

# CAN Network Security Protocol of Autonomous Vehicle based on FPGA in the Loop

by

FARAG MOHAMED E. LAGNF

A dissertation submitted in partial fulfillment of the requirements  
for the degree of

DOCTOR OF PHILOSOPHY IN ELECTRICAL AND COMPUTER ENGINEERING

2026

Oakland University  
Rochester, Michigan

Doctoral Advisory Committee:

Subra Ganesan, Ph.D.

Anyi Liu, Ph.D.

Shadi G. Alawneh, Ph.D.

Xiaodong Deng, Ph.D.

© Copyright by Farag Mohamed E Lagnf, 2026  
All rights reserved

الحمد لله ما انتهى جهدي، ولا تم فضلي، ولا وصلنا درب إلا برحمته وكرمه، فالحمد لله على بلوغ النجاح والفلاح.

This dissertation is dedicated to the memory of my mother, who passed away in 2021, and to my father, who has been a constant source of love and support throughout my life. Their unwavering belief in me and their encouragement have been instrumental in shaping my academic journey. I am also deeply grateful to my wife, who has been my rock throughout this challenging endeavor. Her unwavering love and support, along with her understanding of the sacrifices required for this pursuit, have made this accomplishment possible.

## ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Dr. Subra Ganesan, for their unwavering support and guidance throughout my PhD journey. Their insightful feedback and encouragement were instrumental in shaping this thesis. I am also grateful to my committee members, Dr. Shadi G. Alawneh, Dr. Xiaodong Deng, and Dr. Anyi Liu, for their invaluable time, insightful questions, and constructive criticism. My research was made possible by the generous financial support of Libyan Education Ministry, for which I am profoundly grateful. I also extend my thanks to my colleagues at Hary for their collaboration and support. This work is dedicated to the memory of my mother, who sadly passed away in 2021 due to COVID-19. Her unwavering love and belief in my abilities were a constant source of strength during my early years of study, and her absence continues to be deeply felt. Finally, I am eternally grateful to my family and friends for their unwavering love and encouragement. Their belief in me has been a constant source of strength throughout this challenging journey.

FARAG MOHAMED E. LAGNF

## ABSTRACT

### CAN Network Security Protocol of Autonomous Vehicle based on FPGA in the Loop

by

FARAG MOHAMED E. LAGNF

Adviser: SUBRA GANESAN, Ph.D.

In Software-Defined Vehicles (SDVs) the transition towards zonal topologies decreases the attack surface and improves the security requirements of in-vehicle communication networks such as CAN FD. This dissertation introduces a lightweight FPGA-based security frame designed to mitigate replay attacks and spoofing threats with a deterministic synchronized Freshness Value (FV) derived from a non linear function. The FV is transformed into a cryptographically balanced bitstream and integrated with AES 128 Encryption and lightweight authentication methods to make sure secrecy, integrity, and freshness verification with low computing burden.

The proposed architecture was executed and verified by an FPGA-in-the-Loop (FIL) codesign approach on a Xilinx Arty A7 platform. MATLAB and VHDL integration have been utilized. With loss of 30 % of CAN FD messages, experimental findings indicated

that precise synchronization has achieved between transmitter (Tx) and receiver (Rx) at 100 MHz, with observed latencies of 170 ns for Tx and 160 ns for Rx, resulting in an overall throughput of roughly 123 Mbps. encryption, ensures freshness and reduces replay attacks, thereby overcoming the limitations of conventional freshness mechanisms.

In order to accomplish this, we suggest a hardware architecture that is predicated on an FPGA (Arty 7A), which has been verified by our simulation results to provide high throughput and minimal payload. The real-time hardware implementation effectively limits replay attacks and protects data integrity and authenticity.

Furthermore, the message counter and sequences with SHA-512 are integrated as part of the improved security method for CAN XL. Although the default security measures of CAN XL are still in the process of evolving, our method guarantees the freshness of messages by utilizing a cross-correlation function to maintain them within a specified time frame, thereby preventing replay attacks. This necessitated modifications to FPGA-based architecture, which was validated using MATLAB and VHDL. This demonstrated its superiority over CAN FD's security measures in effectively mitigating replay and denial-of-service attacks.

As contributions of this work introduces a non-linear sequence function to increase the unpredictability of freshness values, thereby preventing adversaries from predicting communication sequences or deducing message patterns. This method eliminates the

necessity of transmitting freshness messages at predetermined intervals and enables multiple nodes to maintain unique freshness values, thereby further reducing vulnerabilities.

To further reduce resource utilization, a lightweight XOR-based authentication tag was designed and integrated. This 1-byte tag combines the FV, one ciphertext byte, and a key-schedule byte to provide an integrity indicator with minimal hardware overhead. Despite its small size, the tag maintained high randomness (entropy  $\approx 7.26$  bits) and demonstrated 100% replay resistance during simulation, confirming that the underlying FV entropy ensures message integrity and uniqueness even under constrained hardware conditions.

Our FPGA-in-the-loop (FIL) validation guarantees the real-time testing of cryptographic protocols in a hardware-accelerated environment, thereby facilitating the practical deployment of the security architecture for real-time embedded systems. In order to verify the protocol's robustness, we implement replay attacks that illustrate effective threat mitigation and protection.

## TABLE OF CONTENTS

|                                                                                           |      |
|-------------------------------------------------------------------------------------------|------|
| ACKNOWLEDGMENTS                                                                           | iv   |
| ABSTRACT                                                                                  | v    |
| TABLE OF CONTENTS                                                                         | ix   |
| LIST OF TABLES                                                                            | xiii |
| LIST OF FIGURES                                                                           | xiv  |
| CHAPTER ONE                                                                               |      |
| INTRODUCTION                                                                              | 1    |
| 1.1 Controller Area Network (CAN)                                                         | 2    |
| 1.2 Literature Review                                                                     | 7    |
| 1.3 Research Problem                                                                      | 15   |
| 1.4 Research Design / Methodology                                                         | 18   |
| 1.5 Research Objectives                                                                   | 19   |
| 1.5.1 Design FPGA-Based Security Protocols for CAN FD and<br>CAN XL Communication Systems | 19   |
| 1.5.2 Validate the Proposed Protocol Through MATLAB<br>Simulation                         | 20   |
| 1.5.3 Evaluate the Proposed Protocol Under Replay Attack<br>Scenarios                     | 20   |
| 1.5.4 Experimentally Evaluate the Proposed Protocol                                       | 20   |
| 1.6 Thesis Outline                                                                        | 22   |
| CHAPTER TWO                                                                               |      |
| AUTOMOTIVE EMBEDDED SECURITY BACKGROUND                                                   | 24   |

## TABLE OF CONTENTS—Continued

|               |                                                            |    |
|---------------|------------------------------------------------------------|----|
| 2.1           | Hardware Security in the Automotive Industry               | 24 |
| 2.2           | Embedded System Types                                      | 24 |
| 2.3           | Encryption Algorithms and Hash Functions in CAN Networks   | 28 |
| 2.3.1         | AES-128                                                    | 31 |
| 2.3.2         | Message Authentication Codes (MACs)                        | 37 |
| 2.3.3         | SHA-256                                                    | 38 |
| 2.3.4         | SHA-512                                                    | 40 |
| 2.4           | FPGA-in-the-Loop (FIL) Methodology Using MATLAB            | 41 |
| CHAPTER THREE |                                                            |    |
|               | FPGA-BASED SECURITY ARCHITECTURES FOR CAN FD AND CAN XL    | 44 |
| 3.1           | Overview of the Proposed FPGA-Based Security Architectures | 44 |
| 3.2           | Design of FPGA-Based Security Measures for CAN FD          | 44 |
| 3.2.1         | Proposed Hardware Design                                   | 44 |
| 3.2.2         | CAN FD Transmitter                                         | 46 |
| 3.2.3         | CAN FD Receiver                                            | 46 |
| 3.2.4         | Simulation Results                                         | 46 |
| 3.2.5         | Performance Analysis                                       | 52 |
| 3.2.6         | Conclusion                                                 | 53 |
| 3.3           | Design of FPGA-Based Security Measures for CAN XL          | 54 |
| 3.3.1         | Proposed CAN XL Hardware Design                            | 54 |
| 3.3.2         | CAN XL Transmitter                                         | 55 |

## TABLE OF CONTENTS—Continued

|                                                                                                                                                                       |                                                             |    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|----|
| 3.3.3                                                                                                                                                                 | CAN XL Receiver                                             | 59 |
| 3.3.4                                                                                                                                                                 | Simulation and Verification Results                         | 60 |
| 3.3.5                                                                                                                                                                 | Performance Analysis                                        | 66 |
| 3.3.6                                                                                                                                                                 | Conclusion                                                  | 67 |
| CHAPTER FOUR                                                                                                                                                          |                                                             |    |
| DESIGN EVOLUTION AND VALIDATION OF A NON-LINEAR FRESHNESS<br>VALUE PROTOCOL: FROM SHA-256 ON VIRTEX-6 TO A LIGHTWEIGHT FPGA-<br>IN-THE-LOOP IMPLEMENTATION ON ARTY A7 |                                                             | 69 |
| 4.1                                                                                                                                                                   | Threat Model                                                | 69 |
| 4.2                                                                                                                                                                   | Protocol Security Design: An Evolutionary Approach          | 70 |
| 4.2.1                                                                                                                                                                 | Initial Protocol Architecture: SHA-256-Based Authentication | 71 |
| 4.2.2                                                                                                                                                                 | Initialize the synchronization sequence                     | 72 |
| 4.2.3                                                                                                                                                                 | Message Generation                                          | 72 |
| 4.3.4                                                                                                                                                                 | Message Encryption                                          | 72 |
| 4.3.5                                                                                                                                                                 | Compute Hash                                                | 73 |
| 4.3.6                                                                                                                                                                 | Sync Sequence Management                                    | 73 |
| 4.3.7                                                                                                                                                                 | Transmit packet                                             | 73 |
| 4.3.8                                                                                                                                                                 | Await acknowledgement                                       | 73 |
| 4.3                                                                                                                                                                   | Proposed Protocol Architecture Overview                     | 78 |
| 4.3.1                                                                                                                                                                 | Non-linear Freshness Value Generation                       | 78 |
| 4.3.2                                                                                                                                                                 | Chaotic Freshness Value Enhancement                         | 79 |

## TABLE OF CONTENTS—Continued

|                            |                                                                             |     |
|----------------------------|-----------------------------------------------------------------------------|-----|
| 4.3.3                      | Lightweight Authentication Tag                                              | 82  |
| 4.3.4                      | Protocol Operation                                                          | 83  |
| 4.4                        | Freshness Value (FV) Transmitter and Receiver Validation                    | 83  |
| 4.4.1                      | Generation and Transmission of Freshness Value                              | 83  |
| 4.4.2                      | Receiver Validation Protocol                                                | 86  |
| 4.4.3                      | Transmitter Validation Protocol                                             | 86  |
| 4.5                        | FPGA Implementation Setup                                                   | 86  |
| 4.6                        | Protocol Analysis and Validation                                            | 87  |
| 4.6.1                      | Implementation of the Security Protocol at the Transmitter and the Receiver | 87  |
| 4.6.2                      | FPGA-in-the-Loop Implementation of the Replay Attack                        | 98  |
| 4.7                        | Optimized FPGA Implementation with Reduced I/O and Lightweight              | 103 |
| 4.7.1                      | Transmitter/Receiver FPGA-in-the-Loop (FIL) Implementation                  | 112 |
| 4.7.2                      | Statistical Security Evaluation of the Non-Linear Freshness Value Protocol  | 123 |
| 4.7.3                      | Results and Performance Evaluation                                          | 127 |
| CHAPTER FIVE               |                                                                             |     |
| DISCUSSION AND CONCLUSIONS |                                                                             | 130 |
| 5.1                        | Discussion                                                                  | 130 |
| 5.2                        | Conclusion                                                                  | 131 |

## TABLE OF CONTENTS—Continued

|                                                             |     |
|-------------------------------------------------------------|-----|
| CHAPTER SIX                                                 |     |
| FUTURE WORK                                                 | 133 |
| 6.1 Integration of Machine Learning for Intrusion Detection | 133 |
| 6.2 Expansion to Multi-Node and Zonal Architectures         | 133 |
| REFERENCES                                                  | 134 |

## LIST OF TABLES

|          |                                                                          |     |
|----------|--------------------------------------------------------------------------|-----|
| Table 1  | Validation of AES 128 Encryption and Decryption with SHA256              | 88  |
| Table 2  | Validation of 6 Messages with No Resend Attempts                         | 91  |
| Table 3  | Validation of Certain Resend Attempts and all Resend Attempts            | 94  |
| Table 4  | Validation of mixed Resend Attempts with Hash Flags                      | 97  |
| Table 5  | Corruption of Encrypted Data and Hash Value Validation                   | 98  |
| Table 6  | Mechanism of lightweight Hashing each Transmission                       | 108 |
| Table 7  | Validation of AES 128 Encryption and Decryption with Lightweight Hashing | 110 |
| Table 8  | Three CAN FD Frames with Correct FVs                                     | 115 |
| Table 9  | Rx Results of All -Success Case                                          | 115 |
| Table 10 | 6 CAN FD Frames each message sent with wrong and Correct FVs             | 116 |
| Table 11 | Rx Results of CAN FD Frames sent with wrong and Correct FVs              | 117 |
| Table 12 | Five CAN FD Frames of mixed Correct or Wrong FVs                         | 117 |
| Table 13 | Rx Results of CAN FD Frames sent with mixed Correct or Wrong FVs         | 117 |
| Table 14 | Statical Analysis                                                        | 124 |

## LIST OF FIGURES

|           |                                              |    |
|-----------|----------------------------------------------|----|
| Figure 1  | CAN and CAN FD Architecture                  | 4  |
| Figure 2  | CAN XL Architecture                          | 5  |
| Figure 3  | AUTOSAR Replay Attack Counter Measures       | 8  |
| Figure 4  | Freshness Value Structure                    | 8  |
| Figure 5  | Freshness Value Behaviour                    | 10 |
| Figure 6  | Freshness Value Reconstruction               | 10 |
| Figure 7  | Synchronization Message                      | 10 |
| Figure 8  | Secured I-PDU Including Truncated FV and MAC | 11 |
| Figure 9  | Synchronization Request                      | 11 |
| Figure 10 | Counter Approach Replay Attack               | 12 |
| Figure 11 | Freshness Values Based on Time Stamps        | 13 |
| Figure 12 | Freshness Values Based on Counters           | 14 |
| Figure 13 | The basic AES-128 cryptographic architecture | 31 |
| Figure 14 | AES 128 Sub-Bytes                            | 33 |
| Figure 15 | AES 128 Shift Rows                           | 33 |
| Figure 16 | AES 128 Mix Column                           | 34 |
| Figure 17 | AES 128 Add Round                            | 34 |
| Figure 18 | AES 128 Decryption- Inverse Shift Row        | 35 |
| Figure 19 | AES 128 Decryption- Inverse Sub Bytes        | 36 |

## LIST OF FIGURES—Continued

|           |                                                      |    |
|-----------|------------------------------------------------------|----|
| Figure 20 | AES 128 Decryption- Inverse Mix Column               | 36 |
| Figure 21 | SHA-256 Architecture                                 | 39 |
| Figure 22 | SHA-512 Architecture                                 | 40 |
| Figure 23 | Proposed FPGA-Based Security Architecture for CAN FD | 45 |
| Figure 24 | Proposed CAN FD Transmitter Architecture             | 45 |
| Figure 25 | Proposed CAN FD Receiver Architecture                | 45 |
| Figure 26 | AES 128 encryption core RTL                          | 47 |
| Figure 27 | AES128 encryption verification                       | 48 |
| Figure 28 | AES 128 Decryption RTL                               | 48 |
| Figure 29 | AES 128 Decryption Verification                      | 49 |
| Figure 30 | SHA 256 RTL                                          | 49 |
| Figure 31 | Proposed CAN FD Sender RTL                           | 50 |
| Figure 32 | Proposed CAN FD Sender Verification                  | 51 |
| Figure 33 | CAN FD Receiver RTL                                  | 51 |
| Figure 34 | CAN FD Receiver Verification                         | 52 |
| Figure 35 | CAN XL Security Measures                             | 54 |
| Figure 36 | Non-Linear Function with different Frequencies       | 56 |
| Figure 37 | Non-Linear Function with high and Low Frequencies    | 57 |
| Figure 38 | Non-Linear Functions with other Frequencies          | 57 |
| Figure 39 | Two Identical Linear Functions and their Difference  | 58 |

## LIST OF FIGURES—Continued

|           |                                                      |     |
|-----------|------------------------------------------------------|-----|
| Figure 40 | CAN XL Receiver                                      | 60  |
| Figure 41 | Preamble Verification at the Receiver                | 62  |
| Figure 42 | The Result of 1-bit delay Preamble Verification Core | 62  |
| Figure 43 | SHA 512 RTL                                          | 63  |
| Figure 44 | SHA 512 Output                                       | 63  |
| Figure 45 | CAN XL Sender RTL                                    | 64  |
| Figure 46 | CAN XL Sender Verification                           | 64  |
| Figure 47 | CAN XL Receiver RTL                                  | 65  |
| Figure 48 | CAN XL Sender Verification                           | 66  |
| Figure 49 | Transmitter/Master Flowcharts                        | 72  |
| Figure 50 | Secured CAN FD protocol                              | 72  |
| Figure 51 | Receiver / Slave Flowchart                           | 75  |
| Figure 52 | MATLAB HDL Coder of CAN security Protocol -Tx        | 99  |
| Figure 53 | Fixed-Point Data Type Representation-Tx              | 100 |
| Figure 54 | CAN FD Tx Flowcharts- Lightweight Cryptography       | 105 |
| Figure 55 | CAN FD Rx Flowchart - Lightweight Cryptography       | 106 |
| Figure 56 | PC-FPGA Setup -ARTY A7 board                         | 112 |
| Figure 57 | FPGA in the Loop implementation of CAN FD TX         | 113 |
| Figure 58 | FPGA in the Loop Implementation of CAN FD Rx         | 114 |

## LIST OF FIGURES—Continued

|           |                                                       |     |
|-----------|-------------------------------------------------------|-----|
| Figure 59 | CAN F D Tx Simulation Verification by Vivado          | 119 |
| Figure 60 | CAN FD Tx Simulation Verification by Aldec Active-HDL | 119 |
| Figure 61 | CAN FD Rx Simulation Verification by Vivado           | 120 |
| Figure 62 | CAN FD Rx Simulation Verification by Aldec Active-HDL | 121 |

## CHAPTER ONE

### INTRODUCTION

The rapid evolution of software-defined vehicles (SDVs) and zonal Electrical/Electronic (E/E) architectures has significantly increased the demand for secure and reliable in-vehicle communication systems. Controller Area Network (CAN) and its advanced variants, including CAN FD and CAN XL, remain fundamental communication protocols in modern automotive systems. However, these protocols inherently lack essential security mechanisms such as authentication, data integrity verification, and protection against replay and spoofing attacks.

To address these limitations, existing standards, particularly AUTOSAR Secure Onboard Communication (SecOC), introduce message authentication codes (MACs) and freshness values (FVs). Despite these enhancements, SecOC relies on deterministic, monotonically increasing counters to generate freshness values. This approach introduces a critical vulnerability: predictability. Adversaries can exploit this predictability to perform desynchronization and replay attacks, potentially causing legitimate messages to be rejected or enabling denial-of-service (DoS) conditions.

This dissertation proposes a hardware-efficient security protocol that replaces linear freshness counters with deterministic non-linear freshness values. The proposed approach integrates lightweight authentication mechanisms and is validated using FPGA-in-the-loop (FIL) methodology to ensure real-time performance, synchronization reliability, and robustness against cyber-attacks. The main contributions of this work are summarized as follows:

- Development of a deterministic non-linear freshness value (FV) generation mechanism
- Design of a chaotic FV enhancement to improve entropy and unpredictability
- Implementation of a lightweight XOR-based authentication tag suitable for FPGA-based systems
- Design of a full transmitter/receiver (Tx/Rx) synchronization and resynchronization protocol
- Validation of the proposed approach using FPGA-in-the-loop (FIL), demonstrating real-time performance

### 1.1 Controller Area Network (CAN)

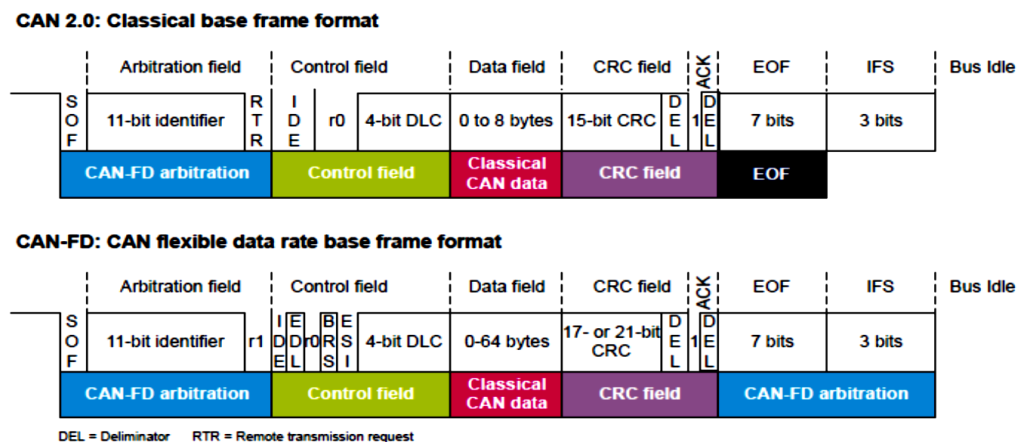
CAN remains one of the most widely deployed in-vehicle communication protocols for enabling real time data exchange among ECUs in modern automotive systems [1, 2] Its low implementation cost, deterministic arbitration mechanism, and reliability have contributed to its extensive adoption across powertrain, body control, infotainment, and advanced driver assistance subsystems. As modern vehicles increasingly depend on distributed ECU architectures, maintaining reliable secure communication has become critical for ensuring functional safety, real time responsiveness, and operational integrity. Despite its widespread adoption, the CAN protocol was originally designed with no embedded cybersecurity mechanisms such as authentication, integrity verification, or confidentiality protection. Consequently, CAN networks remain vulnerable to several cyberattacks, particularly replay attacks, spoofing attacks, and message injection attacks [3, 4]. In replay attacks, adversaries capture legitimate CAN frames and retransmit them

at later time instance to manipulate ECU behavior or bypass system verification mechanisms. Such attacks can introduce unsafe operating conditions and compromise critical vehicle functions [5]. Since CAN lacks native sender authentication and freshness validation, replayed messages may be interpreted as legitimate frames by receiving ECUs, potentially causing unauthorized state transitions, denial of service conditions, or unsafe actuator responses [6, 7]. Such attacks may manipulate sensor data and control decisions, thereby affecting vehicle safety and operational reliability [8] Consequently, mitigating these vulnerabilities is essential for maintaining trust in autonomous and software-defined vehicle technologies [9].

Several experimental studies have demonstrated the practical feasibility of replay attacks against automotive communication systems. For example, prior work has shown that attackers can intercept and retransmit CAN traffic using relatively inexpensive hardware platforms, thereby compromising communication integrity and ECU trust relationships [10]. The system was able to successfully extract the information stored in each packet at a rate of about 70 Mbps, which corresponds to the data rate used by the CAN bus protocol commonly used in modern automobiles [10]. To mitigate those, modern automotive security frameworks integrate cryptographic mechanisms such as encryption, message authentication, and freshness verification, However, conventional encryption, alone can not fully prevent replay attacks unless combined with synchronized aware freshness validation techniques [11]. While this approach is effective at protecting against some types of attacks, it is insufficient to protect against replay attacks that involve capturing data from unencrypted communications channels [12].

The current state of the automotive industry has been hampered by a number of limitations with the CAN network. CAN is a communication protocol that has been used in automobiles for many years, but it has several limitations [13]. A major limitation with CAN is the way that the data is transmitted over the network. CAN is a synchronous protocol which means that the transmission of data is synchronized to the clock of the network device. This works well in many applications but creates issues when the data to be communicated is irregular or contains a large amount of data. As a result, most CAN networks only transmit small amounts of data at one time, which can lead to long delays in the network if the transmission must be repeated several times before it is received correctly [14].

Recently securing messages has played a major role over Automotive CAN networks [15, 16] After their enhancement, the payload field has been suitable to implement security algorithms. For example, CAN FD and XL networks are the new generation [17] of 8-Byte CAN networks with 64 bytes and 2048 bytes respectively as shown in Figure 1 and 2.



**Figure 1** CAN and CAN FD Architecture [17]

Whereas CAN FD coexists already with CAN networks in automotive industry and

| SOF | Arbitration | Control | Data (field)   | CRC    | ACK   | EOF   |
|-----|-------------|---------|----------------|--------|-------|-------|
| 1   | 15 bit      | 81 bit  | 1 to 2048 byte | 36 bit | 6 bit | 7 bit |

**Figure 2** CAN XL Architecture [18]

adopts security measures according to AUTOSAR (Automotive Open System Architecture) specifications, CAN XL security measures are still under development. One of the measures have been undertaken regarding to AUTOSAR specifications is freshness values. To tackle replay attacks, freshness values embedded in the secured message over developed CAN networks. These values are to be sent and received within a guaranteed time window. Therefore, such attacks can be mitigated. That feature can be accomplished either by freshness counters or freshness time stamps as mentioned in AUTOSAR specifications [19]. Although both methods provide capabilities to reduce this attack, they have faced some challenges that can be exploited by an adversary to conduct such attacks. On one hand, software-based time stamp algorithms between the master ECU and Slave ECU suffer inaccuracy in time leading to synchronization issues between senders and receivers. Depending on hardware-Based timestamps to improve that issue adds more cost on the existing CAN infrastructures. With high accuracy levels of freshness counter-based algorithms, on the other hand, this method was not fully investigated.

Cryptographic hash functions and Message Authentication Codes (MACs) are two fundamental cryptographic mechanisms used to protect modern communication systems.

Cryptographic hash functions generate a fixed-length digest from an input message and are primarily employed to verify data integrity, whereas MAC algorithms combine a secret key with the message to provide both data integrity and source authentication. Among the cryptographic hash algorithms, the Secure Hash Algorithm 2 (SHA-2) family is the successor to SHA-1 and has become the preferred standard for modern security applications because of its strong resistance to collision and preimage attacks. Consequently, SHA-2 and MAC-based mechanisms are widely adopted in secure communication protocols and automotive cybersecurity applications to protect transmitted data against unauthorized modification and forgery [20].

Although both SHA-2 and MAC mechanisms contribute to communication security, they serve different purposes. SHA-2 is designed to generate a unique message digest for integrity verification, whereas MAC algorithms additionally authenticate the origin of the transmitted message through the use of a shared secret key. In practical automotive systems, cryptographic hash functions are frequently incorporated into authentication schemes such as Hash-based Message Authentication Codes (HMACs), while lightweight MAC constructions are often adopted to satisfy the stringent timing and computational requirements of embedded Electronic Control Units (ECUs).

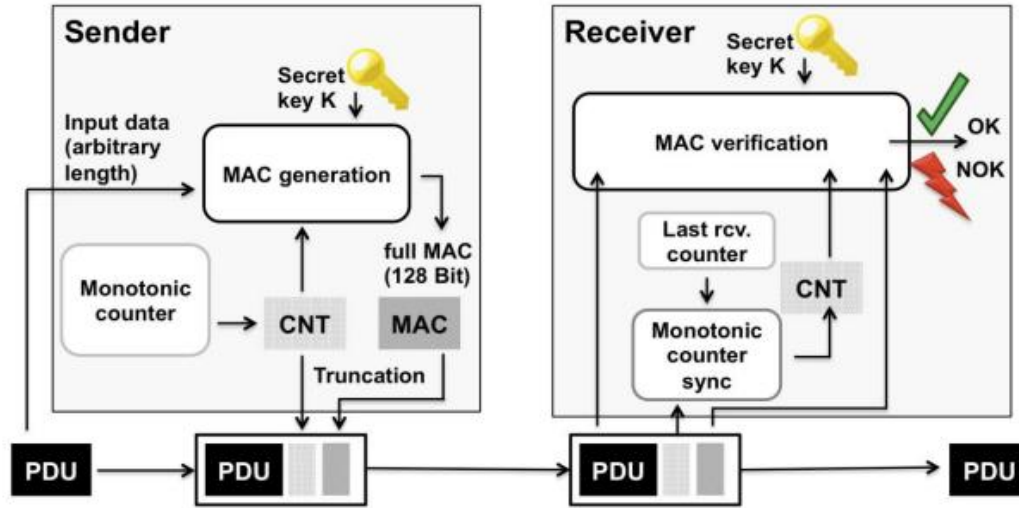
Implementing cryptographic protection in automotive communication systems requires balancing security with hardware efficiency. While cryptographic algorithms such as SHA-256 provide strong security properties, their implementation may introduce additional computational complexity, hardware resource utilization, and latency in real-time embedded platforms. Consequently, modern automotive security research has

increasingly focused on developing lightweight authentication mechanisms capable of preserving message integrity and authenticity while satisfying the strict real-time and resource constraints of CAN-based communication systems. These challenges provide part of the motivation for the lightweight hardware-oriented security protocol proposed in this dissertation. [21].

