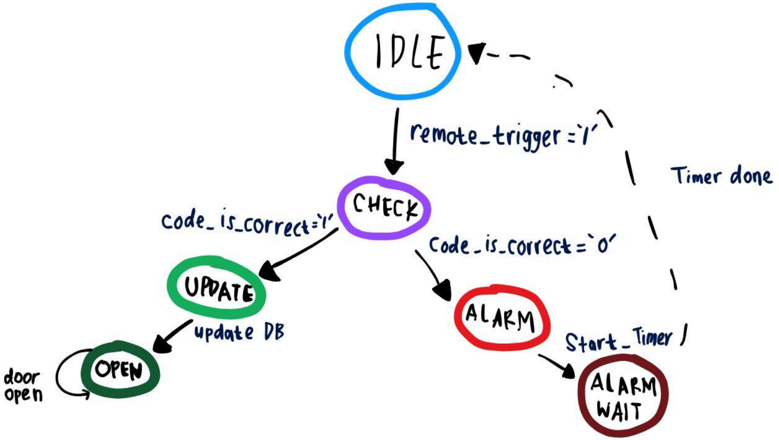
It starts in IDLE and transitions to **START_ID** when the Button_Press is received. The FSM subsequently transmits the User_ID, the Counter followed by the OTP (Rolling Code) and states such as **WAIT_DONE_ID**, **WAIT_DONE_CNT** and **WAIT_DONE_OTP** are used to synchronize the SPI after a packet.

Importantly, the **WAIT_BUSY_CNT** state provides the obligatory operation of the increased value of the counter state, the **Counter++**, prepping system for the next transaction. Last, the UPDATE state indicates success of the transaction with the car and it'll permanently records counter in Key_DB if successful, & goes back to IDLE.

Complete transmitting!

FSM Diagram

Server FSM: Security Validation and Alarm Logic



Our Command Sequence for Car FSM,

IDLE transitions to **CHECK** upon receiving the **initial trigger**. In **CHECK**, the FSM quickly performs the critical validation: **receiving all three packets** (ID, Counter, OTP) via SPI, **looking up the User ID**, calculating the **Expected OTP**, and **comparing the two OTP values.**

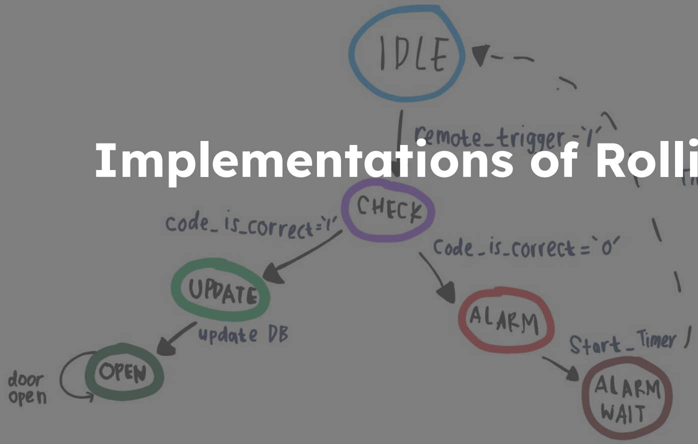
Success Path (**Code_Is_Correct = '1'**): The FSM passes to UPDATE (permanently incrementing the Counter C in the database), and then to OPEN (setting **Door_Open = '1'**), and again to IDLE.

Failure Path (**Code_Is_Correct = '0'**): When ALARM (activating Alarm_Siren = '1') is entered, ALARM_WAIT (forcing a lockout timer to prevent brute-force attacks) is enforced, then at last the FSM is returned to IDLE.

System protects the Car!

FSM Diagram

Server FSM: Security Validation and Alarm Logic



Implementations of Rolling CarKey, on next slide!

Our Command Sequence for Car FSM,

IDLE transitions to **CHECK** upon receiving the **initial trigger**. In **CHECK**, the FSM quickly performs the critical validation: **receiving all three packets** (ID, Counter, OTP) via SPI, **looking up the User ID**, calculating the **Expected OTP**, and **comparing the two OTP values.**

Success Path (**Code_Is_Correct = '1'**): The FSM passes to UPDATE (permanently incrementing the Counter C in the database), and then to OPEN (setting **Door_Open = '1'**), and again to IDLE.