## 1.2 Literature Review

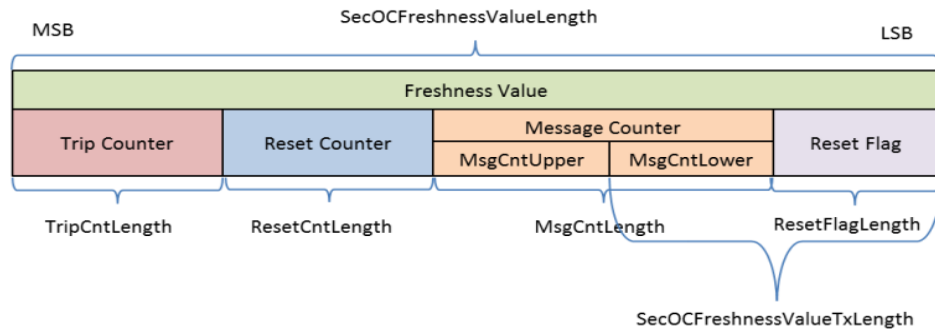
To better understand the limitations of existing approaches, the following section reviews current automotive cybersecurity mechanisms and their associated challenges. After developing the automotive infrastructure, adding security measures enables the protection for bus messages. In order to challenge those measures, a number of attacks should be immune to. For example, replay attacks are easy to perform in terms of simplicity once an access takes place. In general, AUTOSAR standards [22] have adopted a mechanism based on their prospective of replay attacks counter measurements as shown in Figure 3. In a solution based on a single freshness counter, the FVM provides the freshness value to network-connected nodes and increments the counter each time a frame is transferred across the communication channel. Both the sender and receiver's FCs should be increased at the same time to ensure freshness. As a result, the FC needs to be increased for each frame that leaves after being verified on the receiving end. Trip Counter, Reset Counter, Message Counter, and Reset Flag are the four counters used in the Multiple Freshness Counters technique as shown in Figure 4 and 5. The Trip and Reset Counters are both under the control of the FVM. According to predetermined conditions, these

counters are increased . For instance, at regular intervals, the Reset Counter is increased by 1. The ClearAcceptanceWindow option must be used with the multiple freshness



**Figure 3** AUTOSAR replay attacks counter measures [23]

counter strategy to reflect an acceptable freshness interval and prevent potential synchronization issues [23].

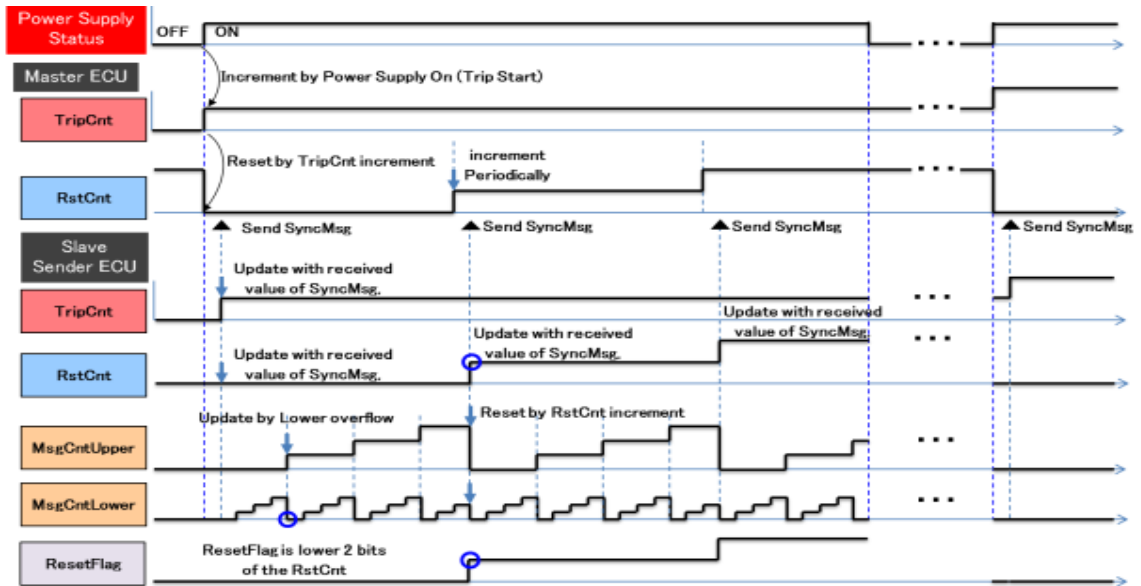


**Figure 4** Freshness value structure [23]

Figure 4 shows the maximum size of the freshness value field which is 64 bits including the maximum size of the trip counter (24 bits), the reset counter (24 bits), the

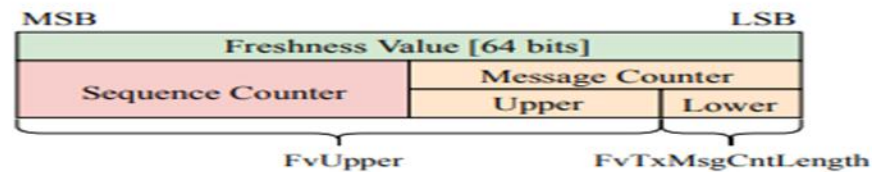
message counter (48 bits ) and rest flag. Firstly, trip counter increments by 1 only in 4 states: (1), the master ECU (freshness value manager) starts, (2) on wakeup, (3) on rest and (4) when Ignition status changes from off to On. periodically, the rest counter increments by 1. To illustrate more, Figure 5 exemplifies the behavior of Freshness values at master and slave sides. To start with, the Trip Counter at the master side shows that its increment by 1 occurs after the power supply turns on. This state does not change unless the power supply changes its state from “OFF “to “ON”. Consequently, Reset Counter increments by 1 and keeps incrementing periodically if sync messages received (which is around 1 second). At the slave side, updating Trip Counter occurs with every sync message received by the slave. Similarly, Reset Counter updates its value. Message Counters (high and low) increments by 1. Once reset counter is being updated, Message Counters rest to zero. While Reset Flag is considered as a reference if verification is needed. Since it is the lower 2 bits of Reset Counter, the flag follows the same pattern of that counter [22].

In general, it follows similar AUTOSAR specifications with slight change [19]. These changes are based on two situations where AUTOSAR fails to reconstruct freshness values. It focused on a new security method of the counter approach to enhance the resynchronization speed. This method was based on adopting two operation modes ;where the first mode is similar to AUTOSAR specifications and the second mode works such as expediting requesting synchronization messages in case it needed instead of waiting. Furthermore, Figure 6, 7 and 8 show reconstructing the freshness values.

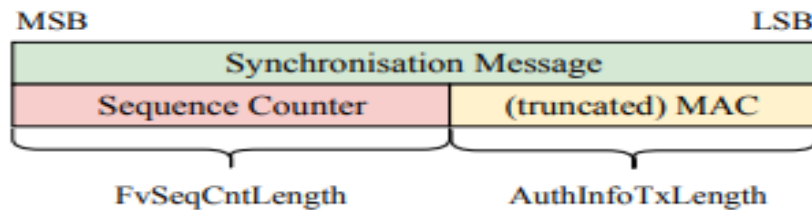


**Figure 5** the Behavior of Freshness Values

As seen in Figure 6, it displays that sequence counter has replaced the trip and reset counters in Figure 4. Its size becomes 28 bits long. Therefore, the message counter becomes 36 bits long.

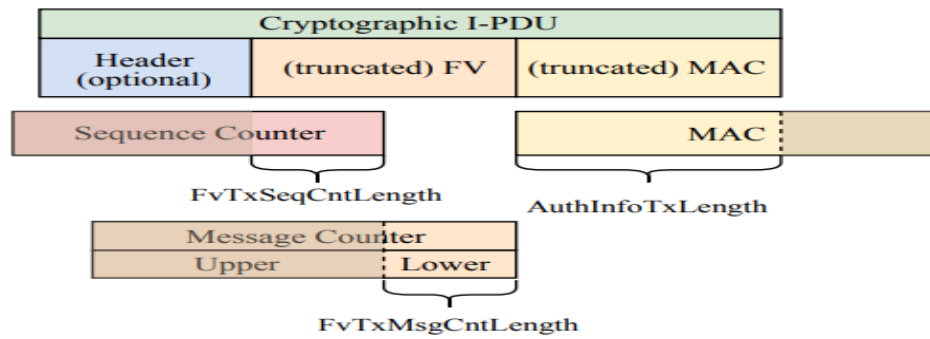


**Figure 6** Freshness Value Reconstruction [12]



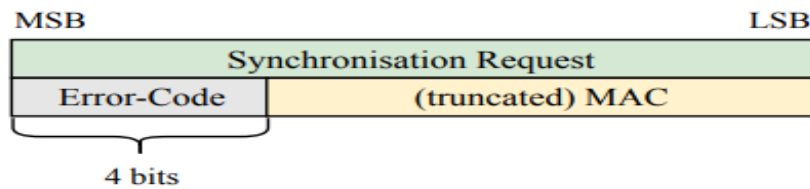
**Figure 7** Synchronization Message [12]

In Figure 7, the entire synchronization message payload equals 56 bits long (28 bits each of Sequence Counter and truncated MAC). As well as, Figure 8 illustrates how truncated fields have been set for each. Firstly, MAC is truncated just high significant bits. Secondly, freshness value is truncated by combining two parts of low significant bits of sequence and message counters.



**Figure 8** Secured I-PDU Including Truncated FV and MAC [12]

As seen from Figure.9, the synchronization request payload is divided into two parts including error code (low 4 bits of sequence counters) and truncated MAC. With this mentioned enhancement, the frequency of the sync messages and on demand sync requests were not better than the AUTOSAR specifications in terms of time doamin.

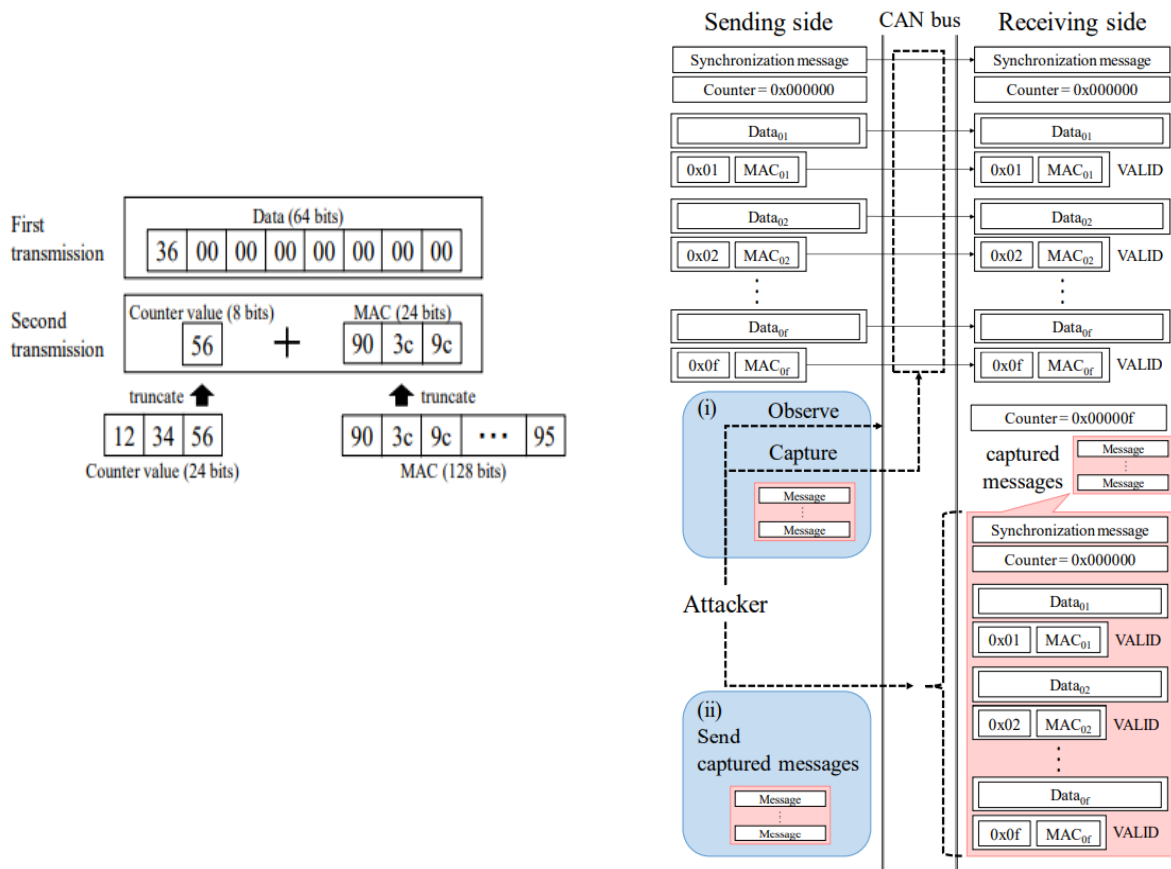


**Figure 9** Synchronization Request [12]

For example, the sync message frequency for [12] is up to 5 Hz which is 5 times of AUTOSAR (1 Hz). Aslo, the frequency of sync requests equals 1 Hz which is the same as

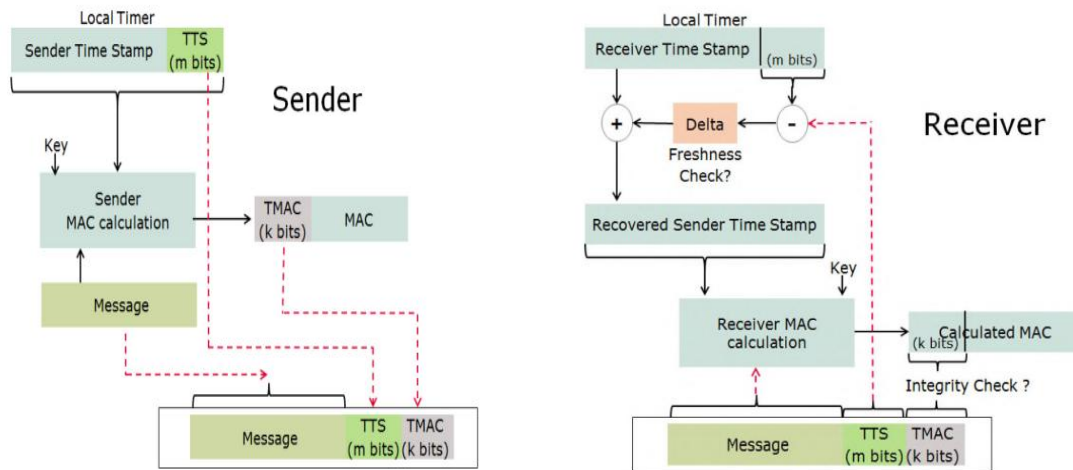
AUTOSAR sync messages. Since the sequence counter characteristic was not mentioned in details, another reason is considered to not follow this option.

As [24] proposed a method that has performed a replay attack based on resynchronization messages of counter approaches, which was implemented successfully. In that, the attacker was assumed to have an access to the bus, and it was able to read the messages on the bus as illustrated in Figure 10. After observing the messages over the bus, it recommended that securing the synchronization messages should be considered. However, this approach was based on the simple approach of freshness value reconstruction.

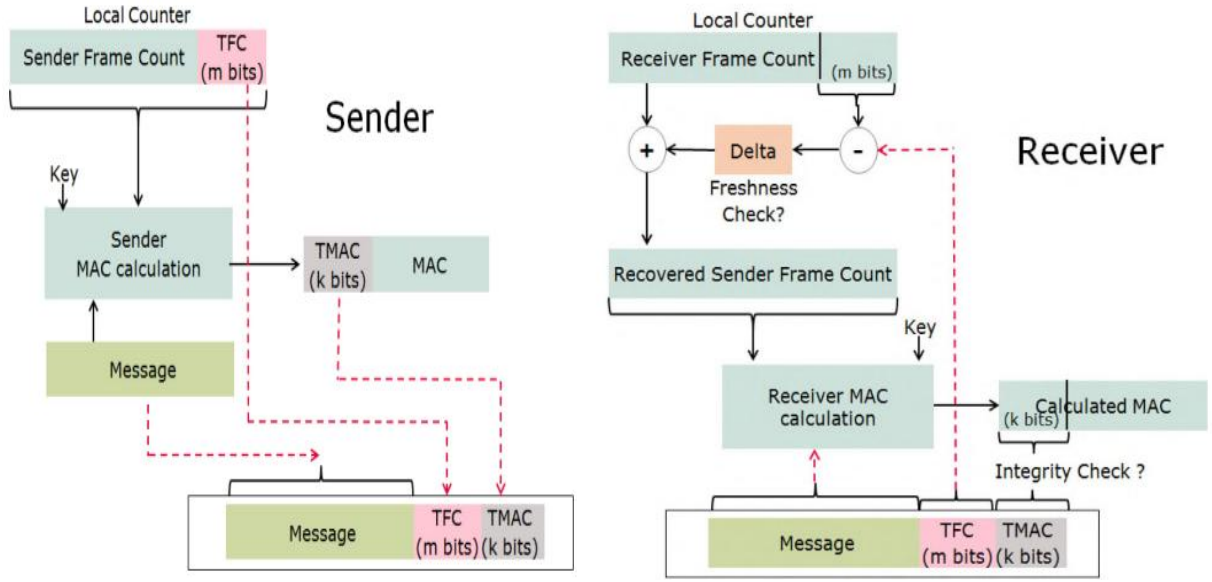


**Figure 10** Counter Approach Replay Attack [13]

According to [25], it investigated AUTOSAR specification in terms of freshness value method (time and counter) as shown in Figures 11 and 12. For the time approach, the major limitation is the oscillator deviation by time of its own clock which is called delta. As a result, it loses the synchronization between ECU senders and receivers. This needs to be corrected by synchronization approaches. Since CAN networks do not support Master clocks, it is hard to implement any mechanism to correct that. Therefore, it has put all attention on the counter approach limitation solutions such as sending a sync message. In case of any loss of synchronization, it uses a local counter at the receiver to fix a replay window and etc. However, the synchronization process was not implemented as planned for the counter approach and no solutions presented for the time stamps approach. Whereas [26] proposed a hardware based counter solution adding a counter of each bus. This proposal has been claimed that was proven theoretically not practically.



**Figure 11** Freshness Values Based on Time Stamps [15]



**Figure 12** Freshness Values Based on Counters [15]

Based on the above discussion, all methods have been implemented and verified over microcontrollers without any mechanism of validations. However, FPGA has been used for CAN networks [27], and it has also been used as a hardware security approach. For example, Hash functions and Encryption algorithms have been implemented into it [28, 29, 30]. In addition, [31] proposed AES 128 and SHA-1 as security algorithms based on ECUs in automotive securing CAN FD messages, while [32] developed a security measure using AES 128 and SHA256 based on CAN FD protocol.

Automotive cybersecurity solutions have primarily relied on AUTOSAR SecOC, which utilizes truncated freshness counters and MACs. While effective under ideal conditions, counter-based mechanisms suffer from predictability and synchronization issues. Chaos-based cryptographic methods provide high entropy but lack protocol-level features such as authentication and synchronization. Similarly, lightweight cryptographic algorithms

focus on encryption efficiency but do not address replay protection or freshness validation. FPGA-based validation has been used to evaluate cryptographic primitives; however, most works fail to validate full transmitter/receiver systems under realistic timing conditions. This gap motivates the proposed integrated protocol design. Despite these advancements, several limitations remain unresolved, which motivate the research problem addressed in this dissertation.

### 1.3 Research Problem

One of the primary research problems in automotive security is the threat of replay attacks. A replay attack occurs when an attacker intercepts a message sent between two ECUs, records the message, and then later replays the message to impersonate one of the ECUs. This can allow an attacker to gain unauthorized access to the vehicle or its systems, or to manipulate the behavior of the vehicle in dangerous ways. There are several potential countermeasures that have been proposed to address the threat of replay attacks in automotive systems. These include [33]:

- **Message Authentication:** One approach is to use message authentication techniques such as MAC algorithms, which can help to ensure that messages are not modified or replayed.
- **Timestamping:** Another approach is to use time-stamping techniques to ensure that messages are not replayed after a certain period of time. This can help to limit the window of opportunity for attackers to carry out replay attacks.
- **Nonce-Based Approaches:** A third approach is to use nonce-based approaches, which involve the use of a one-time value (nonce) that is included in each message. The

receiver can then verify that the nonce is unique and has not been used before, which can help to prevent replay attacks.

- **Cryptographic Techniques:** Finally, cryptographic techniques such as digital signatures and encryption can also be used to help prevent replay attacks. Digital signatures can help to ensure the authenticity of messages, while encryption can help to prevent eavesdropping and unauthorized access to the vehicle's systems. Overall, the research problem of automotive replay attacks countermeasures requires developing and implementing effective techniques to prevent these attacks, while also ensuring that these techniques are scalable, efficient, and practical for use in automotive systems. This involves exploring the strengths and weaknesses of various approaches, and developing solutions that can provide a high level of security while lessening any negative impact

It's important to note that no security mechanism is fool proof, and determined attackers may still be able to find ways to bypass these measures. Therefore, it's important to use multiple layers of security and to keep systems up to date with the latest security patches and protocols. Therefore, freshness value is commonly used to protect against replay attacks in various communication protocols. A replay attack is a type of network attack where a valid data transmission is intercepted and then re-transmitted by an attacker in order to gain unauthorized access to a system or data. In order to protect against replay attacks using freshness values, a system can incorporate a timestamp or sequence number into the data being transmitted. This value can be checked by the receiver to determine if

the data is fresh or not. If the freshness value is too old, the receiver can reject the data. There are some weaknesses of freshness value approached need to be strengthened.

One weakness of using freshness values to protect against replay attacks is the potential for synchronization issues. If the clocks of the sender and receiver are not synchronized, the receiver may consider valid data as old and reject it. Alternatively, the receiver may accept old data as fresh and allow unauthorized access. This can be a challenge in automotive systems, where there may be many different components and devices, each with its own clock.

As well as, the use of freshness values can introduce additional overhead in the system, which can affect performance and scalability. This is especially true if the freshness values are generated using computationally intensive techniques, such as cryptographic algorithms.

Some freshness value techniques may be predictable, which can make them vulnerable to attack. For example, if a timestamp is used as the freshness value, an attacker may be able to predict the value by using the clock skew or by observing the behavior of the system. The freshness value approach can also be vulnerable to attacks that occur within the replay window. For example, if the replay window is set to be too long, an attacker may be able to replay messages within the window and avoid detection.

As the number of nodes in a system increases, the complexity of managing synchronization requests also increases. This can lead to scalability weakness and can make it difficult to maintain consistency across all nodes.

Synchronization requests can be less effective in scenarios where nodes fail or to be offline. In such cases, it can be challenging to maintain consistency across all nodes, particularly if the failed nodes contain critical data. As well as synchronization requests may not be able to handle concurrent updates to the same data by multiple nodes. This can lead to conflicts and inconsistencies in the data across the nodes.

Finally, there is a trade-off between the frequency of synchronization requests and the amount of data that needs to be transferred. If synchronization requests are too frequent, it can result in excessive network traffic and performance degradation.

In this thesis, we will study the freshness value approach in two CAN FD and XL to cover most weaknesses.

#### 1.4 Research Design / Methodology

To address the identified challenges, a structured research methodology is proposed, as described in the following section. Recently embedded systems have been used to conduct research over CAN network security of Automotive such as Microcontrollers and FPGAs. They are used to validate security measures generally. For example, both [12,14] establish their experiments over microcontroller platform. However, in [20, 21] FPGAs are used. Depending on several factors, selecting the platform should facilities validations and verifications of any security design model. Therefore, the proposed design is divided into three steps implemented on the developed CAN networks such as CAN FD and CAN XL based on FPGA boards.

- Firstly, CAN FD buses are used to enhance the counter-based approach by embedded that message counter value into the cryptographic message by AES 128 besides hashing the whole message by SHA 256.
- Secondly, enhancing the time stamps approach was based on adding the cross correlation function to its calculations instead of relying on the time stamp method. The last part was to improve our both methods by adding a novel technique based a trusted history of among ECUs to validate any red flags on any questioning secured messages. With that, if any synchronization loss occurs, there should be more error feedbacks to acknowledge the ECU sender.
- Finally, validating those designs improvements should be conducted by MATLAB simulations using the FPGA In the Loop technology. Xilinx was used for both first and second designs. However, MATLAB has been used only with calculation of cross correlation function. As we can use it as a verification tool to debug our designs to find out easily any flaws in our development stages.

## 1.5 Research Objectives

### 1.5.1 Design FPGA-based Security Protocols for CAN FD and CAN XL Communication Systems

- Implement FPGA-Based AES-128, SHA-256, and SHA-512 Cryptographic Cores based on FPGA
  - Developing the Hardware Design Using VHDL
  - Using Xilinx boards
  - Using 14.3 design Vivado tool and SIM tool

- Verifying the encryption and decryption process of AES 128 separately and validating it to NIST Vector.
  - Verifying the encryption process of SHA 256 and 512 and validating it to a reference
- Utilizing message counter approach for freshness value method
  - Utilizing the cross-correlation function and implementing it based on FPGA.

#### 1.5.2 Validate the Proposed Protocol Through MATLAB Simulation

After simulating our design by MATLAB/Simulink, FPGA In the Loop will be used to verify our design.

#### 1.5.3 Evaluate the Proposed Protocol Under Replay Attack Scenarios

For conducting replay attacks, the same message counter and freshness value will be sent twice and then the receiver should be able to detect this attack and update the freshness value by sending synchronization request message.

#### 1.5.4 Experimentally Evaluate the Proposed Protocol

Over the last two years of my research, 2 steps have been applied to ensure the viability of the proposed solution. The proposed solutions based on both freshness values designs were successfully verified by FPGAs. Where each solution has been divided into stages to be able to verify these separately and in total. However, validation of these solutions using FPGA in the loop should be selected so that the whole designs can be examined in a real time environment. That represents the current steps for our thesis

- Enhance the error messages that exists in case of there are no successful events between senders and receivers(sync message). Also, verifying the approach in a real time by FPGA In the Loop techniques should be conducted.
  - Using MATLAB Vivado design Tool in a real time (virtix 6 board or suitable board in terms of I/O and cost)
- Increase the level of protection by adding trusted history between parties as a novel method for verifying the synchronization message.
  - On demand authentication. When the receiver suspects the secured message, a history trusted message request can be sent to the sender
  - The format of the message is normal secured messages from the past history between the sender and receiver.
  - It should not be fixed every time

In order to achieve these objectives, this research work will leverage a variety of proposed hardware designs, including the verification and validation of those designs. The results of this project will be presented in a comprehensive dissertation, which will be accompanied by a set of recommendations for future work.

This research contributes to the development of a hardware-efficient CAN FD security protocol that leverages deterministic non-linear and chaotic Freshness Value (FV) generation to enhance communication security in automotive networks. Unlike traditional counter-based mechanisms, the proposed approach introduces a

synchronized, unpredictable FV sequence that significantly increases resistance to replay, spoofing, and desynchronization attacks.

The protocol eliminates the reliance on linear and globally synchronized counters by enabling both the transmitter (Tx) and receiver (Rx) to independently regenerate identical FV sequences without explicit transmission. This reduces communication overhead while maintaining deterministic synchronization between nodes.

To further strengthen security, a chaotic FV extension is introduced, combining trigonometric entropy sources, ARX-based transformations, and whitening techniques to achieve high entropy and near-random behavior while preserving reproducibility. This enhancement improves resistance to sequence prediction, denial-of-service (DoS), and desynchronization attacks.

In addition, a lightweight XOR-based authentication tag is proposed, integrating FV slices, AES-128 ciphertext, and key-expansion values. This design ensures message integrity and authenticity with minimal computational overhead, making it suitable for resource-constrained Electronic Control Units (ECUs).

The proposed protocol is validated using FPGA-in-the-Loop (FIL) methodology, enabling real-time hardware evaluation of synchronization, latency, and security performance. Experimental results demonstrate deterministic Tx/Rx synchronization, low latency ( $\sim 170$  ns Tx,  $\sim 160$  ns Rx), and high throughput ( $\sim 123$  Mbps), confirming the feasibility of the approach for real-world automotive systems.

## 1.6 Thesis Outlines

This thesis is organized as follows:

Chapter 1 introduces the research topic and motivation, presents the Controller Area Network (CAN) context, reviews the relevant literature, defines the research problem, describes the research methodology and objectives, and outlines the structure of the dissertation. For Chapter 2, it presents the technical background required for this research, including hardware security in the automotive industry, embedded system types, encryption algorithms, hash functions, and the use of FPGA-in-the-Loop (FIL) with MATLAB for hardware validation. In Chapter 3, the FPGA-based design of security measures for CAN FD and CAN XL communication systems have been described, including the proposed hardware architectures, sender and receiver designs, simulation results, discussion, and conclusions. Chapter 4 presents the proposed CAN FD security protocol, including the threat model, protocol evolution, non-linear and chaotic Freshness Value (FV) generation, lightweight authentication tag construction, transmitter/receiver validation, FPGA-in-the-Loop implementation, and statistical security evaluation. While Chapter 5 provides the discussion and overall conclusion of the proposed work, including interpretation of the experimental results, security implications, and performance observations. Finally, Chapter 6 outlines future research directions, including integration of machine learning for intrusion detection and expansion toward multi-node and zonal automotive architectures.

## CHAPTER TWO

### AUTOMOTIVE EMBEDDED SECURITY BACKGROUND

#### 2.1 Hardware security in the Automotive industry

Hardware security is a fundamental component of automotive cybersecurity because it protects the vehicle from various cyber threats [34]. Furthermore, hardware security involves implementing mechanisms to secure the hardware components of the vehicle against potential attacks or vulnerabilities [35]. One important hardware security mechanism in automotive security is secure boot [36], which verifies the integrity of the software and firmware that runs on the vehicle's hardware. In other words, secure boot ensures that only trusted and verified software can run on the vehicle hardware, thereby preventing attackers from exploiting vulnerabilities in the software or firmware [37]. Another important aspect of hardware security is the use of cryptographic hardware modules for securing sensitive data and managing the storage of cryptographic keys [38]. The module is designed to be tamper-resistant, preventing attackers from accessing or manipulating sensitive information. In particular, hardware security is a component of automotive security, and it is essential to implement robust measures to ensure that the vehicle hardware components are secured and potential attacks are mitigated.

#### 2.2 Embedded System Types [29]

Typical hardware platforms implemented in automotive security include a range of electronic control units (ECUs) and associated hardware that are used to manage various systems and functions within a vehicle. Some of the key hardware platforms used in

automotive security include: Engine Control Unit (ECU): The engine control unit manages engine-related functions such as fuel injection, ignition timing, and emissions control. As one of the most critical ECUs within the vehicle, it requires robust protection against cyberattacks that could compromise vehicle operation. The ECU uses various sensors to monitor the engine's performance and adjust its operation to optimize efficiency and performance. Anti-lock Braking System (ABS): The ABS monitors wheel speed and regulates braking force to prevent wheel lock during emergency braking. Because it performs safety-critical functions, protecting communication with the ABS is essential for maintaining vehicle safety. Advanced Driver Assistance Systems (ADAS): ADAS integrates data from cameras, radar, and LiDAR sensors to support functions such as lane keeping, adaptive cruise control, and automatic emergency braking. The large volume of exchanged sensor data makes these systems particularly dependent on secure and reliable communication. Infotainment Systems: Infotainment systems provide a range of entertainment and communication features to vehicle occupants, including audio and video playback, navigation, and smartphone integration. Infotainment systems can also be used for vehicle diagnostics and maintenance. These automotive functions are typically implemented using embedded platforms such as microcontrollers and FPGAs, depending on the computational, security, and real-time requirements of the application.

Microcontrollers and FPGAs are widely used in the automotive industry to enhance vehicle security features [39, 40]. Typical automotive security applications of microcontrollers and FPGAs include:

- **Cryptography:** Microcontrollers and FPGAs implement cryptographic operations to secure data transmission and storage. For instance, a microcontroller can be used to encrypt and decrypt data, such as data transmitted between the key fob and the vehicle, and data stored on the vehicle internal network.
- **Keyless Entry Systems:** Keyless entry systems are popular in modern vehicles, and they rely on microcontrollers and FPGAs to secure communication between the key fob and the vehicle. The microcontroller or FPGA can be used to implement encryption algorithms, authenticate the key fob, and validate the data exchanged between the key fob and the vehicle.
- **Immobilizers:** Microcontrollers and FPGAs can be used to implement immobilizer systems that prevent unauthorized access to the vehicle. The microcontroller can authenticate the key or electronic fob, and if authentication fails, it can trigger an immobilizer system that disables the engine or fuel system, preventing the vehicle from starting.
- **Anti-Theft Alarms:** Microcontrollers and FPGAs can be used to control and manage the sensors that detect unauthorized access or movement of the vehicle. The microcontroller can activate an alarm, sound the horn, or turn on the vehicle's lights, alerting the owner and deterring potential thieves.
- **Vehicle Tracking Systems:** Microcontrollers and FPGAs can be used to implement GPS tracking systems that allow the owner or law enforcement to locate a stolen vehicle. The microcontroller can collect GPS data and transmit it to a remote server for real-time tracking and monitoring.

Overall, microcontrollers and FPGAs play a critical role in enhancing automotive security. These devices can be programmed to perform complex security functions and control various sensors and actuators in the vehicle. The use of microcontrollers and FPGAs in automotive security is expected to grow as automakers continue to add more security features to their vehicles.

However, FPGAs have emerged as a viable alternative to microcontrollers, which have been the preferred choice for embedded system designers for a long time [41]. This has been due to the increasing complexity and performance demands of modern applications. Customization and Flexibility, Longer Lifecycle and Future-Proofing, FPGA-in-the-Loop (FIL) Capability [42], Low Latency, Scalability, Parallelism, and raw computational capacity are all unique features of FPGAs that make them a superior choice for specific applications [43, 44].

Reconfigurability is one of the primary benefits of FPGAs, as it enables developers to customize the hardware to meet the unique requirements of their application. This can lead to substantial performance enhancements in comparison to a general-purpose microcontroller, as the hardware can be tailored to the specific task at hand. Furthermore, FPGAs offer a degree of adaptability that is exceedingly challenging to accomplish with conventional microcontrollers, as they can be reprogrammed in the field [45].

Additionally, FPGAs provide superior future-proofing and a longer lifespan than microcontrollers. Although microcontrollers are frequently associated with a particular hardware architecture, FPGAs can be upgraded or reconfigured to accommodate evolving needs, thereby guaranteeing a system's extended useful life [46].

FPGAs' parallel processing capabilities are another significant advantage, making them particularly advantageous in high-performance applications such as digital signal processing, image processing, and real-time control systems. In comparison to sequential microcontroller-based systems, FPGAs are capable of executing multiple tasks simultaneously, which leads to significantly quicker processing times. FPGA-based systems can be an appealing option for time-critical applications due to their minimal latency and parallelism [47].

In addition, FPGAs are well-suited for applications that necessitate real-time responsiveness, including industrial automation, robotics, and aerospace systems, due to their deterministic behavior and minimal latency [48]. Conversely, microcontrollers may encounter difficulties in satisfying the stringent timing specifications of these applications, which may result in system malfunctions or suboptimal performance [49]. While selecting an appropriate embedded hardware platform is essential for achieving reliable real-time performance, secure in-vehicle communication also depends on the implementation of efficient cryptographic algorithms capable of protecting transmitted data against unauthorized access, message modification, and replay attacks. Therefore, the following section reviews the cryptographic techniques commonly employed in CAN-based automotive networks.

### 2.3 Encryption Algorithms and Hash Functions in CAN networks

Since CAN and CAN FD networks transmit safety-critical and sensitive information among vehicle ECUs, encryption algorithms and cryptographic hash functions are

employed to protect communication against cyberattacks. The primary cryptographic techniques used in CAN networks are summarized as follows:

Symmetric Encryption Algorithms use the same key for both encryption and decryption [50, 51]. Among the symmetric encryption algorithms employed in automotive communication systems, the Advanced Encryption Standard (AES) is one of the most widely adopted [52]. AES is a symmetric block cipher with key lengths of 128, 192, or 256 bits, and is considered to be highly secure [53]. Asymmetric Encryption Algorithms use two different keys are used for encryption and decryption [50,51]. One of the most widely used asymmetric encryption algorithms in automotive security applications is the Rivest-Shamir-Adleman (RSA) algorithm [54]. RSA employs a public key and a private key pair and is commonly used for digital signature and secure key establishment [55]. Hash functions generate a fixed-size digest from an input message, which is commonly used to verify the integrity of transmitted data [50, 51]. Automotive embedded systems are constrained by limited computational capability, memory, power consumption, and real-time requirements. Consequently, cryptographic algorithms used in vehicles should provide strong security while maintaining computational efficiency and low hardware overhead [56].

The Secure Hash Standard (SHS), specified in FIPS 180-4, defines the SHA-1 and SHA-2 families of cryptographic hash algorithms, which generate fixed-length message digests for verifying data integrity and detecting unauthorized message modifications [57]. Although FIPS 180-4 specifies both SHA-1 and SHA-2 algorithms, SHA-1 is no longer recommended for new security applications because of its demonstrated

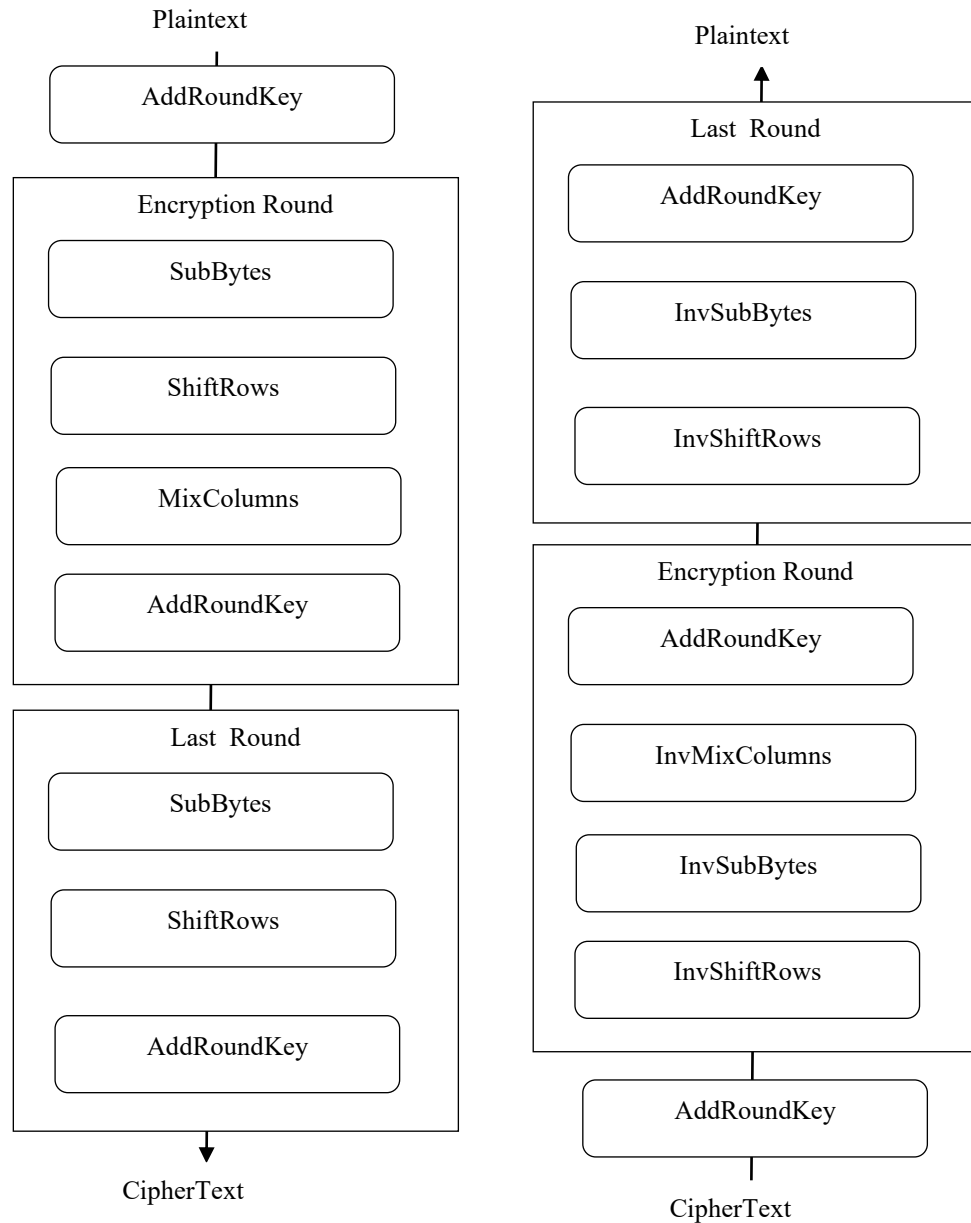
vulnerability to collision attacks. Consequently, NIST has announced that future revisions of the Secure Hash Standard will remove the SHA-1 specification, making SHA-256 and other SHA-2 algorithms the preferred choices for modern cryptographic applications [57, 58].

Several hash functions have been used in automotive communication systems, including the SHA-2 family of hash algorithms, while SHA-1 and MD5 are found primarily in legacy systems [50, 51]. These hash functions are commonly employed in authentication mechanisms, such as Hash-based Message Authentication Codes (HMACs), to verify message integrity and authenticity [50, 51]. The selection of encryption algorithms and hash functions depends on several factors, including the required security level, the computational capability of the Electronic Control Units (ECUs), and the available hardware resources [50, 51, 56]. Therefore, selecting cryptographic algorithms that provide strong security while maintaining computational efficiency is essential for modern automotive systems. Overall, encryption algorithms and cryptographic hash functions play a fundamental role in protecting in-vehicle communication systems against cyberattacks by ensuring the confidentiality, integrity, and authenticity of transmitted data.

It is important to choose algorithms and functions that provide strong security while still being efficient and practical for use in automotive systems. In summary, encryption algorithms and cryptographic hash functions play a fundamental role in securing automotive communication systems. These techniques help protect against cyberattacks while ensuring the confidentiality, integrity, and authenticity of transmitted data.

### 2.3.1 AES 128

The AES-128 encryption and decryption processes, including key expansion, the round transformations, and the corresponding decryption operations, are summarized below according to the AES specification [53] and illustrated in Figure 13.



**Figure 13** The basic AES-128 cryptographic architecture [60, 61]

- Key Expansion:

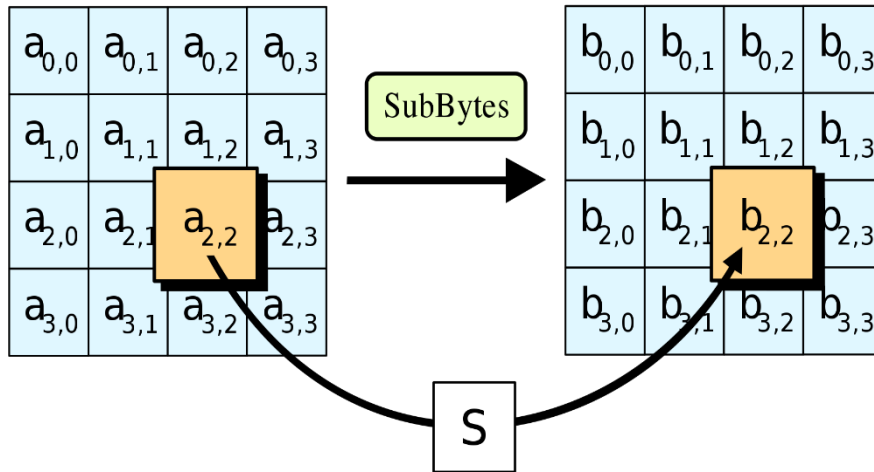
The original 128-bit key is expanded to generate a key schedule consisting of eleven 128-bit round keys. As specified in the AES standard (FIPS 197), each subsequent round key is derived from the previous one using a combination of byte rotation (RotWord), byte substitution (SubWord), round constants (Rcon), and XOR operations [53, 60].

- Encryption Process:

- The AES-128 encryption process consists of an initial AddRoundKey transformation followed by nine identical rounds and one final round [53, 60, 61].

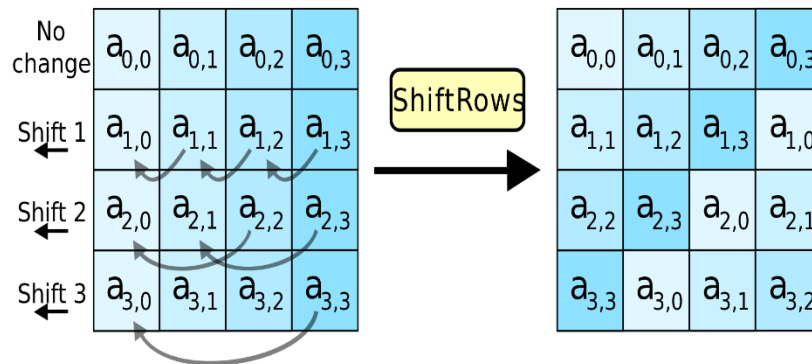
- The 128-bit plaintext is divided into 16 bytes and arranged in a 4x4 matrix.
- The initial round begins with an AddRoundKey operation, where the 16 bytes of the input are XORed with the first 16 bytes of the key schedule.
- The remaining 9 rounds consist of four operations as illustrated in Figures 14-17:

- SubBytes: Each byte of the input is substituted with a corresponding byte from a fixed S-box.



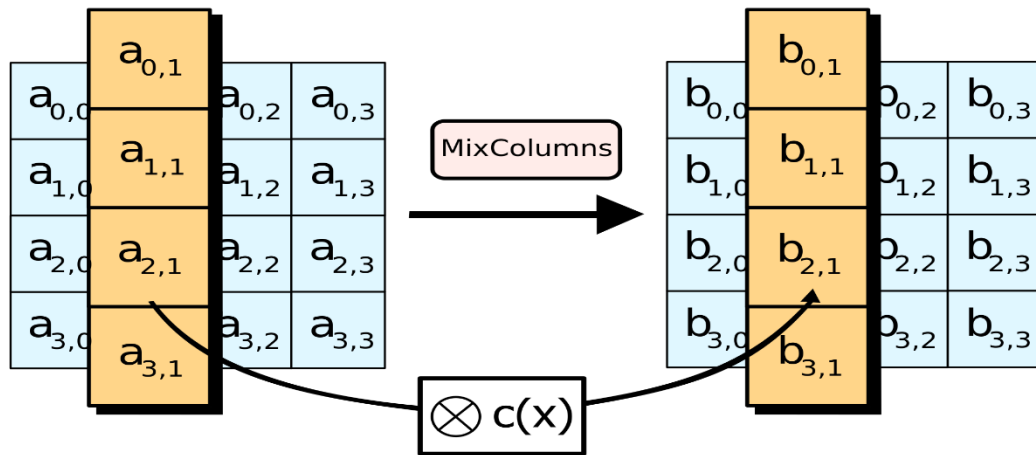
**Figure 14** AES 128 Sub-Bytes

- ShiftRows: The second row of the input is shifted left by one byte, the third row is shifted left by two bytes, and the fourth row is shifted left by three bytes.



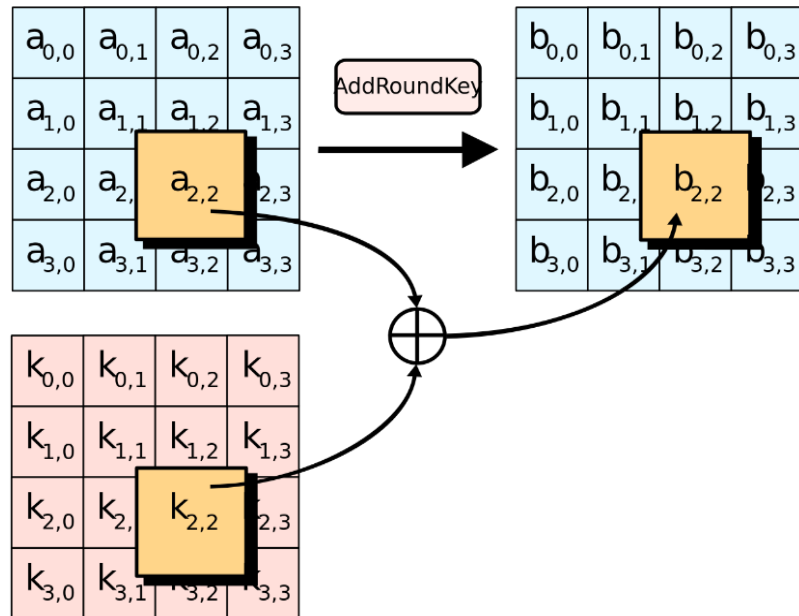
**Figure 15** AES 128 Shift Rows

- MixColumns: Each column of the input is multiplied by a fixed matrix in the Galois field.



**Figure 16** AES 128 Mix Column

- **AddRoundKey:** The 16 bytes of the input are XORed with the 16 bytes of the key schedule that correspond to the current round.



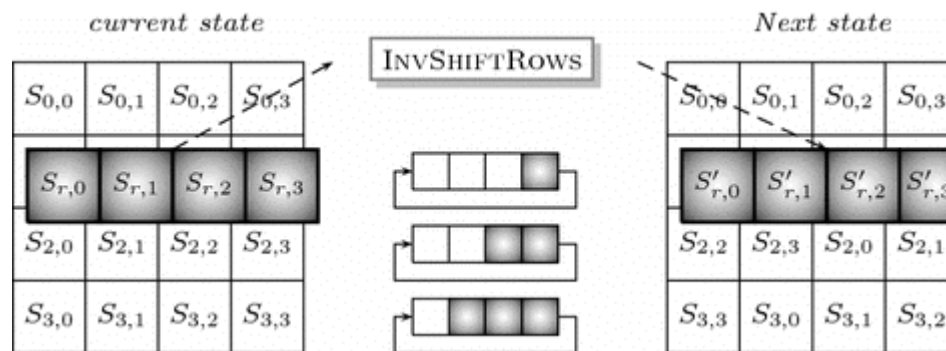
**Figure 17** AES 128 Add Round

- The final round omits the MixColumns operation.

- The resulting 16 bytes are the ciphertext.
- Decryption Process
 

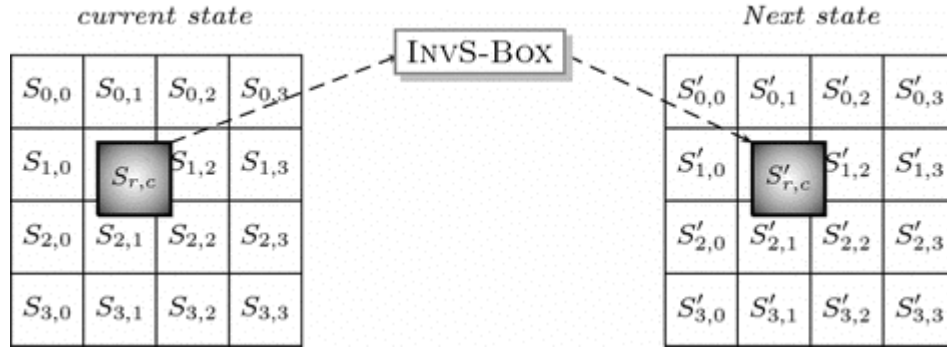
The AES-128 decryption process follows the inverse sequence of the encryption operations and reconstructs the original plaintext from the ciphertext [53, 60, 61].

  - The 128-bit ciphertext is divided into 16 bytes and arranged in a 4x4 matrix.
  - The initial round begins with an AddRoundKey operation, where the 16 bytes of the input are XORed with the last 16 bytes of the key schedule.
  - The remaining 9 rounds consist of four operations as shown in Figures 18, 19 and 20 in reverse order
    - InvShiftRows: The second row of the input is shifted right by one byte, the third row is shifted right by two bytes, and the fourth row is shifted right by three bytes.



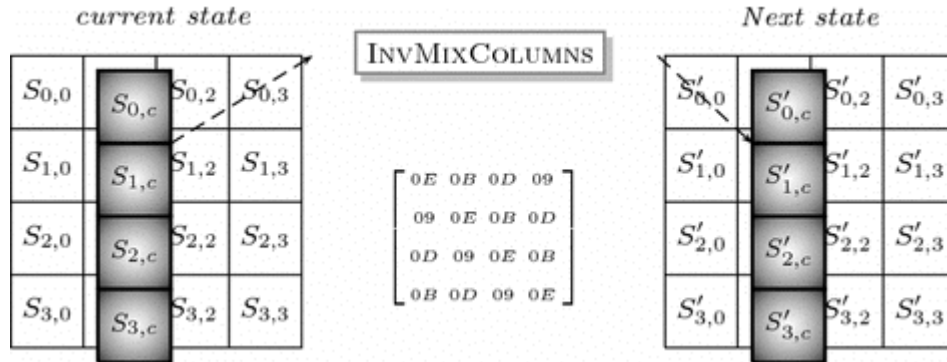
**Figure 18** AES 128 Decryption- Inverse Shift Row

- InvSubBytes: Each byte of the input is substituted with a corresponding byte from the inverse S-box.



**Figure 19** AES 128 Decryption- Inverse Sub Bytes

- InvMixColumns: Each column of the input is multiplied by a fixed matrix in the Galois field.



**Figure 20** AES 128 Decryption- Inverse Mix Column

- AddRoundKey: The 16 bytes of the input are XORed with the 16 bytes of the key schedule that correspond to the current round as shown in Figure 17

- The final round omits the InvMixColumns operation.
- The resulting 16 bytes are the plaintext. In both encryption and decryption, the input and output consist of 128-bit data blocks represented internally as a  $4 \times 4$  byte matrix, referred to as the State. Furthermore, the AES standard defines the S-box, inverse S-box, and the fixed Galois field matrices used in the MixColumns and InvMixColumns transformations, thereby ensuring deterministic and interoperable implementations across hardware and software platforms [53].

### 2.3.2 Message Authentication Codes (MACs)

A Message Authentication Code (MAC) is generated by applying a cryptographic algorithm together with a secret key to a message, producing a unique authentication value. The generated MAC is transmitted with the message and is used by the receiver to verify both the integrity of the message and the authenticity of its sender. Any modification to the message or the use of an incorrect secret key results in a different MAC value, allowing unauthorized changes to be detected [50, 51].

MAC algorithms are widely used in various applications, including network security, digital signatures, and electronic transactions. Common MAC algorithms include Hash-based Message Authentication Code (HMAC), Cipher-based Message Authentication Code (CMAC), and Galois Message Authentication Code (GMAC).