Failure Path (**Code_Is_Correct = '0'**): When ALARM (activating Alarm_Siren = '1') is entered, ALARM_WAIT (forcing a lockout timer to prevent brute-force attacks) is enforced, then at last the FSM is returned to IDLE.

System protects the Car!

Codes

Important codes for the Rolling CarKey system: Key Database

Key_Database.vhd

```vhdl
ibrary IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Key_Database is
    Port (
            clk           : in  STD_LOGIC;                        -- Synchronous's read clock
            User_ID       : in  STD_LOGIC_VECTOR (1 downto 0);   -- User ID (00-11)
            Secret_Key_Out : out STD_LOGIC_VECTOR (31 downto 0) -- Secret key output
            );
end Key_Database;

architecture Behavioral of Key_Database is
    -- Array of 4 secret keys (ROM)
    type key_rom_type is array (0 to 3) of std_logic_vector(31 downto 0);

    constant KEY_ROM : key_rom_type := (
        0 => x"11111111",
        1 => x"AABBCCDD",
        2 => x"12345678",
        3 => x"FEDCBA98"
    );
begin
    process(clk)
    begin
        if rising_edge(clk) then
            -- Lookup key according to the User ID
            Secret_Key_Out <= KEY_ROM(to_integer(unsigned(User_ID)));
        end if;
    end process;
end Behavioral;
```

Key Database

Key_Database is a storage mechanism for secret keys that's associated with specific user IDs. This code implements a simple synchronous read from a ROM (Read-Only Memory) structure when the clock edge is detected. It takes a User_ID as inputs, and Secret_Key_Out is the outputs.

Core of this database is **KEY_ROM**, which is an array of 4 secret keys that maps an index that has been derived from input, to a specific 64-bit secret key.

Codes

Important codes for the Rolling CarKey system : Car Database

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Car_Database is
    Port (
        clk : in std_logic;
        rst : in std_logic;

        -- Lookup interface to read credentials
        lookup_id : in std_logic_vector(7 downto 0);
        lookup_en : in std_logic;

        -- Lookup results
        key_found : out std_logic;
        stored_key : out std_logic_vector(31 downto 0);
        stored_counter : out std_logic_vector(7 downto 0);

        -- Counter update interface that used when authentication success
        update_en : in std_logic;
        update_id : in std_logic_vector(7 downto 0);
        update_counter : in std_logic_vector(7 downto 0)
    );
end Car_Database;

architecture Behavioral of Car_Database is
  -- Database Structure
    type key_record is record
        id      : std_logic_vector(7 downto 0);
        key     : std_logic_vector(31 downto 0);
        counter : std_logic_vector(7 downto 0);
        valid   : std_logic;
    end record;

    type db_array is array(0 to 3) of key_record;
    signal database : db_array := (
        0 => (id => x"02", key => x"0000000B", counter => x"00", valid => '1'), -- Key #2
        1 => (id => x"03", key => x"0000000F", counter => x"00", valid => '1'), -- Key #3
        2 => (id => x"04", key => x"00000014", counter => x"00", valid => '1'), -- Key #4
        3 => (id => x"FF", key => x"00000000", counter => x"00", valid => '0')  -- Empty slot
    );
```

Car Database

Car_Database is for the secure memory module within the Car (Server) for the Rolling CarKey system. Implemented using behavioral architecture, this component manages the credentials and synchronization data for multiple keys. It holds a fixed-size array (db_array) or key_record structs where has the function of records a unique **ID (8 bits), secret Key (32 bits)**, synchronization **Counter (8 bits)** and a valid **flag.**

Two Operations in this module:

1. **Lookup :** when lookup_en signal is high, then the component iterates through the internal database to find ID that matches the input from lookup_id. If it matches and valid, key_found is asserted high and the associated stored_key (K) and stored_counter © are outputted for use in OTP calculation and comparison.
2. **Counter Update** : when update_en signal is high, the module iterates to find the entry matching update_id. If matches, database permanently updates that key's counter value with new update_counter.

Codes

Important codes for the Rolling CarKey system

Car.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Car is
    Port (
            clk             : in  STD_LOGIC;
            reset           : in  STD_LOGIC;
            SPI_SCLK        : in  STD_LOGIC;
            SPI_MOSI        : in  STD_LOGIC;
            SPI_CS_N        : in  STD_LOGIC;
            SPI_MISO        : out STD_LOGIC;
            Door_Lock       : out STD_LOGIC;
            Siren           : out STD_LOGIC
            );
end Car;

architecture Behavioral of Car is

    component SPI_Slave
    port (
        CLK_IN      : in  std_logic;
        RESET_N     : in  std_logic;
        SPI_SCLK    : in  std_logic;
        SPI_MOSI    : in  std_logic;
        SPI_CS_N    : in  std_logic;
        DATA_IN     : in  std_logic_vector(31 downto 0);
        DATA_OUT    : out std_logic_vector(31 downto 0);
        NEW_DATA    : out std_logic;
        SPI_MISO_OUT: out std_logic
    );
    end component;

    component Car_Database
    Port (
        clk : in std_logic;
        rst : in std_logic;
        lookup_id : in std_logic_vector(7 downto 0);
        lookup_en : in std_logic;
        key_found : out std_logic;
        stored_key : out std_logic_vector(31 downto 0);
        stored_counter : out std_logic_vector(7 downto 0);
        update_en : in std_logic;
        update_id : in std_logic_vector(7 downto 0);
        update_counter : in std_logic_vector(7 downto 0)
```

Car

Car represents the car side controller of a secure rolling-code key system. This registers three packet data of the SPI of the remote key, which are user ID, counter number, and OTP code, in internal registers. The system then checks the secret key of the key and the stored counter with the received user ID by querying the Car Database on whether the key is present and then retrieving the corresponding secret key. Using this information, the OTP Generator estimates the expected OTP by adding the received counter and the secret key that is stored.

In order to ensure synchronization and to avoid replay attacks, the received counter is checked within a small window of the stored counter. The system next cross compares the received OTP and the expected OTP and ensures that the counter is validated and the key registered. In case of a successful authentication, the Car_FSM opens the door, changes the counter stored and leaves the alarm off. When validation is not successful the FSM is used to activate the siren. In general, this module incorporates data reception, database search, OTP reconstruction, counter coordination, and security management that provides the safety of secure and reliable access to cars.

Codes
Important codes for the Rolling CarKey system

Key.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Key is
    Port (
            clk         : in  STD_LOGIC;
            reset       : in  STD_LOGIC;
            Button      : in  STD_LOGIC;
            SPI_SCK     : out STD_LOGIC;
            SPI_MOSI    : out STD_LOGIC;
            SPI_SS      : out STD_LOGIC;
            Tx_Active   : out STD_LOGIC
            );
end Key;

architecture Hybrid of Key is

--component declarations (structural part):
    --button debouncer comp

    component Button_Debouncer is
        Port(
            clk     : in std_logic;
            btn_in  : in std_logic;
            btn_out : out std_logic
        );
    end component;