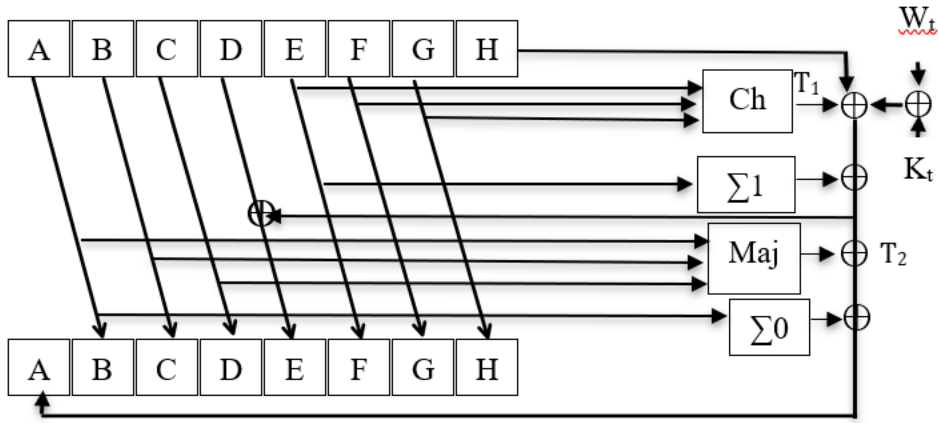
The Hash-based Message Authentication Code (HMAC) is one of the most widely used MAC algorithms. It combines a secret key with a cryptographic hash function to generate

a message authentication code for a given message. The resulting authentication tag is transmitted together with the message. Upon reception, the receiver recalculates the HMAC using the same secret key and compares it with the received value. If the two values match, the integrity and authenticity of the message are successfully verified [50, 51, 57]. The Cipher-based Message Authentication Code (CMAC) is another widely used MAC algorithm. It employs a symmetric block cipher, typically AES, together with a secret key to generate an authentication tag for a given message. The generated CMAC value is transmitted with the message, and the receiver independently recalculates the CMAC using the same secret key. If the computed and received CMAC values match, the integrity and authenticity of the message are successfully verified [50, 51, 53].

The Galois Message Authentication Code (GMAC) is a MAC algorithm derived from the Galois/Counter Mode (GCM) of operation for AES. It generates an authentication tag using AES together with Galois field multiplication, providing message integrity and authentication without encrypting the message payload. The receiver verifies the message by recomputing the GMAC using the same secret key and comparing it with the received authentication tag [50, 51].

### 2.3.3 SHA-256

The SHA-256 algorithm accepts an input message of arbitrary length and produces a fixed 256-bit (32-byte) message digest. The algorithm processes the input using a sequence of bitwise operations, modular arithmetic, and logical functions to generate a unique hash value [57]. A simplified overview of the SHA-256 algorithm is illustrated in Figure 21.

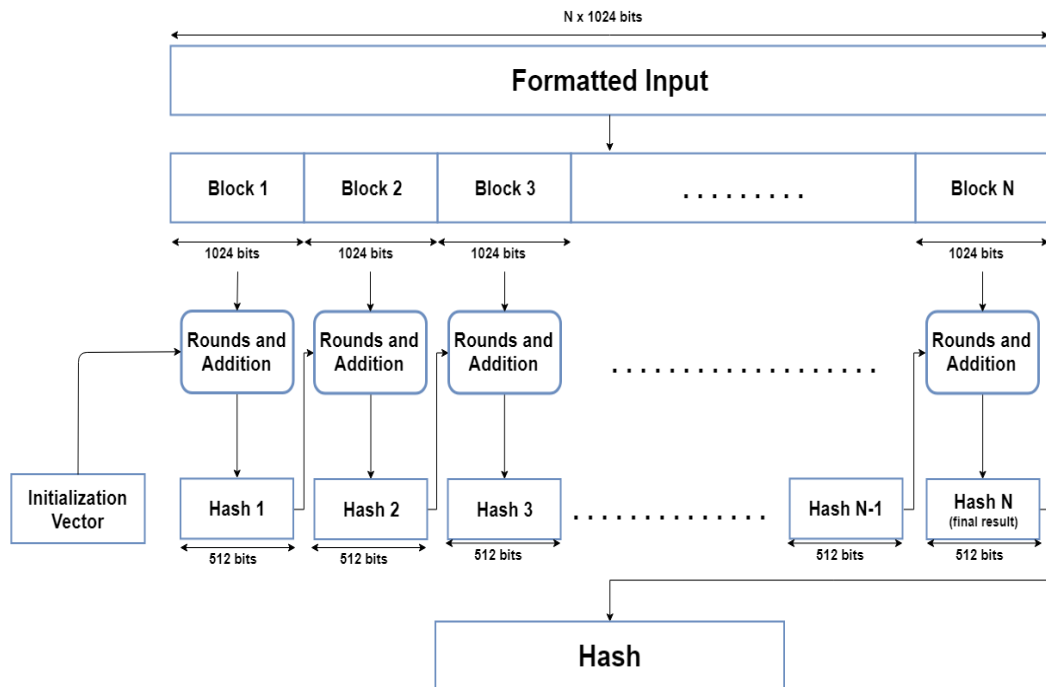


**Figure 21** Simplified architecture of the SHA-256 algorithm

- **Initialization:** Initialize eight 32-bit hash values and sixty-four predefined 32-bit round constants as specified in the Secure Hash Standard (SHS) [57].
- **Message Padding:** Append a single '1' bit to the message, followed by sufficient '0' bits so that the message length becomes congruent to 448 modulo 512. Finally, append the original message length as a 64-bit big-endian integer [57].
- **Message Processing:** The padded message is divided into 512-bit blocks. Each block is expanded into a 64-word message schedule and processed through 64 compression rounds using eight working variables, predefined constants, logical functions, and modular additions. After processing each block, the intermediate hash values are updated [57].
- **Hash Generation:** After all message blocks have been processed, the eight final 32-bit hash values are concatenated to produce the 256-bit message digest [57].

### 2.3.4 SHA-512

As illustrated in Figure 22, the SHA-512 algorithm follows the same overall design as SHA-256 but generates a 512-bit (64-byte) message digest. Unlike SHA-256, SHA-512 operates on 64-bit words instead of 32-bit words and processes each message block through 80 compression rounds using 80 predefined round constants. The larger word size and longer digest provide a higher theoretical security margin than SHA-256 while maintaining the same Merkle–Damgård construction defined in the Secure Hash Standard (SHS) [57].



**Figure 22** SHA 512 Architecture [57]

The following is a simplified description of the SHA-512 algorithm in pseudocode:

- **Initialization:** Initialize eight 64-bit hash values and eighty predefined 64-bit round constants as specified in the Secure Hash Standard (SHS) [57].
- **Message Padding:** Append a single '1' bit followed by sufficient '0' bits so that the message length becomes congruent to 896 modulo 1024. Finally, append the original message length as a 128-bit big-endian integer [57].
- **Message Processing:** The padded message is divided into 1024-bit blocks. Each block is expanded into an 80-word message schedule and processed through 80 compression rounds using eight working variables, predefined constants, logical functions, and modular additions. After processing each block, the intermediate hash values are updated [57].
- **Hash Generation:** After all message blocks have been processed, the eight final 64-bit hash values are concatenated to produce the 512-bit message digest [57].

## 2.4 FPGA-in-the-Loop (FIL) Methodology Using MATLAB

This section presents the FPGA-in-the-Loop (FIL) methodology implemented using MATLAB/Simulink and outlines the typical workflow used to validate FPGA-based hardware designs. FPGA-in-the-Loop verification enables co-simulation between high-level MATLAB/Simulink models and RTL implementations executing on FPGA hardware, thereby providing faster and more realistic verification than software-only simulation [63, 64]. The typical FIL implementation procedure consists of the following steps:

- **Hardware setup:** The first step is to set up the FPGA hardware and connect it to a host computer. This involves installing the necessary drivers, configuring the FPGA board, and connecting it to a host computer via a USB or Ethernet cable [64].
- **FPGA design:** Once the hardware is set up, you need to design the FPGA logic using a hardware description language such as VHDL or Verilog. This involves creating a digital circuit that implements the functionality you want to test or prototype.
- **HDL workflow:** MATLAB/Simulink provides a Hardware Description Language (HDL) Workflow Advisor that guides you through the process of generating HDL code from your Simulink model. The advisor automates the steps of the HDL code generation process by providing a sequence of preconfigured steps.
- **IP Core Generation:** Once you have a Simulink model that implements the desired functionality, you can generate an IP core from the model. An IP core is a block of HDL code that implements a specific function or algorithm.
- **Verification:** After the HDL code has been generated, you can simulate the design to ensure that it works as expected. This can be performed using HDL simulation tools such as ModelSim or Vivado Simulator before programming the FPGA device.
- **Programming the FPGA:** Once you have verified the HDL design, you can program the FPGA with the generated bitstream. This can be done using a programming tool provided by the FPGA vendor or by using MATLAB/Simulink to directly program the FPGA.
- **Co-Simulation:** Finally, MATLAB/Simulink communicates with the FPGA hardware in real time by streaming data between the Simulink model and the programmed

FPGA device. This involves configuring the Simulink model to communicate with the FPGA and streaming data between the two systems. In summary, FPGA-in-the-Loop implementation using MATLAB/Simulink consists of hardware setup, FPGA design, HDL code generation, verification, FPGA programming, and real-time co-simulation.

The primary advantage of FPGA-in-the-Loop validation is that it enables real-time verification of FPGA implementations while maintaining interaction with MATLAB/Simulink models. Compared with software-only simulation, FIL provides faster execution by performing computations directly on FPGA hardware while preserving co-simulation with high-level models. This approach reduces development time, improves validation accuracy, and allows hardware designs to be evaluated under realistic operating conditions before deployment [63, 64].

## CHAPTER THREE

### FPGA-BASED SECURITY ARCHITECTURES FOR CAN FD AND CAN XL

#### 3.1 Overview of the Proposed FPGA-Based Security Architectures

Building upon the background presented in Chapter 2, this chapter proposes FPGA-based security architectures for CAN FD and CAN XL communications. Classical CAN is also discussed to highlight its security limitations and the constraints imposed by its limited payload size, which restricts the implementation of comprehensive hardware-based security mechanisms. This chapter presents the proposed FPGA-based security protocols for CAN FD and CAN XL and discusses the corresponding hardware architectures, while using Classical CAN as a baseline for comparison.

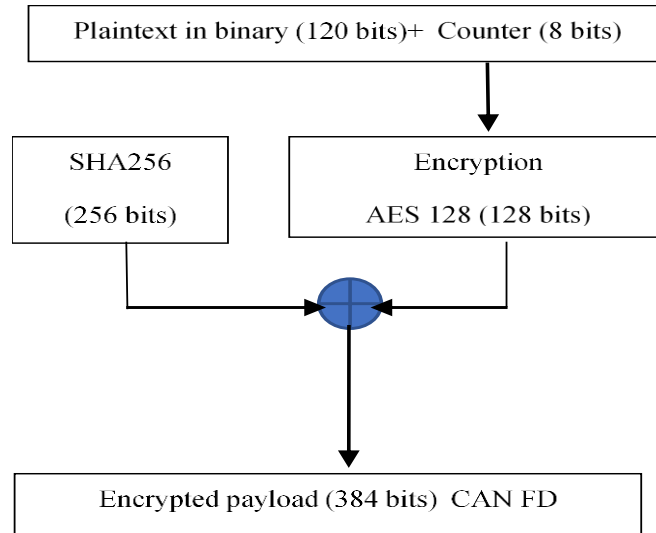
#### 3.2 Proposed FPGA-Based Security Architecture for CAN FD

##### 3.2.1 Proposed Hardware Design

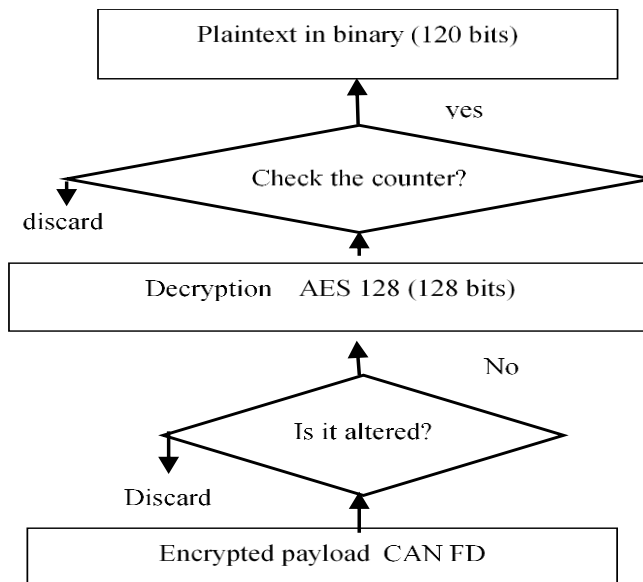
The proposed CAN FD security architecture employs AES-128 encryption, SHA-256 authentication, and a message counter to provide confidentiality, integrity, and replay attack protection. As illustrated in Figure 23, these components are integrated into a 384-bit security packet suitable for FPGA implementation. The transmitter and receiver architectures are presented in Figures 24 and 25, respectively. The transmitted security packet consists of a 256-bit SHA-256 authentication value and a 128-bit AES-128 encrypted block containing both the application data and the message counter. At the receiver, the received SHA-256 authentication value is compared with a locally computed hash. If both values match, the encrypted payload is forwarded to the AES-128 decryption stage for message recovery.



**Figure 23** Proposed FPGA-Based Security Architecture for CAN FD



**Figure 24** Proposed CAN FD Transmitter Architecture



**Figure 25** Proposed CAN FD Receiver Architecture

### 3.2.2 CAN FD Transmitter

- In accordance with the AES-128 specification discussed in Chapter 2, the AES-128 core encrypts the 120-bit application message together with the 8-bit message counter using the secret key, producing a 128-bit ciphertext block.
- The resulting ciphertext is subsequently processed by the SHA-256 module to generate a 256-bit authentication value. The SHA-256 output and the AES-128 encrypted payload are then combined to form the proposed 384-bit CAN FD security packet, which is transmitted over the CAN FD bus.

### 3.2.3 CAN FD Receiver

- The SHA-256 core [71] computes a local 256-bit authentication value from the received encrypted payload and compares it with the received SHA-256 value. If both values are identical, the integrity of the received encrypted payload is verified, and the payload is forwarded to the AES-128 decryption core. Otherwise, the received packet is discarded.
- The AES-128 decryption core decrypts the received ciphertext using the shared secret key to recover the original application message and the 8-bit message counter. The recovered message counter is then compared with the locally maintained counter value to determine whether the received message is fresh or represents a replay attack.

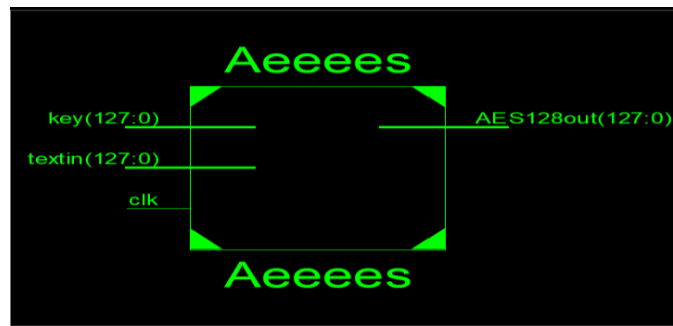
### 3.2.4 Simulation Results

This section presents the implementation and verification results of the proposed CAN FD security architecture using the Xilinx ISE 14.7 Design Suite. Representative test vectors

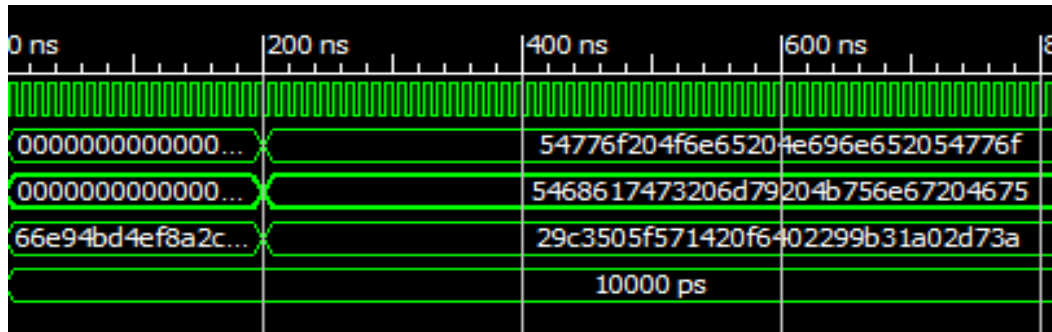
were applied using the Xilinx ISim simulator to verify the AES-128 encryption core, AES-128 decryption core, and SHA-256 core individually. Following the successful verification of the individual hardware cores, the complete CAN FD transmitter and receiver architectures were verified to confirm the correct operation of the proposed design.

#### 3.2.4.1 AES 128 Encryption Core Verification

The Register Transfer Level (RTL) architecture of the AES-128 encryption core is shown in Figure 26, while the corresponding simulation results are presented in Figure 27. Three representative test vectors were used to verify the functionality of the AES-128 encryption core. As an example, Figure 27 shows the encryption key as 5468617473206D79204B756E67204675 and the input plaintext as 54776F204F6E65204E696E652054776F. From top to bottom, the plaintext, the encryption key, and the resulting ciphertext are displayed in Figure 27. The simulated encryption core produces the ciphertext 29C3505F571420F6402299B31A02D73A, which is subsequently recovered and confirmed through the AES-128 decryption results. These simulation results confirm that the AES-128 encryption core produces the expected ciphertext and operates correctly.



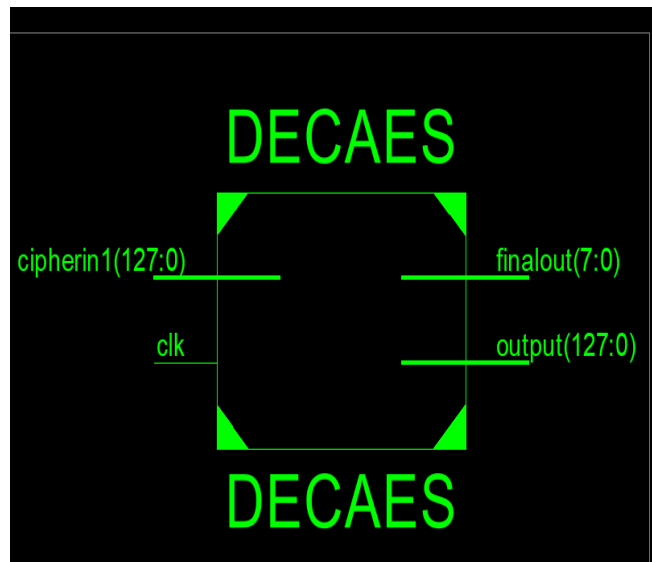
**Figure 26** AES 128 encryption core RTL



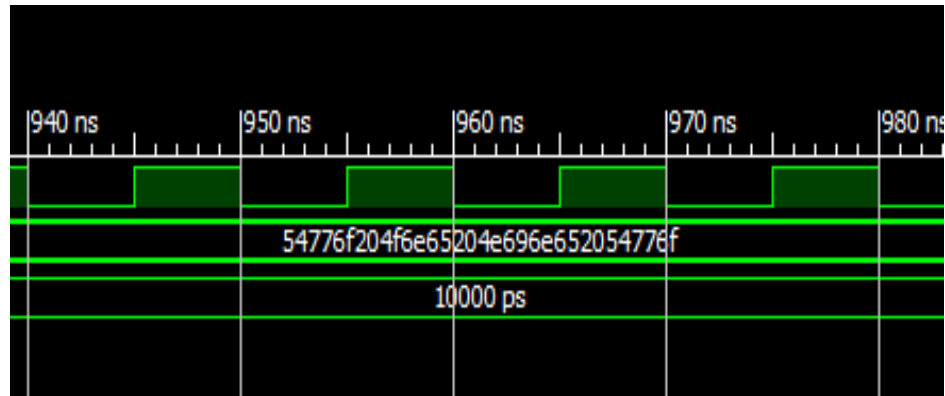
**Figure 27** AES128 encryption verification

### 3.2.4.2 AES 128 Decryption Core Verification

Figure 28 and Figure 29 depict the RTL of the AES 128 decryption core and the outcome of simulating that core respectively. The simulation results show that the decrypted output matches the original plaintext applied to the AES-128 encryption core, confirming the correct functionality of the AES-128 decryption core.



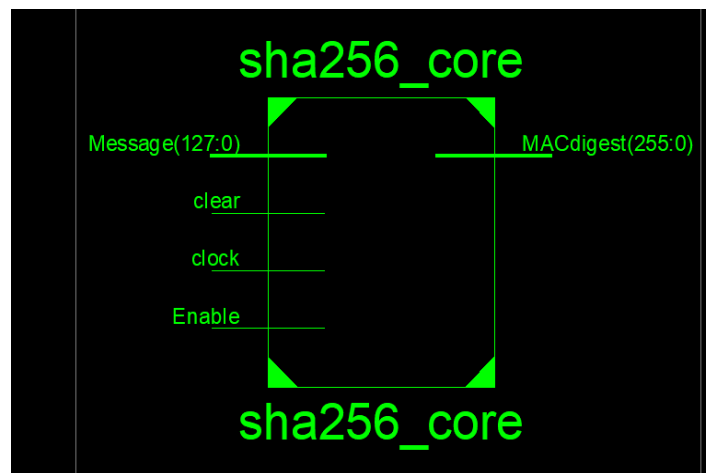
**Figure 28** AES 128 Decryption RTL



**Figure 29** AES 128 Decryption Verification

### 3.2.4.3 SHA 256 Core Verification

The 24 bits ASCII input "abc" (NIST test vector 24 bits input) is used to successfully verify the SHA256 Core [57]. The simulation results as given in Figure 30, which show the RTL of the SHA256 core, confirms the output 256'h ba7816bf 8f01cfea 414140de 5dae2223 b00361a3 96177a9c b410ff61 f20015ad. In addition to that, the SHA256 Core input was 128 bits, and its output was 256 bits.



**Figure 30** SHA 256 RTL

#### 3.2.4.4 The CAN FD Transmitter verification

The RTL implementation of the proposed CAN FD transmitter is shown in Figure 31, where the resulting output packet length is identified as 384 bits. The corresponding simulation results are presented in Figure 32 using the same AES-128 test vectors. For the SHA-256 core verification, “abdb” was used as a separate input, resulting in the 256-bit SHA-256 output of

A50C197C6D8BC491F3E88A8FF14F17DE409E10A43C182786A839C602C6392184.

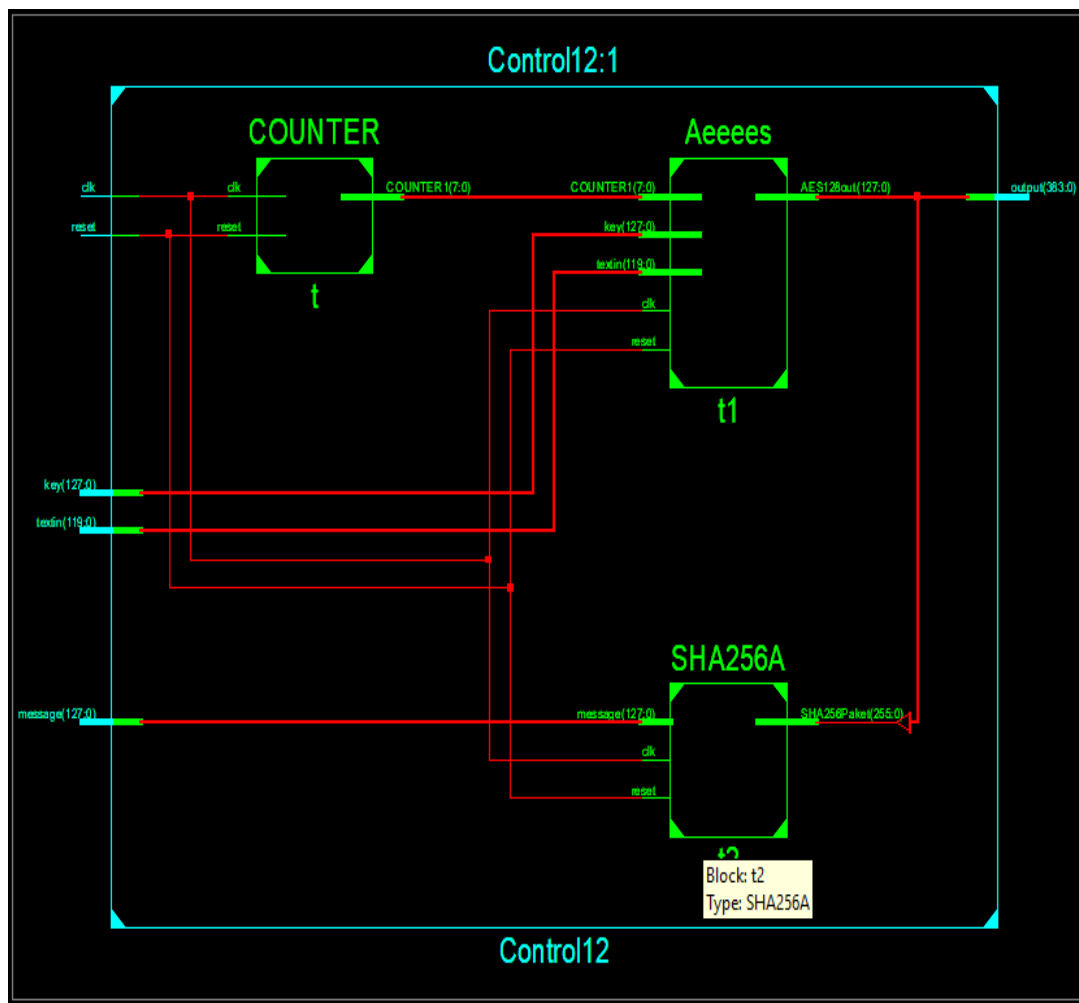


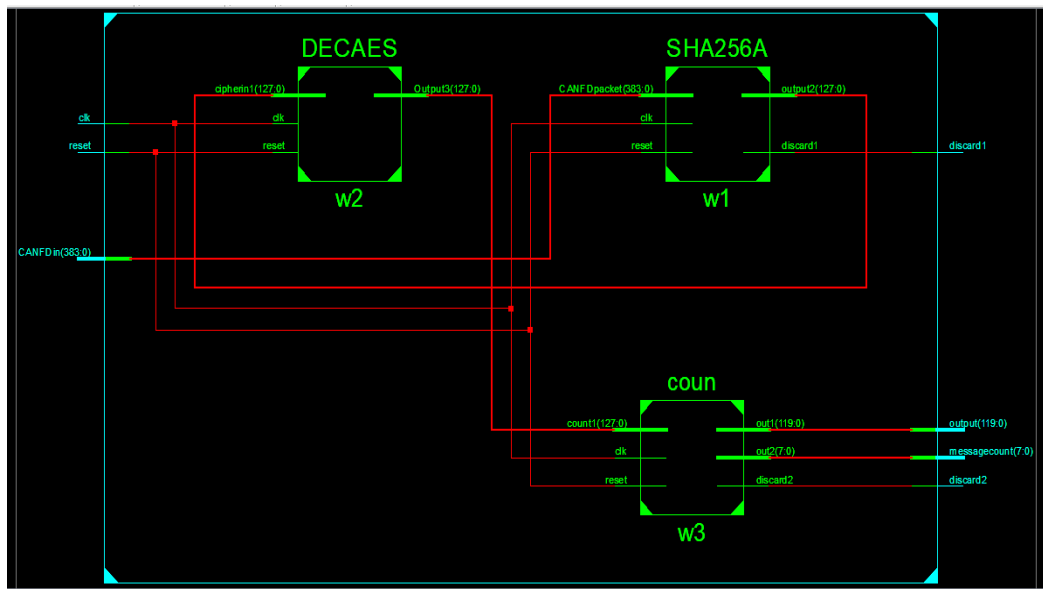
Figure 31 Proposed CAN FD Sender RTL

|                                                                                                  |  |                                  |  |  |
|--------------------------------------------------------------------------------------------------|--|----------------------------------|--|--|
|                                                                                                  |  | 54776f204f6e65204e696e65205477   |  |  |
|                                                                                                  |  | 5468617473206d79204b756e67204675 |  |  |
|                                                                                                  |  | abdb0000000000000000000000000000 |  |  |
| 87fcc48b8f178e78c776197e396a70b3a50c197c6d8bc491f3e88a8ff14f17de409e10a43c182786a839c602c6392184 |  |                                  |  |  |
|                                                                                                  |  | 30                               |  |  |

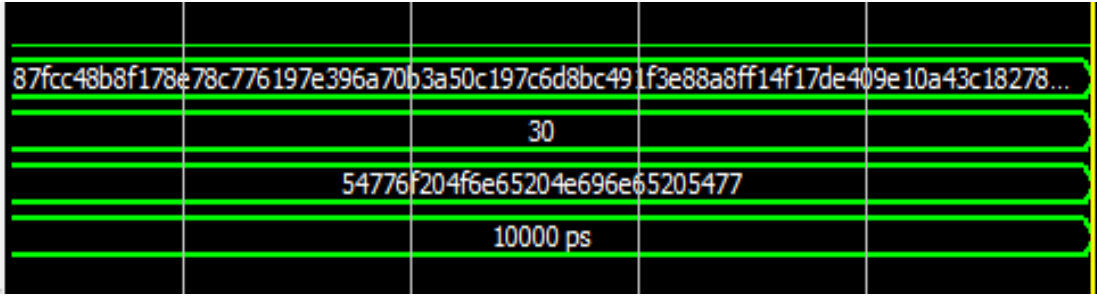
**Figure 32** Proposed CAN FD Sender Verification

#### 3.2.4.5 The CAN FD Receiver Verification

Figure 33 illustrates the RTL of the proposed CAN FD receiver, where the received SHA-256 authentication value is compared with the locally generated hash. If both hash values are identical, the encrypted payload is forwarded to the decryption stage; otherwise, the received packet is discarded and no further processing is performed. Figure 34 presents the successful verification of the proposed CAN FD receiver.