    --counter increment comp
    component Counter_Inc is
        Port (
            clk      : in  STD_LOGIC;
            reset    : in  STD_LOGIC;
            enable   : in  STD_LOGIC;
            inc      : in  STD_LOGIC; --increment pulse
            load     : in  STD_LOGIC; --load new value
            data_in  : in  STD_LOGIC_VECTOR(31 downto 0);
            count    : out STD_LOGIC_VECTOR(31 downto 0);
            overflow : out STD_LOGIC  --overflow indicator
        );
    end component;
```

Key

Key module serves as the remote transmitter of the rolling-code system. Upon pressing the button, the debouncer cleans the signal, and the Key_FSM begins the sending process. The Counter_Inc updated the rolling counter and the OTP Generator generated OTP with this counter and the secret key of the key. The FSM determines the packet to be transmitted (user ID, counter, or OTP), and the selected information is fed to the SPI_Master, which takes care of the actual SPI transmission. In this process, the SPI busy signal signals when transmission is in operation. The general functionality of the module is to create the counter and the OTP and transmit all the three packets to the car each time the button is pressed.

Codes

Important codes for the Rolling CarKey system

otp_gen.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity OTP_Generator is
    Port (
            Counter    : in  STD_LOGIC_VECTOR (31 downto 0);  --counter value
            Secret_Key : in  STD_LOGIC_VECTOR (31 downto 0);  --secret key
            OTP_Result : out STD_LOGIC_VECTOR (31 downto 0)   --generated otp
            );
end OTP_Generator;

architecture Behavioral of OTP_Generator is
begin
    process(Counter, Secret_Key)
        variable temp : unsigned(31 downto 0);
    begin
        --xor counter with secret_Key (first mixing)
        temp := unsigned(Counter) xor unsigned(Secret_Key);

        --non-linear diffusion (add rotated version of itself)
        --rotate left 3 bits
        temp := temp + (temp rol 3);

        --golden ratio constant (to break patterns)
        temp := temp + x"9E37";

        --rotate left 7 bits
        temp := temp rol 7;

        --result
        OTP_Result <= std_logic_vector(temp);
    end process;

end Behavioral;
```

OTP_Generator

OTP Generator is used in the Rolling CarKey system which has two inputs Counter and Secret_Key (fixed shared secret key between car and key). This OTP is generated through several operations like XOR mixing in which the counter is XORed with the secret key for initial randomness, and then a Non-Linear Diffusion in which the intermediate value is rotated left by 3 bits and added back to the original.

The ratio of the constant is Golden Ratio Constant is added to avoid repetitive output patterns, and lastly the value will be rotated left by 7 bit for additional scrambling. The output will be 32-bits will be converted back to std_logic_vector and assigned as the OTP.

Codes

Important codes for the Rolling CarKey system

otp_gen.vhd

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity OTP_Generator is
    Port (
            Counter    : in  STD_LOGIC_VECTOR (31 downto 0);  --counter value
            Secret_Key : in  STD_LOGIC_VECTOR (31 downto 0);  --secret key
            OTP_Result : out STD_LOGIC_VECTOR (31 downto 0)   --generated otp
            );
end OTP_Generator;

architecture Behavioral of OTP_Generator is
begin
    process(Counter, Secret_Key)
        variable temp : unsigned(31 downto 0);
    begin
        --xor counter with secret_Key (first mixing)
        temp := unsigned(Counter) xor unsigned(Secret_Key);

        --non-linear diffusion (add rotated version of itself)
        --rotate left 3 bits
        temp := temp + (temp rol 3);

        --golden ratio constant (to break patterns)
        temp := temp + x"9E37";

        --rotate left 7 bits
        temp := temp rol 7;

        --result
        OTP_Result <= std_logic_vector(temp);
    end process;