**Figure 33** CAN FD Receiver RTL



**Figure 34** CAN FD Receiver Verification

### 3.2.5 Performance Analysis

AES-128 is a symmetric-key encryption algorithm that provides confidentiality by encrypting data using a shared secret key between two communicating parties. When data is encrypted using AES-128, an attacker intercepting the ciphertext cannot interpret the original message without knowledge of the shared secret key [19, 22]. Therefore, AES-128 protects the confidentiality of the transmitted message. However, encryption alone does not prevent an attacker from retransmitting a previously captured packet.

SHA-256 is a cryptographic hash function that generates a fixed-size output from input data. The hash output represents the processed input, and any modification to the input data is expected to produce a different hash value. In the proposed architecture, the receiver computes a local SHA-256 hash from the received encrypted payload and compares it with the received hash value to detect message modification during transmission [57]. Therefore, SHA-256 provides an integrity verification mechanism within the proposed architecture.

Message counters are used to provide message freshness and protect against replay attacks by assigning a sequential counter value to each transmitted message. When the receiver receives a message, the recovered message counter is checked to determine

whether the message is fresh or has already been received. If the counter indicates that the message has previously been processed, the receiver discards the message. Therefore, the message counter provides the primary mechanism for detecting and preventing the acceptance of replayed packets.

### 3.2.6 Conclusion

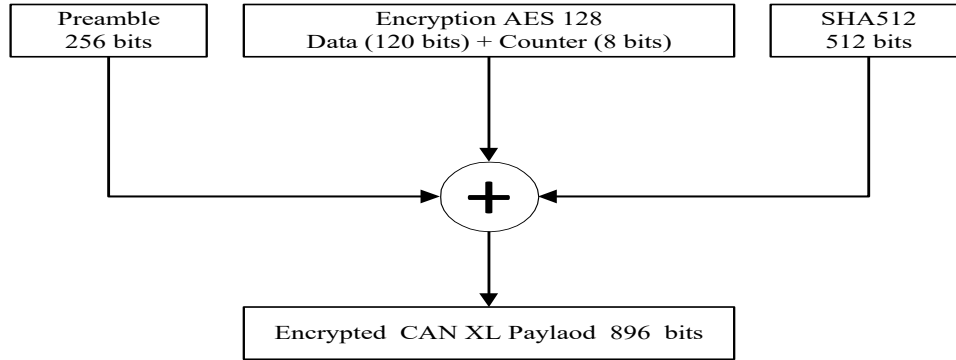
This section presented an FPGA-based hardware architecture designed to enhance the security of data transmitted over the CAN FD bus and protect against replay attacks. The proposed CAN FD transmitter and receiver cores, together with the SHA-256 and AES-128 cores, were implemented and verified on a Xilinx Virtex-7 FPGA. Replay attacks are mitigated using an 8-bit message counter, which introduces relatively low payload overhead while providing a mechanism for message freshness verification. The FPGA-based CAN FD design achieved a maximum clock frequency of 556 MHz and a throughput of 539 Mbps for the transmitter core, while the receiver core achieved a maximum clock frequency of 230 MHz and a throughput of 249 Mbps. The achieved throughput values were higher than those reported in [32]. The hardware implementation results demonstrate the feasibility of integrating the proposed CAN FD transmitter and receiver architecture into ECU-based communication systems to enhance data transmission security. The following section extends the proposed security design to CAN XL communication by considering an additional synchronization mechanism and employing SHA-512 instead of SHA-256.

### 3.3 Design of FPGA-Based Security Measures for CAN XL

This section describes the second proposed FPGA-based hardware design for real-time security implementation in CAN XL communication. The proposed architecture integrates a preamble sequence, SHA-512, and a message counter to enhance security against replay attacks [65].

#### 3.3.1 Proposed CAN XL Hardware Design

The proposed CAN XL hardware architecture consists of two major components: a CAN XL transmitter and a CAN XL receiver. Figure 35 illustrates the proposed architecture and its security mechanisms. The message counter length and AES-128 encryption mechanism remain unchanged from the previously proposed CAN FD architecture [65]. However, the CAN XL architecture introduces a preamble sequence and SHA-512 as additional security mechanisms.



**Figure 35** CAN XL Security Measures

The preamble sequence is processed using a cross-correlation technique to identify message delay, while the message counter is processed using SHA-512 to strengthen the freshness verification mechanism against replay attacks. The proposed CAN XL security packet has a total length of 896 bits, consisting of a 256-bit preamble sequence, a 128-bit

AES-128 encrypted block containing the 120-bit application message and 8-bit message counter, and a 512-bit SHA-512 output.

### 3.3.2 CAN XL Transmitter

After a communication session is established between the transmitter and receiver, the following steps are performed:

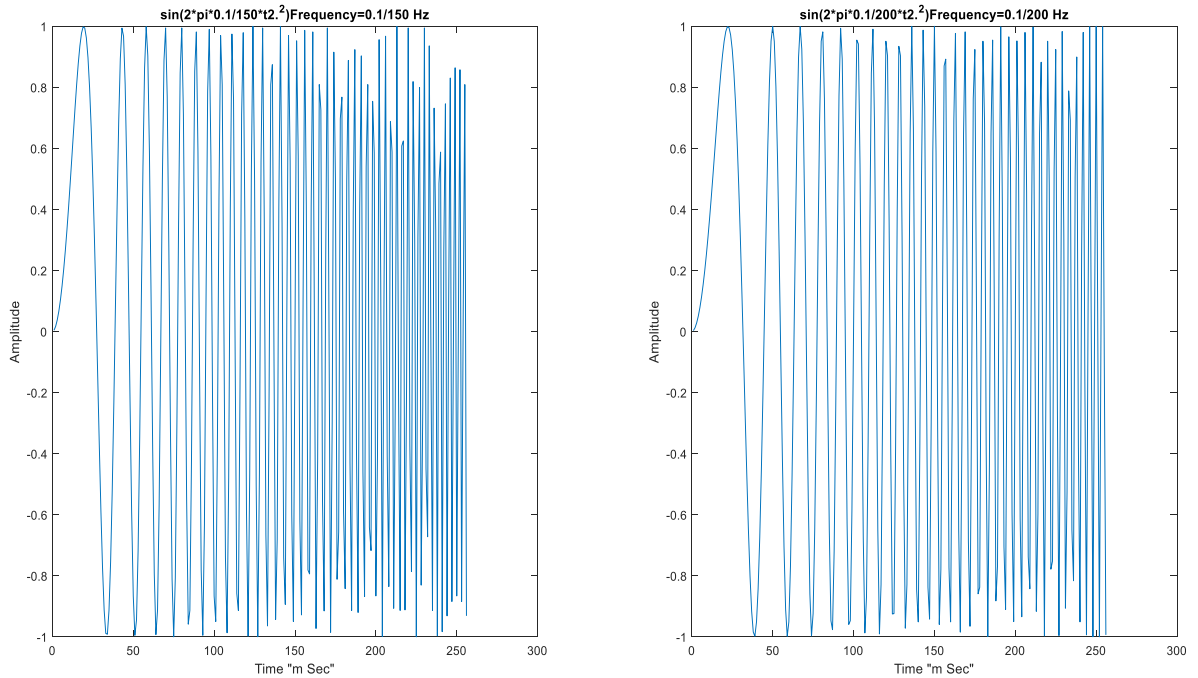
- The preamble sequence is generated and processed before being incorporated into the CAN XL transmitter packet. As proposed in the previous work [66], the sequence is generated using the output of a nonlinear function with increasing frequency. The resulting output is then processed using the normalization procedure described in [66], where each value is mapped to either 0 or 1 according to predefined criteria shared between the transmitter and receiver. The resulting binary preamble sequence is appended to the beginning of the proposed CAN XL packet. The resulting output is then processed using the normalization procedure described in [10], where each value is mapped to either 0 or 1 according to predefined criteria shared between the transmitter and receiver. The resulting binary preamble sequence is appended to the beginning of the proposed CAN XL packet.
- The 120-bit application message and the 8-bit message counter are combined to form a 128-bit input block, which is encrypted using a 128-bit secret key according to the AES-128 specification.
- The message counter is then processed using SHA-512 to generate a 512-bit hash output. The preamble sequence, AES-128 encrypted block, and SHA-512 output are

combined to form the proposed 896-bit CAN XL security packet. Finally, the complete packet is transmitted over the CAN XL communication channel.

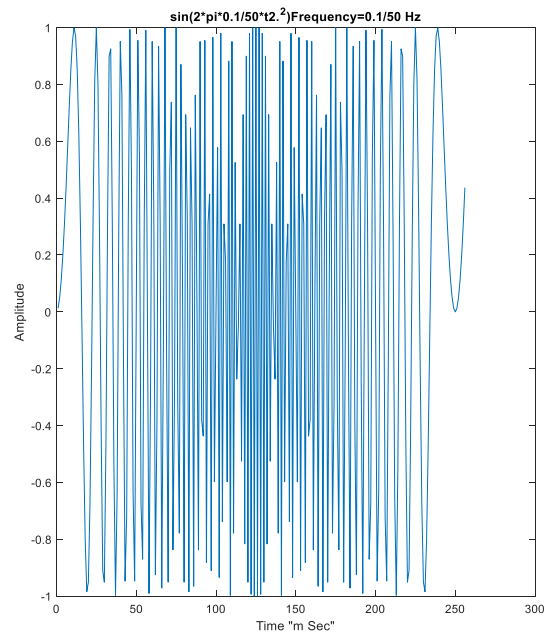
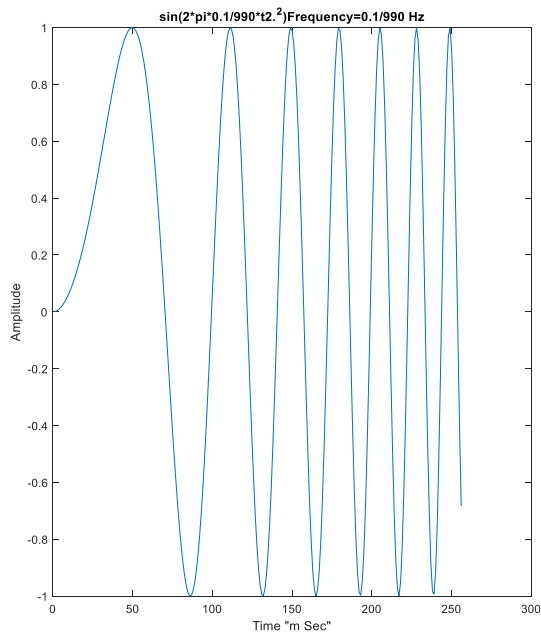
### 3.3.2.1 Preamble Sequence Generator

A preamble sequence is typically a known sequence of symbols transmitted at the beginning of a communication frame to support receiver synchronization and transmission detection. In this work, the preamble sequence is additionally used as a timing reference to estimate the delay of a received message. To generate the proposed preamble sequence, an increasing-frequency sine wave is employed, as expressed in Equation (1). The nonlinear output of this function varies with the selected frequency and time parameters. Examples of the generated signals at different frequencies are illustrated in Figures 36, 37, and 38.

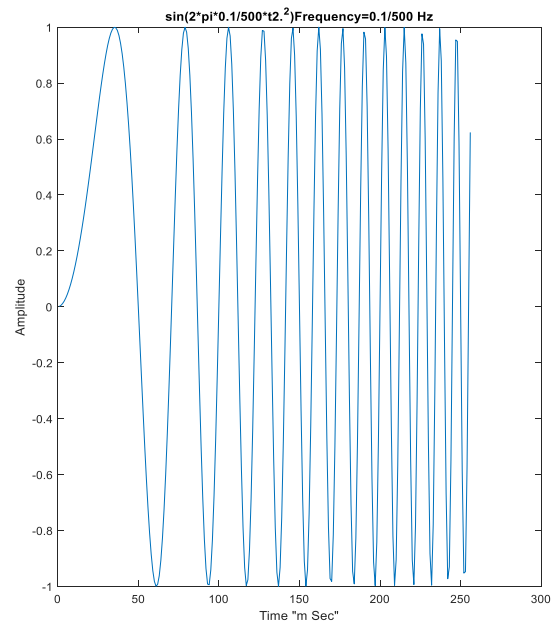
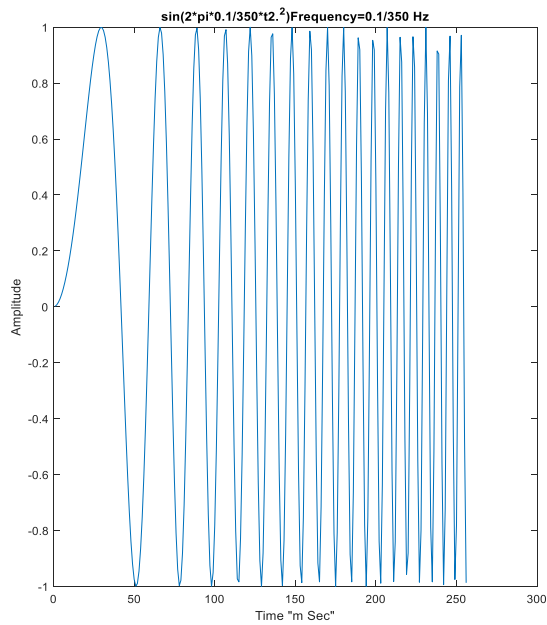
$$Y(t) = \sin(2 * \pi * \text{Frequency} * \text{time}^2) \quad (1)$$



**Figure 36** Non-Linear Function with different Frequencies

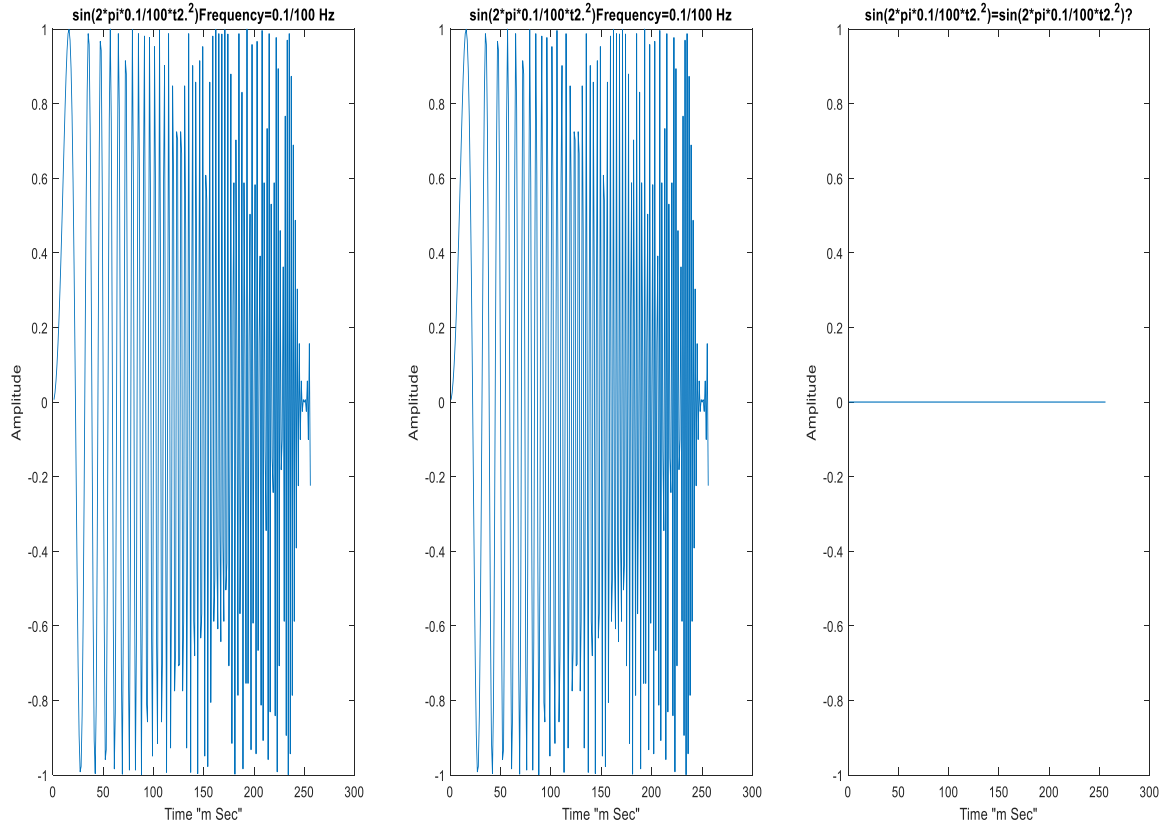


**Figure 37** Non\_linear Function with high and Low frequencies



**Figure 38** Non-Linear Functions with other frequencies

Figure 39 compares two signals generated using identical parameters ( $f=0.001$  Hz). The results show that the two generated signals are identical, resulting in a zero difference between their corresponding values.



**Figure 39** Two Identical linear functions and their difference

### 3.3.2.2. Normalization of the Nonlinear Function

The generated nonlinear signals are converted into binary sequences using a normalization process. In the proposed approach, the output values are mapped to either 0 or 1 according to a predefined thresholding criterion. The resulting binary values form the preamble sequences used by the transmitter and receiver. An example of the normalized sequences is presented in the following matrix

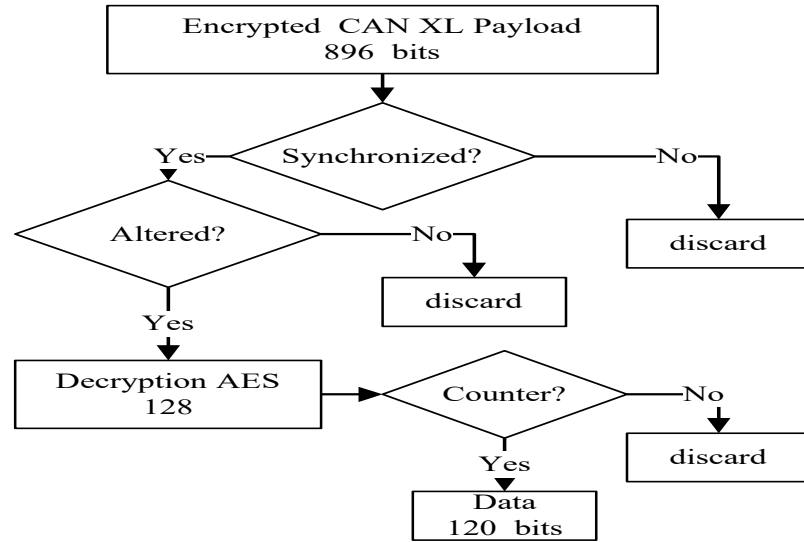
$$A = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

### 3.3.3 CAN XL Receiver

As shown in Figure 40, once the proposed CAN XL packet is received, message freshness is evaluated using two verification mechanisms. First, the received preamble sequence is compared with the locally generated preamble using cross-correlation. Second, the recovered message counter is compared with the locally maintained counter value.

- As presented in the previous work [66], the proposed cross-correlation function employed in this work evaluates delayed versions of the received preamble sequence. The number of delayed bits is used to estimate the message delay and determine whether the received packet remains within the predefined acceptable timing window. If the received preamble matches the locally generated preamble within the predefined timing window, the received message proceeds to the second verification stage. Otherwise, the packet is rejected and no further processing is performed.

- If the received preamble satisfies the cross-correlation timing criterion and the message counter verification is successful, the received packet is accepted for further processing. Otherwise, the packet is discarded and no further processing is performed.



**Figure 40** CAN XL Receiver

### 3.3.4 Simulation and Verification Results

The proposed CAN XL architecture was implemented and evaluated using the Xilinx ISE 14.7 Design Suite and MATLAB. MATLAB was used to generate the nonlinear signal sequence and process it to obtain the binary preamble sequence, reducing the complexity associated with directly generating the nonlinear signal in FPGA hardware. The CAN XL transmitter and receiver architectures were then implemented and simulated using the Xilinx ISE 14.7 Design Suite. ISim was used to apply representative test vectors and verify the individual hardware cores and the complete CAN XL architecture. The preamble verification core, SHA-512 core, CAN XL transmitter, and CAN XL receiver were evaluated independently to confirm their expected functionality.

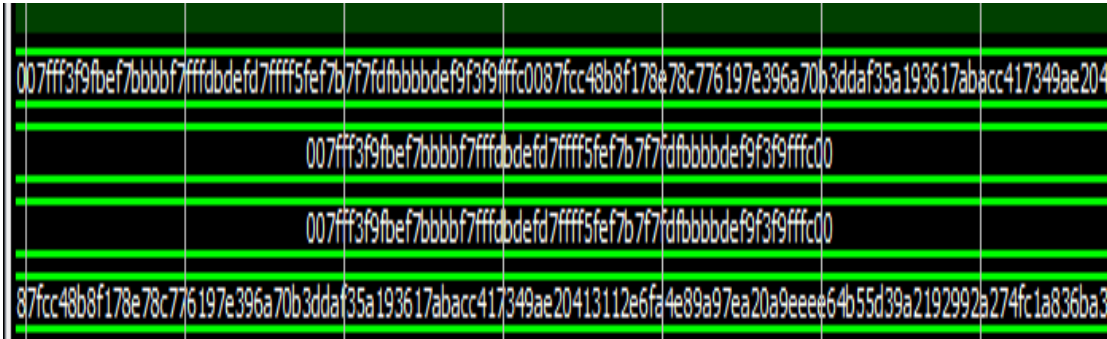
#### 3.3.4.1 Preamble Core Verification at the Receiver

Figures 41 and 42 present the verification results of the preamble core when the message is received without delay and with an introduced delay, respectively. The simulation vectors, presented from top to bottom, represent the received packet, the received preamble, the locally generated preamble, and the remaining message after removing the received preamble, respectively. For example, the first test vector represents the received CAN XL packet:

```
{007FFF3F9FBEF7BBBBF7FFFDDBDEFD7FFFF5FEF7B7F7FDFBBBBDEF9F3F9FFFc0087fcc48b8f178e78c776197e396a70b3ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f}. The third test vector represents the locally generated preamble:
```

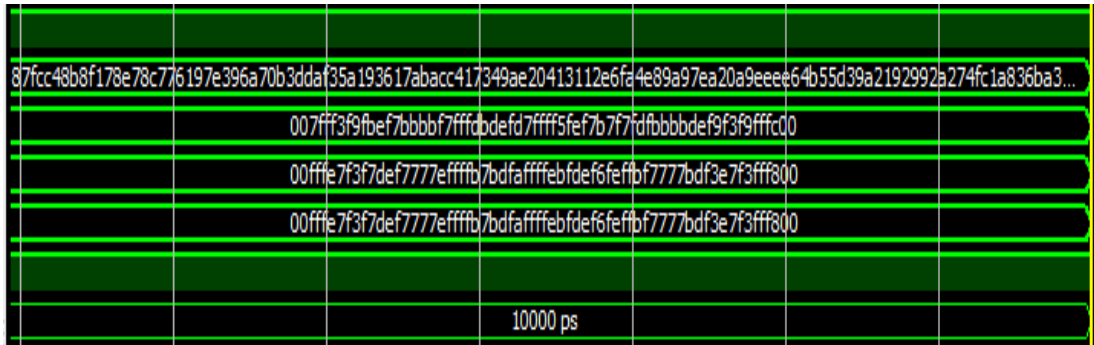
```
{007FFF3F9FBEF7BBBBF7FFFDDBDEFD7FFFF5FEF7BF7FDFBBBBDEF9F3F9FFFc00}, which matches the received preamble. Following preamble verification, the preamble core removes the preamble portion and forwards the remaining packet data for further processing, as represented by the output vector:
```

```
{87fcc48b8f178e78c776197e396a70b3ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f}. For verification purposes, a maximum delay of 9 bits was considered in the simulation to evaluate the operation of the preamble core under delayed-message conditions. The cross-correlation function was applied to delayed versions of the received preamble to estimate the introduced bit delay. Nine delayed preamble vectors were evaluated; four representative vectors are presented below:
```



**Figure 41** Preamble Verification at the CAN XL Receiver

- 1-bit delay preamble {00FFFE7F3F7DEF7777EFFFFB7BDFAFFFFEBFDEF6FEFF  
FBF7777BDF3E7F3FFF800}.
- 2-bit delay preamble {01FFFCFE7EFBDEEEEFDFFFF6F7BF5FFFD7FBDEDF  
DFF7EEEEF7BE7CFE7FFF000}.
- d. 9-bit delay preamble {FFFE7F3F7DEF7777EFFFFB7BDFAFFFFEBFDEF6FEFF  
BF7777BDF3E7F3FFF80000}.



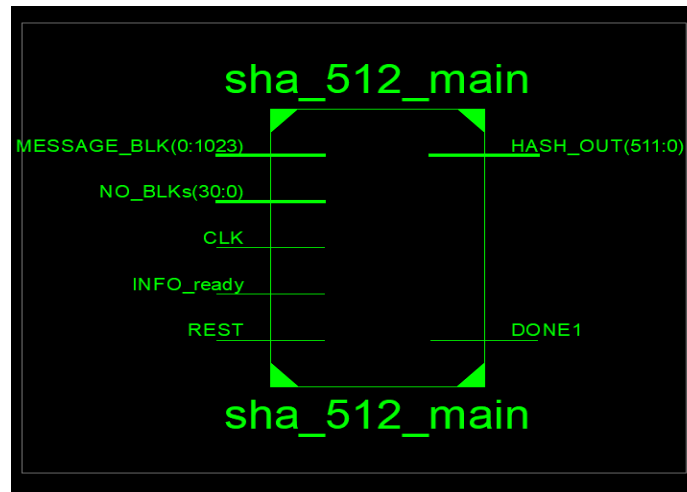
**Figure 42** Preamble Core Verification with a 1-Bit Delay

### 3.3.4.2 SHA-512 Core Verification

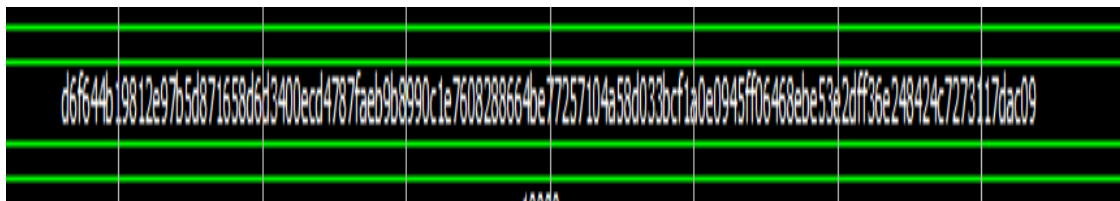
Figure 43 illustrates the RTL of the SHA-512 core, which was implemented based on the SHA-512 operations described in [57]. Figure 44 presents the corresponding

simulation results. The 24-bit ASCII input “aaa” was used as a representative test vector. The simulation produced the 512-bit output:

d6f644b19812e97b5d871658d6d3400ecd4787faeb9b8990c1e7608288664be77257104a58d033bcf1a0e0945ff06468ebe53e2dff36e248424c7273117dac09. The obtained output was compared with the SHA-512 digest generated using an independent online SHA-512 verification tool [65], and both results were identical. The matching outputs confirm the correct functionality of the implemented SHA-512 core.



**Figure 43** SHA 512 RTL

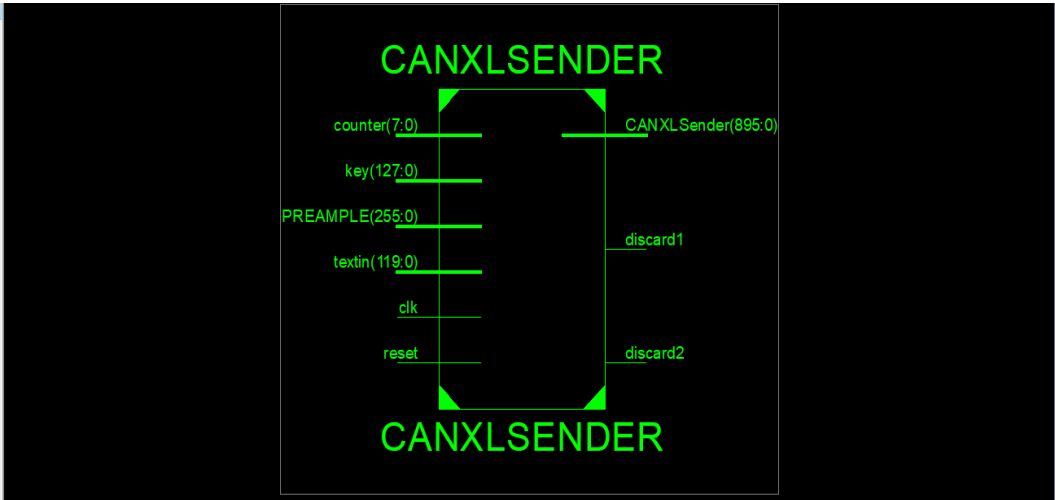


**Figure 44** SHA 512 Output

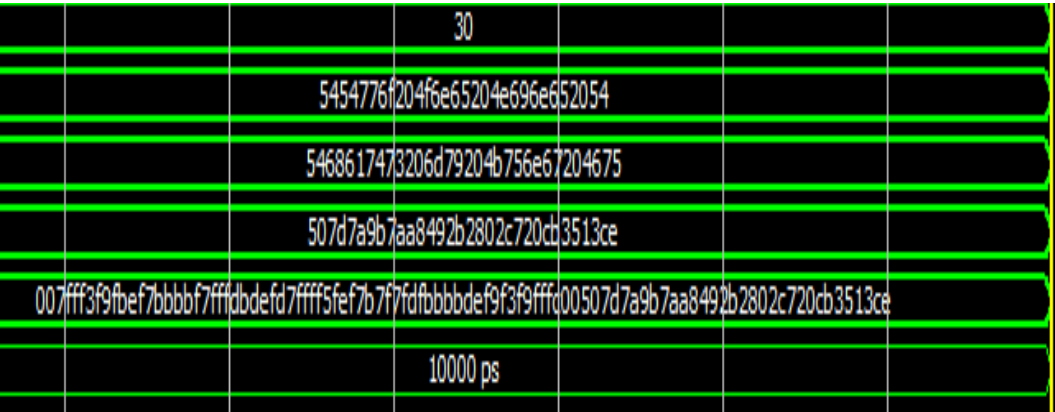
### 3.3.4.3 CAN XL Transmitter Verification

Figure 45 illustrates the RTL of the proposed CAN XL transmitter, while Figure 46 presents the corresponding simulation results. The CAN XL transmitter generates an

output packet containing the preamble sequence, the AES-128 encrypted block, and the SHA-512 output [65]. Due to the length of the complete 896-bit packet, it is challenging to display the entire output vector in a single simulation window. Therefore, only the least significant portion of the output vector is presented in Figure 46 for illustration purposes.



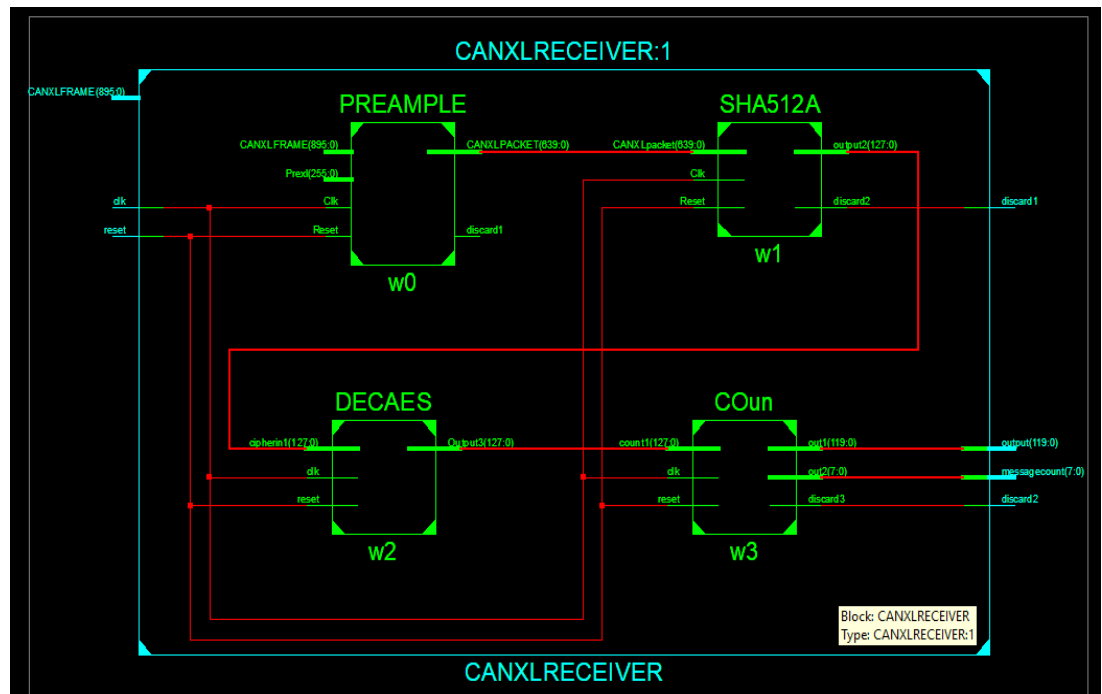
**Figure 45** CAN XL Transmitter RTL



**Figure 46** CAN XL Transmitter Verification

### 3.3.4.4 CAN XL Receiver Verification

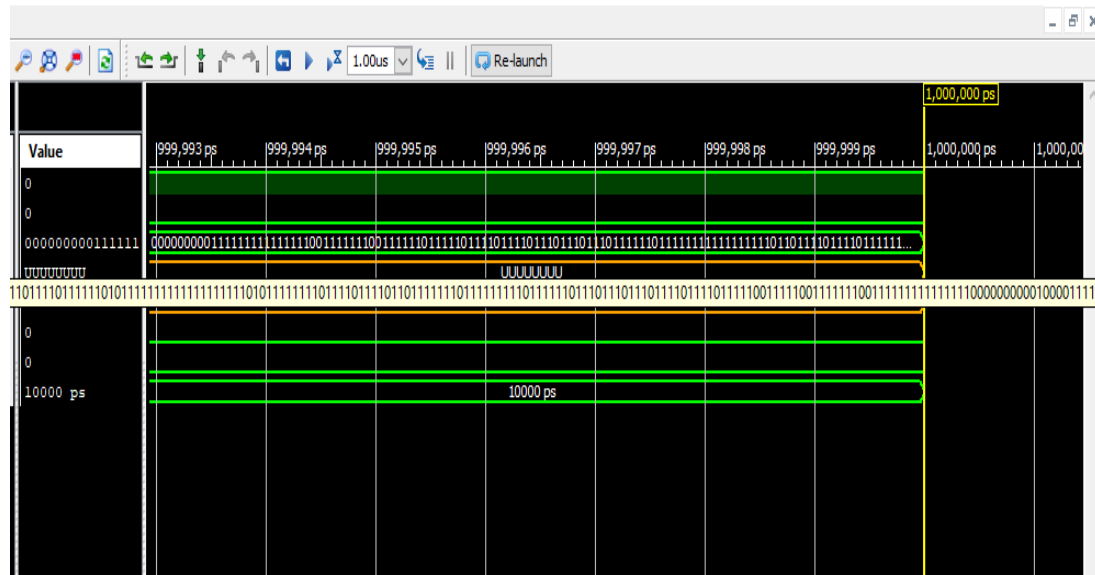
Figure 47 illustrates the RTL of the proposed CAN XL receiver. The receiver was evaluated under two message timing conditions: packets received without an introduced delay and packets received with a predefined bit delay. Several test vectors were applied to verify the complete receiver design. Figure 48 illustrates the operation of the proposed CAN XL receiver by presenting,



**Figure 47** CAN XL Receiver RTL

from top to bottom, the received CAN XL packet, the recovered message counter, and the recovered application message, respectively. If either the timing verification or the message counter verification fails, the received packet is discarded and no recovered

message is produced.



### Figure 48 CAN XL Receiver Verification

### 3.3.5 Performance Analysis

The implementation of preamble-based cross-correlation and SHA-512-based message counter verification on an FPGA platform provides additional mechanisms for detecting replayed messages. The cross-correlation mechanism evaluates bit-delayed versions of the received preamble to estimate the delay associated with a received packet. However, as the delay increases beyond the predefined evaluation range, reliable timing verification may no longer be achieved. Therefore, an appropriate timing threshold should be determined to distinguish acceptable communication delays from delays that may indicate a replayed or stale message. The preamble-based cross-correlation mechanism was selected to provide bit-level delay estimation and extend the freshness verification approach beyond the message counter mechanism employed in the previous CAN FD design. The current implementation was evaluated for delays of up to 9 bits. Although the

simulation results demonstrate the operation of the cross-correlation mechanism within this range, the optimal timing threshold was not determined in this study. Further investigation is required to establish an appropriate timing window under realistic CAN XL network conditions. The proposed architecture assumes secure storage of the cryptographic key and message counter state within the hardware platform. The security of key storage itself is outside the scope of this study and represents a limitation of the current implementation. The 8-bit message counter represents another limitation of the proposed architecture because it provides only 256 distinct counter values before counter rollover. Therefore, a counter rollover or reinitialization mechanism would be required for continuous long-term communication. The additional preamble processing, cross-correlation, SHA-512 computation, AES-128 decryption, and counter verification stages introduce processing overhead at the receiver. Although the FPGA architecture supports hardware-based and parallel processing, these additional security operations increase the complexity of the proposed design and should be considered when evaluating its real-time performance.

### 3.3.6 Conclusion

This work presented an FPGA-based hardware architecture designed to enhance replay attack protection for data transmitted over CAN XL communication. The proposed architecture extends the previous CAN FD design by introducing preamble-based cross-correlation to estimate received-message delays at the bit level. In addition to AES-128 encryption, the message counter is processed using SHA-512 as part of the proposed counter verification mechanism. As an extension of the previous CAN FD architecture,

the SHA-512 core and the proposed CAN XL receiver were implemented and verified on a Xilinx Virtex-7 FPGA. As the payload of our design was similar to our previous work, replay attacks can effectively be mitigated. The performance of CAN XL FPGA chip design was about 40 MHz maximum clock frequency( 199 Mbps throughput) for the CAN XL sender core, and 192 MHz maximum clock frequency (207 Mbps throughput). According to our hardware design results, CAN XL sender and receiver is a promising solution for securing data transmitted over ECUs (CAN XL). In the future work, increasing the message counter value and calculating how many bits delay as a threshold should be identified. Moreover, increasing the complexity level of CAN XL network should be implemented by three CAN XL nodes instead of two. Finally, securing the potential integration between the CAN XL and Ethernet should be investigated in depth.

## CHAPTER FOUR

### DESIGN EVOLUTION AND VALIDATION OF A NON-LINEAR FRESHNESS VALUE PROTOCOL: FROM SHA-256 ON VIRTEX-6 TO A LIGHTWEIGHT FPGA-IN-THE-LOOP IMPLEMENTATION ON ARTY A7

This chapter presents the MATLAB simulation and validation framework for the proposed CAN FD security protocol. Following a structured approach of threat modeling, protocol design, and protocol analysis, the research evolved from an initial SHA-256 implementation on Virtex-6 FPGA to an optimized lightweight protocol featuring non-linear Freshness Value generation, ultimately validated through FPGA-in-the-Loop testing on ARTY A7. The simulation process enabled thorough verification of security properties and performance characteristics before hardware implementation, reducing development risk and ensuring automotive-grade reliability.

#### 4.1 Threat Model

Our security protocol is designed to mitigate threats targeting message freshness, a critical vulnerability in automotive networks exploited by replay and spoofing attacks. Conventional methods, such as linear counters or timestamps, are inherently predictable and susceptible to manipulation, leading to synchronization loss and acceptance of replayed messages. These vulnerabilities are exacerbated in zonal E/E architectures where gateway processing introduces variable latency. The proposed threat model assumes a network-level adversary capable of:

- Eavesdropping on all CAN FD bus traffic.
- Injecting, modifying, and replaying arbitrary messages.
- Exploiting predictable freshness mechanisms to force desynchronization.

The protocol does not address physical key extraction, as key compromise is considered a separate security perimeter. Our primary security objectives are:

- **Replay Attack Resistance:** Ensure each message is processed only once, without relying on global time synchronization.
- **Spoofing Prevention:** Prevent message forgery by verifying authenticity and integrity.
- **Desynchronization Resilience:** Maintain Tx/Rx synchronization without creating denial-of-service (DoS) vulnerabilities.
- **Hardware Efficiency:** Achieve these goals with minimal computational overhead and memory footprint, suitable for resource-constrained ECUs., the protocol gains enhanced resistance against replay and synchronization attacks.

## 4.2 Protocol Security Design: An Evolutionary Approach

This section details the evolution of the security protocol, beginning with the initial comprehensive design and culminating in the optimized final implementation. This structured progression highlights the engineering decisions driven by practical constraints and performance analysis. The following architecture represents the initial design investigated during the early stages of this research. Several limitations identified during its evaluation motivated the development of the optimized protocol presented in Section 4.

### 4.2.1 Initial Protocol Architecture: SHA-256 based Authentication

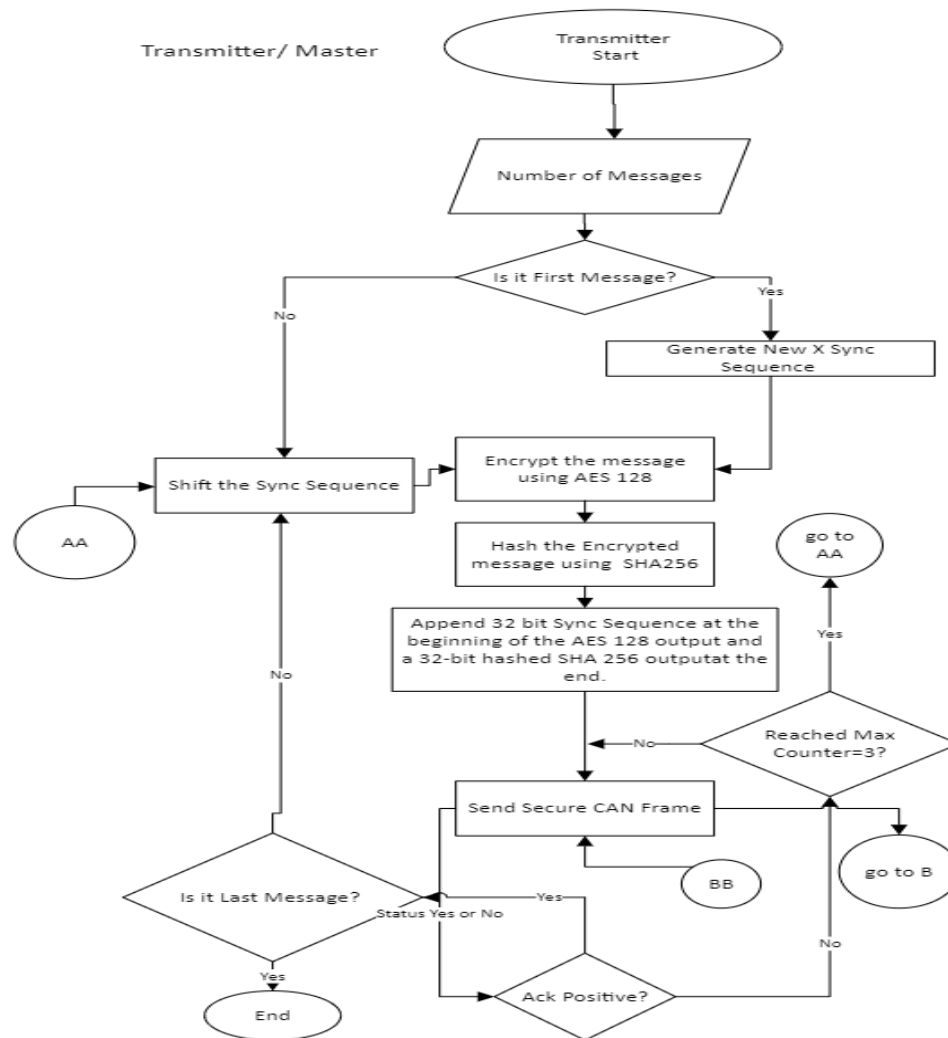
The foundational design for this research prioritized cryptographic robustness, employing a full SHA-256 hash to ensure maximum integrity. The transmitter plays a vital role in our security protocol as shown in flowchart in Figure 49. The system

employs AES-128 encryption to secure outgoing communications, so assuring confidentiality. Additionally, it adds a 32-bit synchronization sequence to further ensure message uniqueness, so mitigating the risk of replay attacks. The integrity verification process involves the computation of a 32-bit SHA-256 hash. Upon transmission of the 192-bit frame (sync sequence + AES 128+ SHA-256) illustrated in Figure 50, the transmitter awaits a positive acknowledgment (ACK) from the receiver before proceeding to increment and send the next message. In the event of a negative response, such as obtained from an outdated or manipulated sequence, the final message is transmitted with the identical sequence, therefore guaranteeing resilience against any assaults.

Nevertheless, the protocol employs a sequence generator that is 256 bits in length, which poses difficulties when the numbers of messages surpass the length of the sequence. Once the counter exceeds 256, it becomes necessary to employ an alternative mechanism for generating sequences. After attaining a number of 256, we can proceed to include an extra value (e.g., 20) into the sequence at both the transmitter and receiver, therefore guaranteeing their synchronization. This process may be iterated for future broadcasts. If the number of messages is fewer than the number of synchronizations sequences that are currently available, one possible approach would be to regularly update or regenerate the sequence using a predetermined process. The following are the steps for the transmitter:

#### 4.2.2 Initialize the synchronization sequence

Establish a 256-bit sequence as the initial reference, distinct for every communication session.



**Figure 49** Transmitter/Master Flowcharts

|                  |                    |             |
|------------------|--------------------|-------------|
| sequence 32 bits | AES 128 – 128 bits | SHA 256– 32 |
|------------------|--------------------|-------------|

**Figure 50** Secured CAN FD protocol

#### 4.2.3 Message Generation

Produce a message that aligns with the transmission requirements, such as speed.

#### 4.2.4 Message Encryption

Employ AES-128 encryption to enhance the confidentiality of the message.

#### 4.2.5 Compute Hash

Calculate a truncated 32-bit SHA-256 hash in order to confirm the integrity of the encrypted data.

#### 4.2.6 Sync Sequence Management

If the number of messages surpasses the specified length of the synchronization sequence, either: Reinitiate the sequence by introducing more entropy, or navigate through several sequences if the number of messages exceeds the length of the sequence.

#### 4.2.7 Transmit packet

Transmit the 192-bit packet inclusive of the synchronization sequence (32 bits), the message encrypted with AES (128 bits), and the SHA-256 hash (32 bits).

#### 4.2.8 Await acknowledgement

Upon receiving a positive acknowledgment, adjust the synchronization sequence by either one or two bits as previously negotiated.

To address negative acknowledgment (resulting from replay attack), retransmit the last message using the identical sequence up to three times.

Resend and Adjust: In the event of unsuccessful retries or synchronization problems, both the transmitter and receiver will change the synchronization sequence and try to establish connection using a revised sequence.

A crucial factor to consider is that any method depending on predictable shifts, particularly with a non-linear function, may result in weaknesses. The emergence of

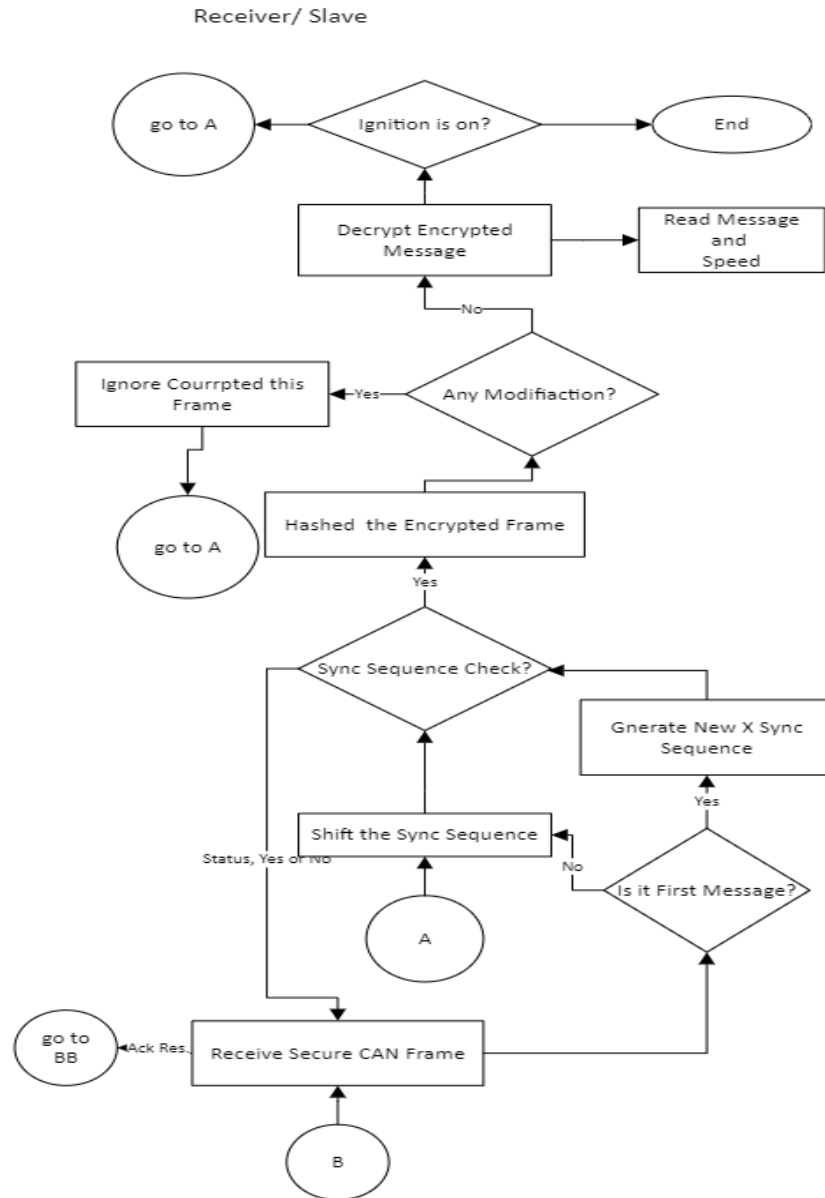
patterns could theoretically enable an attacker to forecast future sequences. Potential alternative approaches include expanding the sequence pool to include greater sizes (e.g., 512-bit or 1024-bit sequences) or increasing the frequency of dynamic sequence regeneration to avoid reuse or predictability. Moreover, use of a hybrid strategy that combines sequence numbers and timestamps could provide additional safeguard against overflow or misalignment.

Finally, it is crucial to maintain the unpredictability of sequence generation in order to prevent assaults that rely on sequence analysis. Conducting an analysis of any overflow or mismatch vulnerabilities will determine if modifications to the sequence length or regeneration mechanism are required.

Both message integrity and synchronization are the responsibility of the receiver. Upon receiving the 192-bit packet, the 32-bit synchronization sequence is initially confirmed to correspond with the sequence supplied by the transmitter. In order to prevent replay attacks, both parties modify their sequences, for example by moving one bit per transmission. Following sequence verification, the recipient decrypts the message using AES-128 and calculates a SHA-256 hash to guarantee the integrity of the message.

Upon detecting a duplicated or faked sequence, the receiver transmits a negative acknowledgment (NAK), which requests the transmitter to retransmit the most recent legitimate message up to three times. Upon failure to receive the accurate sequence, both the transmitter and receiver adjust their sequence by one bit and proceed with communication. This technique guarantees the currency and real-time synchronization of messages while providing protection against replay assaults. An illustrative flowchart of

the receiver's process as depicted in Figure 51 serves to demonstrate these stages, therefore guaranteeing the resilience and synchronization of the system in the face of any security risks.



**Figure 51** Receiver / Slave Flowchart

The protocol may become desynchronized if a message and its synchronization sequence are lost. Assuming the receiver is unable to acknowledge the message, the transmitter will either persist in waiting or make another effort to send the message until it receives an acknowledgment (within a specified number of retries, such as three attempts). In the event that these retries are unsuccessful, the transmitter and receiver will modify their synchronization sequence (e.g., shifting by one bit) and attempt to resume communication. This ensures that both parties remain in sync and mitigates desynchronization issues.

A variety of strategies can be implemented to mitigate potential vulnerabilities in the protocol and to confirm their absence, as evidenced by recent research:

Timing attacks refer to the exploitation of anticipated delays in the execution of synchronization or acknowledgment requests. For this reason, it is crucial to include unpredictability in the response time, therefore increasing the difficulty for attackers to anticipate and take advantage of any patterns. Comprehensive testing should incorporate fluctuations in network latency to guarantee the system's secure behavior across various timing scenarios. Empirical evidence indicates that temporal vulnerabilities can be identified by the simulation of various scenarios in which the network or system may induce delays.

Replay attacks: Despite the presence of synchronization sequences, hostile actors may nevertheless intercept and replay prior messages. In order to counteract this, the implementation of a freshness value that is specifically created for each message guarantees that the receiver will identify and flag any duplicate messages. Evaluation

should include intentional message delays to assess the system's ability to accurately identify and reject replayed packets. A research study highlighted the need of employing replay-resistant algorithms in client-to-client communication to effectively inhibit such attacks.

Desynchronization problems: Efficient synchronization is crucial in your protocol.

Discrepancies in sequence numbers between the transmitter and receiver have the potential to result in failure. The implementation of a resilient fallback mechanism, such as re-synchronization protocols, is necessary to effectively manage instances of lost or disordered packets. Validating this necessitates conducting network simulations in which packet loss or transmission delays are deliberately introduced to guarantee the system's ability to recover and resynchronize efficiently.

Sequence Overflow: Utilizing a 32-bit sequence offers sufficient capacity for the majority of applications. However, it is crucial to handle overflow accurately in order to prevent assaults that anticipate the subsequent sequence. Protocol testing should include scenarios in which the sequence exceeds its maximum capacity, guaranteeing that the resulting sequence values are both unexpected and securely managed after such overflow.

Faulty ACK Handling: The potential for denial-of-service (DoS) attacks is present when messages are continuously resent in response to negative acknowledgments.

Implementing a retry limit, similar to the three-attempt rule, and doing exhaustive testing under high-load scenarios is essential to prevent the system from unintentionally enabling attackers to deplete resources by initiating an infinite number of retries.

In the optimized implementation, the full SHA-256 tag was replaced with a 1-byte XOR tag, leveraging the high-entropy chaotic Freshness Value to maintain replay resistance while reducing hardware complexity. Building upon the limitations identified in the initial SHA-256-based design and the need for a more hardware-efficient and scalable solution, the final optimized protocol architecture is presented as follows. Although the initial protocol provided strong security, its dependence on ACK/NAK-based synchronization, retransmissions, and SHA-256 authentication introduced additional communication and hardware overhead. These limitations motivated the development of the optimized protocol described in the following section.

#### 4.3 Proposed Protocol Architecture Overview

The proposed protocol introduces a multi-layered security architecture centered around non-linear Freshness Value (FV) generation. This design addresses the limitations of conventional counter-based approaches by combining unpredictability, lightweight authentication, and hardware-efficient implementation.