end Behavioral;
```

For all implementation, how does the code works?

OTP_Generator

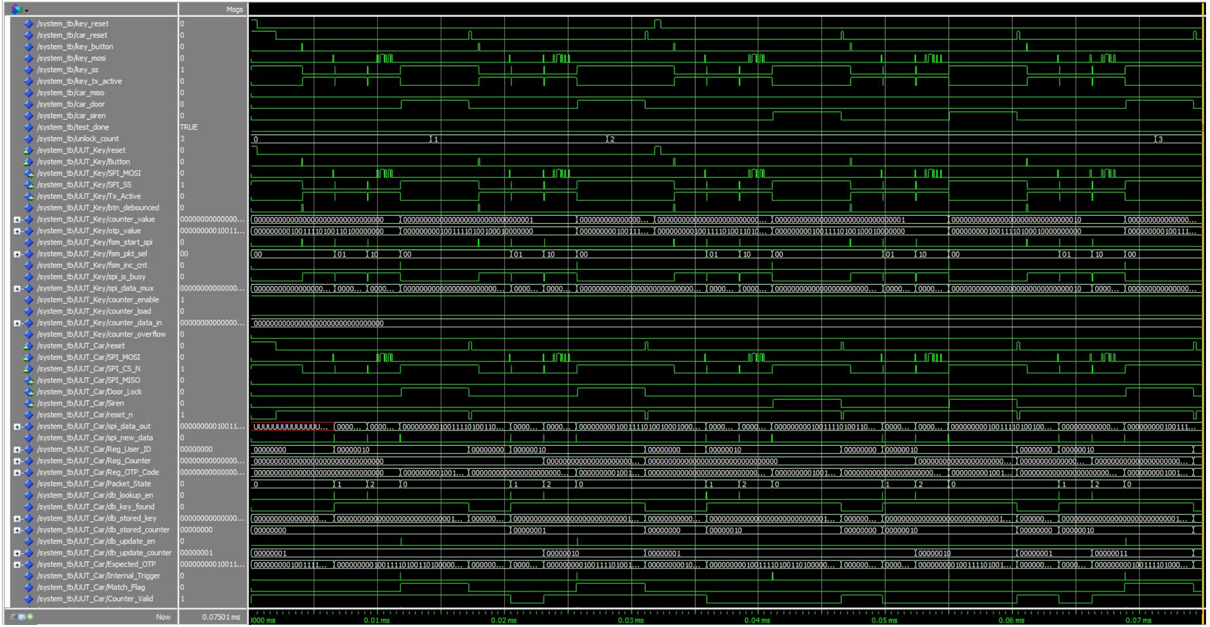
OTP Generator is used in the Rolling CarKey system which has two ... fixed shared secret key between car ... through several operations like ... is XORed with the secret key for initial randomness, and then a Non-Linear Diffusion in which the intermediate value is rotated left by 3 bits and added back to the original.

The ratio of the constant is Golden Ratio Constant is added to avoid repetitive output patterns, and lastly the value will be rotated left by 7 bit for additional scrambling. The output will be 32-bits will be converted back to std_logic_vector and assigned as the OTP.

Test Result

Rolling CarKey system is checked, and effective!

Synthesize of Rolling CarKey

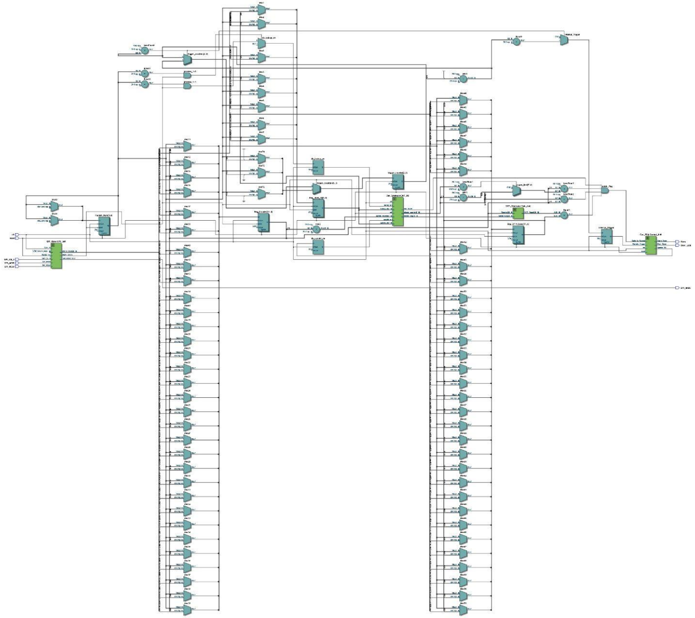


Test Result

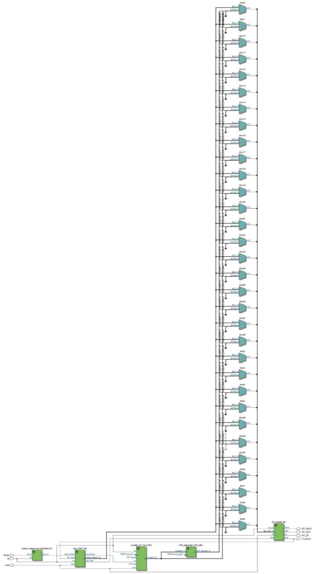
Rolling CarKey system is checked, and effective!

Project Schematic of Rolling CarKey

Car



Key



Thank you!