##### 4.3.1 Non-linear Freshness Value Generation

A quadratic chirp function is used to generate deterministic yet unpredictable sequences. The output is quantized into a balanced binary stream and used to construct a 256-bit entropy reservoir, from which a 32-bit freshness value (FV) is extracted. This enables implicit synchronization between the transmitter (Tx) and receiver (Rx) without requiring explicit FV transmission, thereby reducing communication overhead.

#### 4.3.2 Chaotic Freshness Value Enhancement

To further improve entropy and resistance to prediction attacks, multiple trigonometric functions are combined with Add-Rotate-XOR (ARX) operations and a whitening stage using a linear congruential generator. This results in a highly unpredictable FV sequence suitable [82] for secure automotive communication. To enhance entropy, three non-linear trigonometric bases are defined as:

$$base_1(t) = \sin\left(2 \times \pi \times \left(\frac{f}{100}\right) \times t^2\right) \quad (2)$$

$$base_2(t) = \sin\left(2 \times \pi \times \left(\frac{1.7 * f}{100}\right) \times (t)^{1.5}\right) \quad (3)$$

$$base_3(t) = \sin\left(2 \times \pi \times \left(\frac{0.3 * f}{100}\right) \times (t)^{2.3}\right) \quad (4)$$

Where:  $t$  = frame counter  $\in \mathbb{N}$ ,  $f = 0.1 + (\text{ecu\_id mod } 97)/1000$ .

The parameter  $f$  is derived from the  $\text{ecu\_id}$ , ensuring that the chaotic behavior is personalized for each device or entity. The use of non-integer exponents (1.5, 2.3) and different frequency multipliers (1.7, 0.3) creates complex, non-harmonic, and aperiodic waveforms for different inputs, enhancing the entropy.

Binary quantization via asymmetric thresholds :

$$\begin{aligned} Chaotic_{mix}(t) = & \mathbb{I}[base_1(t) \geq 0.25 \vee base_1(t) \leq -0.35] + \mathbb{I}[base_2(t) \geq \\ & 0.15 \vee base_2(t) \leq -0.25] - \mathbb{I}[base_3(t) \geq 0.35 \vee base_3(t) \leq -0.15] \end{aligned} \quad (5)$$

Where the logical operator  $\vee$  indicates to that Output equals to 1 if any of the specified conditions are satisfied such as  $base_1(t) \geq 0.25$  or  $base_1(t) \leq -0.35$ ; otherwise, output equals to 0

and so on. Furthermore,  $\mathbb{I}[\cdot]$  denotes the indicator function, which equals 1 when the enclosed condition is true and 0 otherwise.

$$Chaotic_{binary}(t) = Chaotic_{mix}(t) \bmod 2 \{0,1\} \quad (6)$$

To enhance State Calculation (32-bit, masked) : This part creates a secure pseudo-random state by mixing the secret key with the counter

Let

$$\text{secret key} \in \{0..2^{32-1}\},$$

$$\text{constant} = 2654435761 .$$

also, Add-Rotate-XOR All operations are on 32-bit words with wraparound (mod  $2^{32}$ ).

$$state_0 = (\text{secret key} \oplus ((t \times \text{constant})) \bmod 2^{32} \quad (7)$$

$$state_1 = state_0 \oplus (state_0 \gg 17) \quad (8)$$

$$state_2 = state_1 \oplus (state_1 \ll 13) \bmod 2^{32} \quad (9)$$

$$state_3 = state_2 \oplus (state_2 \gg 19) \quad (10)$$

$$\text{enhanced state}(t) = state_3 \quad (11)$$

To obtain multi-stage bit manipulation, ARX cascade is implemented as seen in the next

Equations:

$$SS = \text{enhanced state}(t) \quad (12)$$

$$\text{shifts} = [7,19,11,23,5,29,13,31] \quad (13)$$

For  $k=1,2,3,4,5,6,7$  and 8

$$\text{shift}_1 = SS \gg \text{shifts}(k) \quad (14)$$

$$shift_2 = (SS \ll shifts(k \bmod 7) + 1) \bmod^{32} \quad (15)$$

$$SS = SS \oplus shift_1 \oplus shift_2 \quad (16)$$

$$\text{Result: } SS = SS \oplus (SS \gg 16) \quad (17)$$

At this stage is the fusion point where the cryptographic state, the frame counter, and the chaotic bit are combined at the byte level before a final whitening step.

The 32-bit cryptographic state  $SS$  is decomposed into four bytes in big-endian order. Simultaneously, four bytes are extracted from the counter  $t$ , and the single chaotic bit is replicated to influence each byte position

$$Encrypted_{byte}(i) = (SS \gg 8 \times (4 - i)) \text{ AND } 0xFF \quad (18)$$

$$Counter_{influence}(i) = (t \gg 8 \times (i - 1)) \text{ AND } 0xFF \quad (19)$$

$$Chaotic_{influence}(i) = Chaotic_{binary}(t) \ll (i - 1) \quad (20)$$

Note the different shift directions for  $SS$  and  $t$ ; this intentional mismatch in endianness adds an extra layer of complexity to the mixing process. The three influences are combined for each of the four bytes using the XOR operation depicted in Equation 21

$$mixed_{byte}(i) = Encrypted_{byte}(i) \oplus Counter_{influence}(i) \oplus Chaotic_{influence}(i) \quad (21)$$

This step ensures that each byte of the final output is a function of all three primary inputs: the secret key (via  $SS$ ), the public counter, and the device-specific chaotic signal.

To mask any potential residual patterns from the mixing stage, a final, fast whitening step is applied using a Linear Congruential Generator (LCG) with known parameters from Numerical Recipes:

$$Whitening_{seed}(t) = (1664525 \times t + 1013904223) \bmod^{32} \quad (22)$$

For  $i = 1..4$ :

$$Whitening_{byte}(i) = (Whitening_{seed}(t) \gg 8 \times (i - 1) \text{AND } 0xFF) \quad (23)$$

$$Final_{byte}(i) = (Mixed_{byte}(i) \oplus Whitening_{byte}(i)) \quad (24)$$

The four whitened bytes are assembled into the final 32-bit output value in big-endian order. To formulate final FV<sub>32 bit</sub> equation is illustrated in Equation 25

$$FV(t) = \sum_{\{i=1..4\}} final\_byte(i) \times 256^{(4-i)} \quad (25)$$

The final value,  $FV(t)$ , represents a multi-layered combination of chaotic behavior, cryptographic transformation, and deterministic whitening, ensuring high entropy, unpredictability, and robustness against replay and synchronization attacks. It is a complex amalgamation of cryptographic strength, chaotic perturbation, and deterministic whitening, designed to be unique and non-predictable for each input. The multi-layered approach ensures that even if one component is compromised, the overall system security may remain intact due to the interdependent nature of the various stages.

#### 4.3.3 Lightweight Authentication Tag

A lightweight XOR-based authentication tag is constructed using FV bytes, AES-128 ciphertext, and key expansion values. This design significantly reduces computational

overhead while maintaining strong data integrity guarantees, making it suitable for FPGA-based implementation.

#### 4.3.4 Protocol Operation

The transmitter generates the freshness value and authentication tag, constructs the secure message, and transmits it. The receiver independently regenerates the FV and validates both freshness and integrity without requiring explicit FV transmission.

Resynchronization mechanisms ensure robustness under message loss, replay attempts, and adversarial conditions while maintaining deterministic synchronization between Tx and Rx.

### 4.4 Freshness Value ( FV ) Transmitter and Receiver Validation

To assure message authenticity and prevent replay attacks, it is necessary to ensure that the Freshness Value is synchronized between the transmitter and receiver in our protocol. A systematic approach to incorporate validation into our design is described as follows:

#### 4.4.1 Generation and Transmission of Freshness Value

On both the transmitter and receiver side, the Freshness Value is produced using a non-linear function that can be shifted with each transmission. The Freshness Value is conveyed alongside the encrypted message (e.g., AES-128 encrypted message with SHA-256 hash).

Eliminating the predictability of linear counters involves selecting a non-linear function for FV generation. Equation 1 defines the nonlinear function we use in this work, which offers two crucial characteristics for secure communication. The

function is deterministic and reproducible, which allows both the transmitter and the receiver to independently generate identical value sequences when initialized with the same parameters (e.g., initial time  $t$ , frequency  $f$ ). This allows for synchronization without explicitly transmitting the FV. Secondly, it adds unpredictability: the output appears stochastic and unpredictable over brief intervals to an outside observer who is unaware of the initial parameters, providing defence against replay and pre-computation attacks. A 32-byte sequence produced by a non-linear function is known as Equation 1.

That method is inspired by techniques in digital communication systems, where non-linear function (called Chirp) are employed for frame synchronization due to their favorable autocorrelation properties. In this work, we adapt the principle for security purposes, using the chirp's output as an entropy source rather than for signal detection.

Where  $r(t)$  is an output quantized into 256 normalized values and mapped to a binary stream (1/0) to fill a 32-byte vector. In other words, to generate a cryptographically strong FV, the output of the non-linear function is passed through a sign-based quantizer. This method produces a perfectly balanced bitstream, maximizing entropy and ensuring that each new value is effectively unpredictable [82]. This transforms the analogue non-linear function into a digital values suitable for securing automotive communication.

For message  $i$ , only the lowest least significant 4 bytes of FV are used.

FV is bit-shifted by one position after every successful transmission or , after one failed attempt followed by a success (or after two consecutive failures).

In order to show complete mathematical model, the Freshness Value (FV) generation is a discrete-time quadratic chirp. Therefore, let rewrite Equation 1 as follows:

$$S[n] = \sin(2 \times \pi \times f \times (nTs)^2) \quad (26)$$

Where :

- $f \in \mathbb{Z}^+$  be the fundamental frequency, a positive integer derived from a secret key.
- $T_s$  be the sampling period
- $n \in \mathbb{Z} (\geq 0)$  be the discrete-time index, shifted (incremented) with each FV generation attempt. It is synchronized between the transmitter (Tx) and receiver (Rx).

A signum function quantizes the analogue chirp signal into a perfectly balanced binary stream

$$b[n] = \{1 \text{ if } s[n] \geq 0; 0 \text{ otherwise}\} \quad (27)$$

A 256-bit entropy R is built by collecting 256 consecutive bits:  $R[k]=b[k]$  , for  $k=0,1,2,3,..,255$

The 32-bit Freshness Value for message i is a 32-bit segment of this reservoir. Let p be the starting pointer for the segment:

$$FV\_full[i] = R[p, p + 31] \quad (28)$$

For a 32-bit system, only the least significant 4 bytes are used:

$$FV[i] = FV_{full[i]} \bmod 2^{32} \quad (29)$$

#### 4.4.2 Receiver Validation Protocol

When a message is received, Freshness Value is obtained and extracted. Then the process of verifying Freshness Value Synchronization is achieved by contrasting the extracted Freshness Value with the generated Freshness Value at the receiver side .

In case both match each other, the transmission is valid and a flag for resending the attempt is set to “0”. Also, the rest steps should be proceeded such as AES 128

Decryption and SHA 256 verification. While in the event that the Freshness Value is inconsistent: the transmission is invalid and a flag for resending the attempt is set to “1”.

If it fails after three attempts, the message should be ignored.

#### 4.4.3 Transmitter Validation Protocol

Once the receiver indicates an invalid freshness value, the transmitter retransmits the message with the identical freshness value; constrained to a specified number of attempts.

Following three unsuccessful tries, the freshness value should be adjusted (e.g., shifting one bit) and a new message should be sent.

### 4.5 FPGA Implementation Setup

The proposed security protocol is implemented and validated using FPGA-in-the-Loop (FIL) methodology. MATLAB HDL Coder is utilized for hardware code generation, and the design is deployed on a Xilinx FPGA platform.

The system emulates real Electronic Control Unit (ECU) communication, enabling validation of synchronization, latency, and security performance under realistic operating conditions.

The following steps describe the hardware setup and deployment process for implementing the proposed protocol on the FPGA platform in a real-time environment. To utilize this support package, the Xilinx Virtex-6 FPGA DSP Development Kit Support [68] from Simulink (MathWorks) must be installed. This kit provides the necessary design software and development tools for implementing MATLAB and Simulink models on FPGA platforms [66]. After installing the required support packages, the Xilinx Virtex-6 FPGA Connectivity Kit is connected to the FPGA board [68, 69]. Once the hardware and software setup is complete, the Xilinx documentation is used to deploy MATLAB-generated designs onto the Xilinx Virtex-6 FPGA board (XC6VLX240T-1FFG1156C) [68], [69].

#### 4.6 Protocol Analysis and Validation

This section investigates the feasibility of using the Fast Fourier Transform (FFT) to determine the time delay between two nodes. This technique is applied within the proposed security protocol to achieve synchronization between transmitter and receiver nodes. The algorithm is implemented on a Virtex-6 FPGA board, and its performance is evaluated using both time-domain simulation and real-system validation.

##### 4.6.1 Implementation of the security protocol at the Transmitter and the Receiver

After defining the security requirements of the proposed protocol, comprehensive validation is performed through testing and evaluation to ensure that the protocol satisfies the required security objectives without introducing vulnerabilities or weaknesses. Tables 1 through 5 present the evaluation results. Table 1 demonstrates the successful encryption, decryption, and hashing of the encrypted message, verifying the correctness of the cryptographic operations.

**Table 1** Validation of AES 128 Encryption and Decryption with SHA256

|                                                                       |                                                                  |
|-----------------------------------------------------------------------|------------------------------------------------------------------|
| Message= “EngSpeed “, Ignition=”I” Non-Used Bytes =”0000”, Speed=”00” |                                                                  |
| Decimal                                                               | 69 110 103 83 112 101 101 100 32 49 48 48 48 48 48               |
| Hexadecimal                                                           | 456E6753706565642031303030303030                                 |
| Key                                                                   |                                                                  |
| Decimal                                                               | 84 104 97 116 115 32 109 121 32 75 117 110 103 32 70 117         |
| Hexadecimal                                                           | 5468617473206D79204B756E67204675                                 |
| AES 128 Encryption                                                    |                                                                  |
| Decimal Code                                                          | 94 251 229 182 72 139 68 17 185 80 173 123 193 107 22 39         |
| Hexadecimal Code                                                      | 5efbe5b6488b4411b950ad7bc16b1627                                 |
| Hexadecimal. Tool                                                     | 5efbe5b6488b4411b950ad7bc16b1627                                 |
| Results                                                               | Match                                                            |
| AES 128 Decryption                                                    |                                                                  |
| Decimal Code                                                          | 69 110 103 83 112 101 101 100 32 49 48 48 48 48 48               |
| Hexadecimal Code                                                      | 456e6753706565642031303030303030                                 |
| Hexadecimal. Tool                                                     | 456e6753706565642031303030303030                                 |
| Results                                                               | Match                                                            |
| SHA256 (last 4 bytes)                                                 |                                                                  |
| Decimal Code                                                          | 225 211 40 255                                                   |
| Hexadecimal Code                                                      | e0488f13bbc79fd353f4c1afebc9b96dbfb865292e8facac9506fd42e1d328ff |
| Hexadecimal. Tool                                                     | e0488f13bbc79fd353f4c1afebc9b96dbfb865292e8facac9506fd42e1d328ff |
| Results                                                               | Match                                                            |

- Original Message is “EngSpeed 1000000” (ASCII)

The original message is shown in both decimal and hexadecimal formats: Decimal: 69, 110, 103, 83, 112, 101, 101, 100, 32, 49, 48, 48, 48, 48, 48

Hexadecimal: 456E6753706565642031303030303030

- Encryption Key

The AES-128 encryption key is provided in both decimal and hexadecimal formats as well:

- Decimal: 84, 104, 97, 116, 115, 32, 109, 121, 32, 75, 117, 110, 103, 32, 70, 117
- Hexadecimal: 5468617473206D79204B756E67204675

- AES-128 Encryption

After applying AES-128 encryption to the original message using the key, the following results are obtained:

- Encrypted Message (Decimal): 94, 251, 229, 182, 72, 139, 68, 17, 185, 80, 173, 123, 193, 107, 22, 39
- Encrypted Message (Hexadecimal): 5efbe5b6488b4411b950ad7bc16b1627

Table 1 also provides a reference output from an AES-128 Online Tool [66] for comparison, which gives the same result:

- AES-128 Tool Output: 5efbe5b6488b4411b950ad7bc16b1627

The Results field shows a match, confirming that the AES-128 encryption has been correctly implemented, as the computed ciphertext matches the tool's output. This validation step ensures the integrity and correctness of the encryption process.

- AES-128 Decryption

The encrypted message is then decrypted using the same AES-128 key:

- Decrypted Message (Decimal): 69, 110, 103, 83, 112, 101, 101, 100, 32, 49, 48, 48, 48, 48, 48
- Decrypted Message (Hexadecimal): 456E6753706565642031303030303030

The results are also validated against the AES-128 tool's decryption output, which matches exactly:

- AES-128 Tool Output: 456e6753706565642031303030303030

The Results field shows a match, confirming the accuracy of the decryption process. This ensures that the encryption and decryption functions are working symmetrically and that the original message can be accurately recovered from the encrypted data.

- SHA-256 Hashing

After the AES encryption process, the encrypted message undergoes SHA-256 hashing.

Table 1 focuses on the last 4 bytes of the SHA-256 hash:

- SHA-256 Hash (Decimal of Last 4 Bytes): 225, 211, 40, 255
- SHA-256 Hash (Full Hexadecimal):  
e0488f13bbc79fd353f4c1afebc9b96dbfb865292e8facac9506fd42e1d328ff

It provides a reference hash output from an external online tool [67] for comparison, which matches exactly:

- SHA-256 Tool Output:

e0488f13bbc79fd353f4c1afebc9b96dbfb865292e8facac9506fd42e1d328ff

The Results field shows a match, confirming the correct application of the SHA-256 hash function. The match between the computed hash and the reference value ensures that the message's integrity is maintained, and any alteration in the message would result in a different hash value.

Furthermore, Table 2 presents the validation results of six transmitted messages in a secure CAN protocol, showing the successful transmission of messages between the transmitter (TX) and receiver (RX) without requiring any resend attempts. It includes the following components for each of the six messages:

**Table 2** Validation of 6 Messages with No Resend Attempts

| Messages No.                                     | Message (Decimal)                                          | Ignition        | Speed             |
|--------------------------------------------------|------------------------------------------------------------|-----------------|-------------------|
| 1                                                | 69 110 103 83 112 101 101 100 32                           | 49              | 48 48 48 48 48 48 |
| 2                                                | 69 110 103 83 112 101 101 100 32                           | 49              | 48 48 48 48 48 53 |
| 3                                                | 69 110 103 83 112 101 101 100 32                           | 49              | 48 48 48 48 49 48 |
| 4                                                | 69 110 103 83 112 101 101 100 32                           | 49              | 48 48 48 48 50 48 |
| 5                                                | 69 110 103 83 112 101 101 100 32                           | 49              | 48 48 48 48 51 48 |
| 6                                                | 69 110 103 83 112 101 101 100 32                           | 49              | 48 48 48 48 52 48 |
| Transmitter -CAN security Protocol of 6 Messages |                                                            |                 |                   |
| FV                                               | Encrypted MSG                                              | SHA 256         |                   |
| 207 255 224 0                                    | 94 251 229 182 72 139 68 17 185 80 173 123 193 107 22 39   | 225 211 40 255  |                   |
| 159 255 192 0                                    | 52 24 69 78 245 172 248 209 148 63 45 252 215 187 87 23    | 233 177 210 239 |                   |
| 63 255 128 0                                     | 40 16 211 6 28 188 108 5 138 46 49 25 84 4 190 70          | 90 56 201 202   |                   |
| 127 255 0 0                                      | 116 202 93 96 87 228 155 58 233 180 88 250 14 15 212 184   | 46 127 157 242  |                   |
| 255 254 0 0                                      | 76 84 7 191 75 17 1 194 99 255 5 3 19 219 220 25           | 33 26 167 136   |                   |
| 255 252 0 0                                      | 253 158 111 83 185 43 164 255 111 180 104 132 93 8 101 141 | 47 2 132 66     |                   |
| Receiver -CAN security Protocol of 6 Messages    |                                                            |                 |                   |
| FV                                               | Message (Decimal)                                          | SHA 256         |                   |
| 207 255 224 0                                    | 69 110 103 83 112 101 101 100 32 49 48 48 48 48 48 48      | 225 211 40 255  |                   |
| 159 255 192 0                                    | 69 110 103 83 112 101 101 100 32 49 48 48 48 48 48 53      | 233 177 210 239 |                   |
| 63 255 128 0                                     | 69 110 103 83 112 101 101 100 32 49 48 48 48 48 49 48      | 90 56 201 202   |                   |
| 127 255 0 0                                      | 69 110 103 83 112 101 101 100 32 49 48 48 48 48 50 48      | 46 127 157 242  |                   |

|                 |                                                    |                 |                      |              |
|-----------------|----------------------------------------------------|-----------------|----------------------|--------------|
| 255 254 0 0     | 69 110 103 83 112 101 101 100 32 49 48 48 48 51 48 | 33 26 167 136   |                      |              |
| 255 252 0 0     | 69 110 103 83 112 101 101 100 32 49 48 48 48 52 48 | 47 2 132 66     |                      |              |
| Freshness Value | Hashed Value                                       | Resend Attempts | Success transmission | Hashed Flags |
| Match           | Match                                              | [0 0 0 0 0]     | [1 1 1 1 1]          | [0 0 0 0 0]  |

- **Message Data (Decimal):** Each message is represented in its decimal form. This includes both the speed value and the ignition status, which are critical parameters in automotive systems, such as those using the Controller Area Network (CAN).
- **Freshness Value at TX and RX:** The freshness value is a sequence parameter that prevents replay attacks. In Table 2, it displays the freshness value generated and transmitted by the TX, along with the corresponding value at the RX side, which is received and compared for validation purposes. The consistency of freshness values between TX and RX is a key indicator of successful synchronization between the two units.
- **Hashed Value:** The encrypted messages are hashed using the SHA-256 algorithm. The hashed value (least 4 bytes) at the TX side is transmitted along with the message, and upon reception, the RX recalculates the hash and compares its least 4 bytes with the received one. The results of this comparison are also shown.
- **Comparison Results:** it includes a section that details the outcomes of two comparisons:
  - **Freshness Value Comparison:** this verifies whether the freshness value at the RX matches the value transmitted by the TX. A match indicates that the message is fresh and not a replayed message.

- Hashed Value Comparison: This checks whether the recalculated hash at the RX matches the hash transmitted by the TX, confirming the integrity of the message.
- Transmission Success and Hash Flags: For each message, it tracks the outcome of the transmission. A transmission success flag (often a binary array) indicates whether the message was received successfully based on the comparisons of freshness. The hash flag similarly records whether the hash verification was successful. These flags are represented in the table as arrays for each message, providing a clear indication of the communication's reliability and integrity.
- Resend Attempts: it shows that for all six messages, the number of resend attempts is zero. This signifies that each message was transmitted and received correctly on the first attempt, without the need for retransmission due to a mismatch in freshness values or hashed data. This is a crucial system of measurement for evaluating the efficiency and robustness of the protocol under normal working conditions.

The data presented in Table 2 validates the correctness and reliability of the communication protocol over six iterations, with no retransmissions needed. The synchronization of freshness values between TX and RX demonstrates the system's ability to mitigate replay attacks, a common threat in security protocols.

As well as validation of wrong freshness values has been demonstrated in Table 3. In this Table, freshness value mechanism between the transmitter (TX) and receiver (RX) was validated using six messages, which were categorized into four distinct categories. The

protocol's management of resend attempts is evaluated, and each group tests various scenarios in which the freshness values are deliberately modified.

As shown in Table 3, the first two messages are the primary focus of Group 1, who intentionally modified the freshness value of the first message twice and the second message once. On the receiver's end, the protocol's ability to identify inaccurate freshness values was determined, and corresponding retransmission requests were initiated. The transmitter documented the resend attempts in accordance with the feedback provided by the receiver. The receiver responded with a "1" for both freshness checks in the first message, indicating incorrect values. In the second message, the receiver responded with a "1" and a "0." No resend attempts were initiated for the remaining four messages (messages 3–6). The array of resend attempts of these six messages was [1 1 1 0 0 0 0].

Similarly, Category 2 and Category 3 in Table 3 conducted identical scenarios for the subsequent sets of two messages (messages 3 and 4 of Category 2, messages 5 and 6 of Category 3), achieving comparable outcomes.

**Table 3** Validation of Certain Resend Attempts and all Resend Attempts

| Category 1 |               |               |                 |                   |                   |
|------------|---------------|---------------|-----------------|-------------------|-------------------|
| Message ID | FV (TX)       | FV (RX)       | Resend Attempts | Receiver Response | Status            |
| 1          | 207 255 224 0 | 204 255 224 0 | 1               | [1,1]             | Ignored & Shifted |
|            | 207 255 224 0 | 207 255 224 5 | 2               |                   |                   |
| 2          | 159 255 192 0 | 159 244 192 0 | 1               | [1,0]             | Success & Shifted |
|            | 159 255 192 0 | 159 255 192 0 | 2               |                   |                   |
| 3          | 63 255 128 0  | 63 255 128 0  | 0               | [0]               | Success & Shifted |
| 4          | 127 255 0 0   | 127 255 0 0   | 0               | [0]               | Success & Shifted |
| 5          | 255 254 0 0   | 255 254 0 0   | 0               | [0]               | Success & Shifted |
| 6          | 255 252 0 0   | 255 252 0 0   | 0               | [0]               | Success & Shifted |
| Category 2 |               |               |                 |                   |                   |
| Message ID | FV (TX)       | FV (RX)       | Resend Attempts | Receiver Response | Status            |

|            |               |               |                 |                   |                   |
|------------|---------------|---------------|-----------------|-------------------|-------------------|
| 1          | 207 255 224 0 | 207 255 224 0 | 0               | [0]               | Success & Shifted |
| 2          | 159 255 192 0 | 159 255 192 0 | 0               | [0]               | Success & Shifted |
| 3          | 63 255 128 0  | 63 255 3 0    | 1               | [1,1]             | Ignored & Shifted |
|            | 63 255 128 0  | 3 255 128 0   | 1               |                   |                   |
| 4          | 127 255 0 0   | 127 255 10 0  | 1               | [1,0]             | Success & Shifted |
|            | 127 255 0 0   | 127 255 0 0   | 0               |                   |                   |
| 5          | 255 254 0 0   | 255 254 0 0   | 0               | [0]               | Success & Shifted |
| 6          | 255 252 0 0   | 255 252 0 0   | 0               | [0]               | Success & Shifted |
| Category 3 |               |               |                 |                   |                   |
| Message ID | FV (TX)       | FV (RX)       | Resend Attempts | Receiver Response | Status            |
| 1          | 207 255 224 0 | 207 255 224 0 | 0               | [0]               | Success & Shifted |
| 2          | 159 255 192 0 | 159 255 192 0 | 0               | [0]               | Success & Shifted |
| 3          | 63 255 128 0  | 63 255 128 0  | 0               | [0]               | Success & Shifted |
| 4          | 127 255 0 0   | 127 255 0 0   | 0               | [0]               | Success & Shifted |
| 5          | 255 254 0 0   | 255 254 0 1   | 1               | [1,1]             | Ignored & Shifted |
|            | 255 254 0 0   | 5 254 0 0     | 1               |                   |                   |
| 6          | 255 252 0 0   | 255 252 10 0  | 1               | [1,0]             | Success & Shifted |
|            | 255 252 0 0   | 255 252 0 0   | 0               |                   |                   |
| Category 4 |               |               |                 |                   |                   |
| Message ID | FV (TX)       | FV (RX)       | Resend Attempts | Receiver Response | Status            |
| 1          | 207 255 224 0 | 204 255 224 0 | 1               | [1,1]             | Ignored & Shifted |
|            | 207 255 224 0 | 207 255 224 5 | 2               |                   |                   |
| 2          | 159 255 192 0 | 159 244 192 0 | 1               | [1,1]             | Ignored & Shifted |
|            | 159 255 192 0 | 159 255 92 0  | 2               |                   |                   |
| 3          | 63 255 128 0  | 63 255 3 0    | 1               | [1,1]             | Ignored & Shifted |
|            | 63 255 128 0  | 3 255 128 0   | 2               |                   |                   |
| 4          | 127 255 0 0   | 127 255 0 34  | 1               | [1,1]             | Ignored & Shifted |
|            | 127 255 0 0   | 127 255 0 32  | 2               |                   |                   |
| 5          | 255 254 0 0   | 255 254 1 2   | 1               | [1,1]             | Ignored & Shifted |
|            | 255 254 0 0   | 255 254 12 3  | 2               |                   |                   |
| 6          | 255 252 0 0   | 12 252 0 0    | 1               | [1,1]             | Ignored & Shifted |
|            | 255 252 0 0   | 255 1 0 0     | 2               |                   |                   |

By effectively reaching the maximum number of resend attempts for incorrect freshness values and transitioning correctly to the subsequent message, the CAN protocol exhibited its robustness. The system positively reset and continued secure communication upon the confirmation of a legitimate freshness value.

Another case was created for the validation by category 4 as shown in Table 3, which tested a scenario in which each freshness value achieved its maximum limit across all

messages. The system is to abandon the messages and proceed to the next message in the sequence as a result of the freshness value of each message being intentionally set to exceed the maximum allowable resend attempts. Consequently, the CAN protocol maintained its integrity by correctly resetting for the subsequent message and continuing secure communication once a valid freshness value was detected. This illustrates the protocol's ability to manage the most severe conditions, in which message abandonment results from continuous mismatches in freshness values. However, the system remains resilient and does not compromise on security. The protocol's efficiency in preventing the processing of old or fraudulently altered messages is further substantiated by the results of Category 4.

In order to further validate the data from this Secure CAN protocol, Table 4 offers insightful overview about the effectiveness of this protocol in error handling and message integrity mechanisms. This demonstrates 3 groups of testing both Resend attempts and hash Flag parameters. In first group at the top of Table 4, 6 messages with varying number of resend attempts and hash flag statuses are shown. 1st message was resent twice. Although hash flag equals “0 “ in both cases, the message was neither successfully received nor corrupted. With 2nd message, hash flag was “1” and no Resend attempts was received. That is, the message has successfully been received and marked as corrupted. For 3d and 6<sup>th</sup> messages, Each had 2 resend attempts with hash flag equals “0”. However, both were neither successfully received nor corrupted. In case of 4<sup>th</sup> and 5<sup>th</sup> messages, both had no resend attempts. Therefore, both were received with success. In contrast, hash verification succeeded for 4<sup>th</sup> message and failed for 5<sup>th</sup> message.

**Table 4** Validation of mixed Resend Attempts with Hash Flags

| Some Resend Attempts with Hash Flags                    |                         |                                                 |                   |                   |
|---------------------------------------------------------|-------------------------|-------------------------------------------------|-------------------|-------------------|
| Message ID                                              | Resend Attempts Counter | Hash Flag (0 = Success or ignored, 1 = Failure) | Success Reception | Corrupted Message |
| 1                                                       | 2                       | 0                                               | 0                 | 0                 |
| 2                                                       | 0                       | 1                                               | 1                 | 1                 |
| 3                                                       | 2                       | 0                                               | 0                 | 0                 |
| 4                                                       | 0                       | 0                                               | 1                 | 0                 |
| 5                                                       | 0                       | 1                                               | 1                 | 1                 |
| 6                                                       | 2                       | 0                                               | 0                 | 0                 |
| No Resend Attempts with Hash Flags=0                    |                         |                                                 |                   |                   |
| Message ID                                              | Resend Attempts Counter | Hash Flag (0 = Success or ignored, 1 = Failure) | Success Reception | Corrupted Message |
| 1                                                       | 0                       | 0                                               | 1                 | 0                 |
| 2                                                       | 0                       | 0                                               | 1                 | 0                 |
| 3                                                       | 0                       | 0                                               | 1                 | 0                 |
| 4                                                       | 0                       | 0                                               | 1                 | 0                 |
| 5                                                       | 0                       | 0                                               | 1                 | 0                 |
| 6                                                       | 0                       | 0                                               | 1                 | 0                 |
| No Resend Attempts with Hash Flags=1 of Last 3 Messages |                         |                                                 |                   |                   |
| Message ID                                              | Resend Attempts Counter | Hash Flag (0 = Success or ignored, 1 = Failure) | Success Reception | Corrupted Message |
| 1                                                       | 0                       | 0                                               | 1                 | 0                 |
| 2                                                       | 0                       | 0                                               | 1                 | 0                 |
| 3                                                       | 0                       | 0                                               | 1                 | 0                 |
| 4                                                       | 0                       | 1                                               | 0                 | 1                 |
| 5                                                       | 0                       | 1                                               | 0                 | 1                 |
| 6                                                       | 0                       | 1                                               | 0                 | 1                 |

In the second group of Table 4, it depicts six messages with no resend attempts and all hash flags set to 0. all messages were transmitted without resend attempts, and hash verification succeeded for each (Hash Flag = 0). Subsequently, all messages were successfully received and considered uncorrupted.

For the last group of Table 4, it focuses on the last three where hash flags are set to 1. From 1<sup>st</sup> to 3<sup>th</sup> messages, no resend attempts were received and hash verification was successfully pass. Thus, messages were marked as uncorrupted. The rest messages of this

group were successfully received based on no resend attempts. With hash flags set to “1”, messages were labeled as corrupted.

Finally, Table 5 shows different types of corruption, including scenarios where the encrypted message is tampered with but the hash value is correct, and vice versa. It summarizes the results of three test messages, showing how the receiver handles different scenarios of encryption and hash mismatches:

**Table 5** Corruption of Encrypted Data and Hash Value Validation

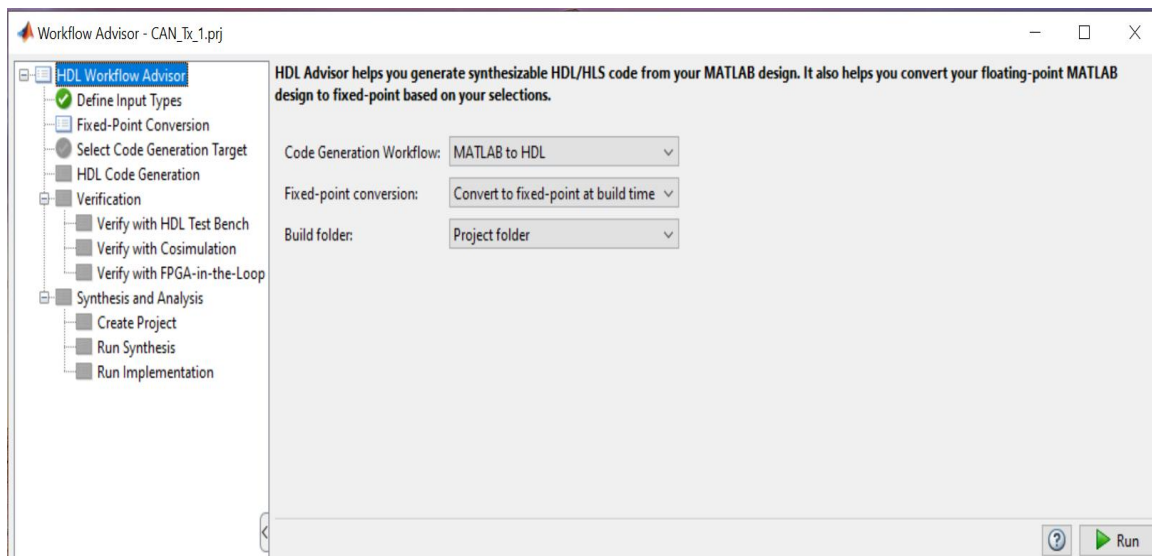
| Message ID | Encrypted Message Corruption (0 = No, 1 = Yes) | Hash Value Corruption (0 = No, 1 = Yes) | Hash Flag (0 = Success, 1 = Failure) | Success Reception (0 = No, 1 = Yes) | Corrupted Message (0 = No, 1 = Yes) |
|------------|------------------------------------------------|-----------------------------------------|--------------------------------------|-------------------------------------|-------------------------------------|
| 1          | 1                                              | 0                                       | 1                                    | 0                                   | 1                                   |
| 2          | 1                                              | 1                                       | 1                                    | 0                                   | 1                                   |
| 3          | 0                                              | 1                                       | 1                                    | 0                                   | 1                                   |

In Table 5, it reveals that even if an encrypted message is corrupted but the hash is correct (Message 1), the receiver can still recognize the problem and reject the corrupted message. In cases where both the encrypted message and hash are corrupted (Message 2), the CAN protocol appropriately marks the message as corrupted. Lastly, Message 3 illustrates the case where a correct encrypted message with an incorrect hash value can still result in unsuccessful reception, showing the protocol’s capability to differentiate between data integrity issues and maintain secure communication.

#### 4.6.2 FPGA in the Loop implementation of the Replay attack

To perform a replay attack on a CAN network’s security protocol using an FPGA board within the FPGA-in-the-loop (FIL) validation framework, we first focus on the transmitter

(TX) side. In our approach, the legitimate data transmission is monitored and captured, and all data is stored in either a file or buffer for later use. To simulate replay attacks, selected messages from the stored data can be copied and replayed while the receiver (RX) side is active. For the FIL validation of the TX side, we manipulate the hash\_flag array and the resend\_attempts array to ensure that the TX responds correctly to various scenarios. For the RX side, we intentionally modify the encrypted\_message, freshness\_values, and hash\_values to observe how the RX reacts under different conditions, including mismatches and corrupted messages. We validate these manipulations by executing the protocol through the full cycle of steps in MATLAB's HDL Coder as shown in Figure (52), ensuring the correctness of the entire system on the FPGA board.



**Figure 52** MATLAB HDL Coder of CAN security Protocol -Tx

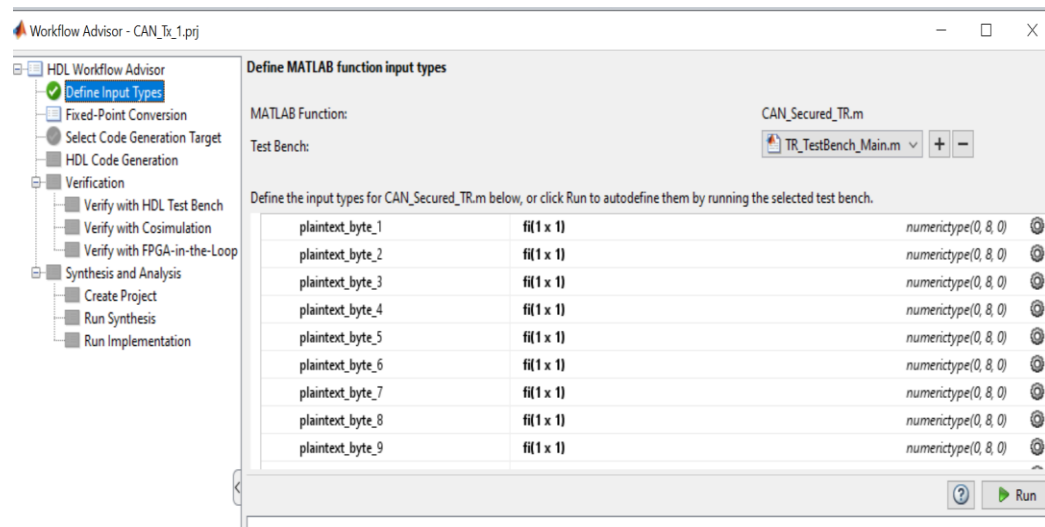
This process allows us to test both TX and RX under potential attack scenarios, verifying their responses and ensuring the security mechanisms are robust against replay attacks.

#### 4.6.2.1 MATLAB to HDL Conversion

The CAN Security protocol-TX involves key security operations like AES-128 encryption, SHA-256 hashing, and managing freshness values, all of which need to be accurately converted into HDL for FPGA deployment. Thus, HDL Coder guarantees that MATLAB code is transformed into efficient, synthesizable VHDL or Verilog code [70].

#### 4.6.2.2 Defining Fixed-Point and HDL Types

Encryption and hashing of the protocol need to handle precise data type of FPGA implementation. As Fixed-point arithmetic is supported by HDL Coder which is crucial for FPGA designs, floating-point operations are not typically feasible for hardware implementation. As a result, defining types such as fixed-point (fi objects) guarantees that the generated HDL code match the constraints architecture and Resources of the FPGA [71]. As you can see from Figure 53 and 54



**Figure 53** Fixed-Point Data Type Representation-TX

#### 4.6.2.3 Code Optimization and Scalarization

Optimization techniques like scalarization can be allowed by HDL Coder during the HDL code generation process. That converts vector-based operations into scalar ones in order to obtain better FPGA performance. This plays a crucial role in minimizing resource consumption while enhancing the performance of operations such as AES-128 encryption and SHA-256 hashing.

#### 4.6.2.4 HDL Verification in MATLAB

Before deploying to the FPGA, the generated HDL code is verified within MATLAB using HDL simulation. This step ensures that the HDL code matches the behavior of the original MATLAB code for the Tx CAN FD Security Protocol. Simulation tests can verify the correctness of message encryption, SHA-256 hashing, and the handling of freshness values, ensuring that the functionality is accurate in hardware terms.

#### 4.6.2.5 FPGA Deployment

After successful verification, the HDL code is deployed to the FPGA. This involves mapping the code to the FPGA's hardware resources (like LUTs, flip-flops, and DSP blocks). Pin assignment files are used to ensure proper communication with external systems, such as the CAN bus, to transmit messages securely.

#### 4.6.2.6 FPGA-in-the-Loop (FIL) Testing

Once deployed, the FPGA operates in real-time with MATLAB via the FIL interface. MATLAB communicates with the FPGA, sending real-time CAN messages to the TX protocol running on the FPGA and receiving feedback for performance evaluation. FIL testing ensures the hardware's real-time response to various scenarios, including:

Message Integrity Checks: Verifying that AES-128 and SHA-256 ensure data integrity and security.

**Synchronization:** Testing the correctness of freshness value handling to prevent replay attacks. **Encryption Validation:** Ensuring that the FPGA performs AES-128 encryption correctly and in real-time for secure CAN communication.

While in the FIL setup of the RX CAN protocol, it is tested in real-time by receiving CAN messages from the FPGA. MATLAB interacts with the FPGA to verify the RX's response to different scenarios, such as mismatched encryption, replayed messages, or corrupted data.

#### 4.6.2.7 Performance Evaluation

FIL enables detailed performance testing of the protocol in a real hardware environment, observing metrics like latency, throughput, and power consumption.

Real-time stress tests can evaluate how the protocol handles security attacks or corruptions, ensuring the robustness of the security measures in the TX CAN Security Protocol. Comparison with simulation results ensures that the FPGA performs as expected, providing validation for deployment in secure CAN systems. Similarly, FIL of the RX , it evaluates the same measures. In addition, stress testing under attack scenarios ensures the resilience and correctness of CAN security protocol.

#### 4.6.2.8 Hardware Constraints and Adjustments

Initial implementation efforts targeted an FPGA board with 1,182 input/output (I/Os) pins, allowing for comprehensive data processing and evaluation. However, due to

hardware limitations—including a broken board and constraints on available resources—the protocol was later adapted to an FPGA board with only 155 I/O pins. This significant reduction in available I/Os necessitated optimizations in the implementation.

Despite these constraints, the security protocol's functionality was preserved. Key adjustments included efficient data routing, reduced pin usage, and resource optimization to fit within the new FPGA's architecture. These optimizations demonstrated that the protocol could remain functional under resource-constrained environments, aligning with practical deployment scenarios in automotive systems.

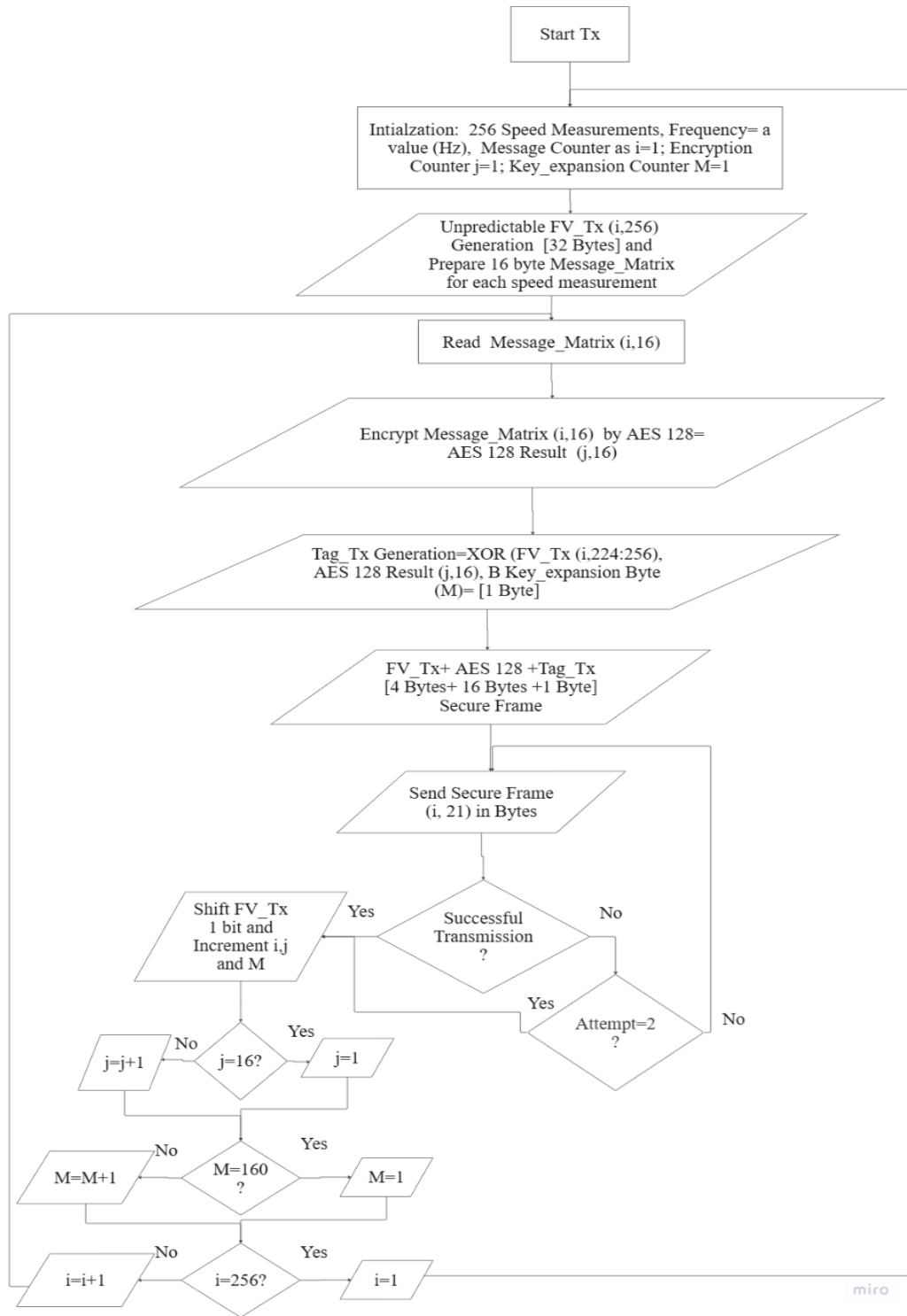
#### 4.7 Optimized FPGA Implementation with Reduced IO and Lightweight Hashing

The FPGA implementation of the CAN security protocol has been optimized to address challenges associated with input/output count, cost, and efficiency. The primary motivations for IO reduction were two factors: the failure of the previous FPGA board, which had approximately 600 IOs, and the inherent issues associated with high IO counts, including resource utilization and development complexity.

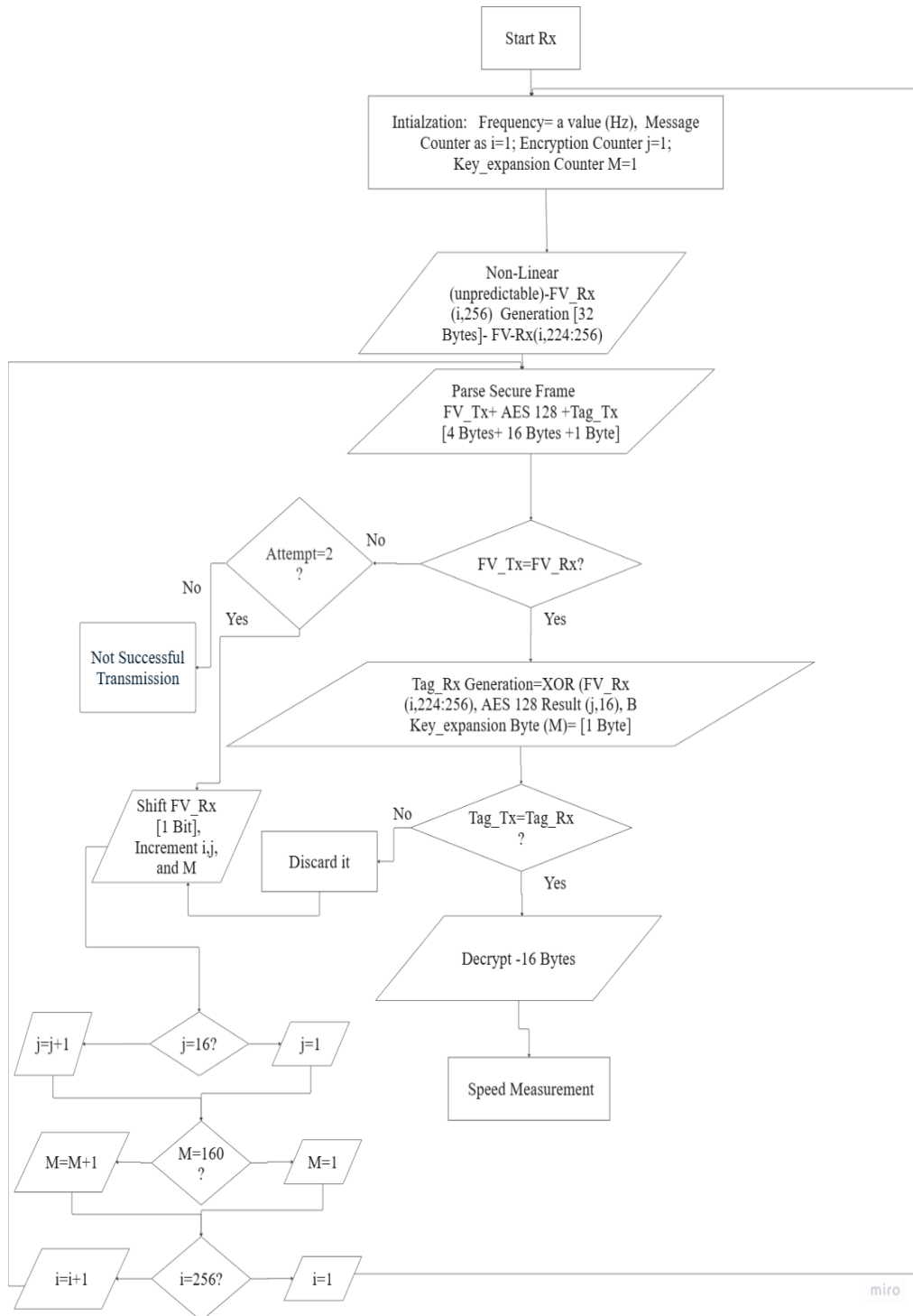
The previous FPGA board, despite remaining functional, would have presented considerable challenges due to its high IO count, resulting in extended development time, increased resource consumption, and diminished scalability. The identified issues required the selection of a new FPGA board characterized by a reduced number of IOs and enhanced efficiency. The updated design decreases the transmitter IO count from 1182 to 152, preserving the functionality of the CAN security protocol. Figure 54 and 55 illustrate these changes, presenting a flowchart of the optimized design workflow.

Several critical modifications were implemented to decrease the transmitter IO count from 1182 to 152. The initial design employed a 16-byte message input, with each byte represented by 8 bits, providing a total of 128 bits ( $16 \times 8$ ). For the optimized design, the input size for the 15th and 16th bytes of the message was reduced to 12 bits ( $2 \times 6$ ), as they were each represented by 6 bits.

To meet cycle and IO constraints on the Arty A7 while preserving replay resistance, we replaced the SHA-256-truncated tag with an ultra-lightweight XOR-based tag that mixes the 32-bit FV, one ciphertext byte, and one key-schedule byte. The strong unpredictability and entropy of the chaotic Freshness Value (FV) satisfy NIST [83] randomness performance metrics, providing the dominant source of security. Consequently, the 1-byte tag functions as an integrity indicator rather than a full cryptographic MAC, since the FV itself ensures message uniqueness and anti-replay protection. Given that the chaotic FV already exhibits near-ideal entropy and passes statistical randomness tests, extending the tag beyond 1 byte would not yield significant additional replay-resistance benefits under the current threat model, while increasing hardware cost. Therefore, the 1-byte tag represents an optimal trade-off between lightweight implementation and sufficient integrity signaling for automotive-grade communication.



**Figure 54** CAN FD Tx Flowcharts- Lightweight Cryptography



**Figure 55** CAN FD Rx Flowchart - Lightweight Cryptography

The initial design produced SHA 256 output of 256 bits, with the 224 to 256 bits being utilized for hashing. The chunked SHA 256 output was represented by 32 bits for each byte, resulting in a total of 1024 bits ( $32 \times 32$ ). In contrast, the optimized design generates a 1-byte hashing output, which is represented by 8 bits and incorporates a 5-bit counter, thereby substantially decreasing the output size. Both designs maintain the same control signals, including reset (rst) and clock (clk).

While the initial design generated a 16-byte output, with each byte represented by 8 bits, the optimized design multiplexes the output in a 3:1 ratio. This implies that the complete outcome of the CAN secure protocol transmission (TX) is obtained every 17 cycles. A series of nested XOR operations that combine the 4-byte freshness value, the encrypted message bytes, and the 160-byte key expansion table are used in the freshness value integration process as indicated in Table 6.

This process guarantees the uniqueness and robust security of each transmission. The following is a detailed explanation of the process: A 4-byte value (F1, F2, F3, F4) generated using a nonlinear function to ensure uniqueness. Furthermore, A 16-byte message (M1, M2, ..., M16), where each byte is 8 bits. Also, a 176-byte table derived from the 16-byte AES key. The first 16 bytes are the original key, and the remaining 160 bytes are expanded values.

**Table 6** Mechanism of lightweight Hashing each Transmission

| Transmission | Encrypted<br>Message<br>Byte | Freshness Value<br>Bytes              | Key<br>Expansion<br>Table<br>Byte | Nested XOR Operation                                                       |
|--------------|------------------------------|---------------------------------------|-----------------------------------|----------------------------------------------------------------------------|
| 1            | $M_1$                        | $(F_1, F_2, F_3, F_4)$                | $K_{17}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_1) \oplus K_{17}$    |
| 2            | $M_2$                        | $(F_1, F_2, F_3, F_4)_{\text{new}}$   | $K_{18}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_2) \oplus K_{18}$    |
| 3            | $M_3$                        | $(F_1, F_2, F_3, F_4)_{\text{new1}}$  | $K_{19}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_3) \oplus K_{19}$    |
| 4            | $M_4$                        | $(F_1, F_2, F_3, F_4)_{\text{new2}}$  | $K_{20}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_4) \oplus K_{20}$    |
| 5            | $M_5$                        | $(F_1, F_2, F_3, F_4)_{\text{new3}}$  | $K_{21}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_5) \oplus K_{21}$    |
| 6            | $M_6$                        | $(F_1, F_2, F_3, F_4)_{\text{new4}}$  | $K_{22}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_6) \oplus K_{22}$    |
| 7            | $M_7$                        | $(F_1, F_2, F_3, F_4)_{\text{new5}}$  | $K_{23}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_7) \oplus K_{23}$    |
| 8            | $M_8$                        | $(F_1, F_2, F_3, F_4)_{\text{new6}}$  | $K_{24}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_8) \oplus K_{24}$    |
| 9            | $M_9$                        | $(F_1, F_2, F_3, F_4)_{\text{new7}}$  | $K_{25}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_9) \oplus K_{25}$    |
| 10           | $M_{10}$                     | $(F_1, F_2, F_3, F_4)_{\text{new8}}$  | $K_{26}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_{10}) \oplus K_{26}$ |
| 11           | $M_{11}$                     | $(F_1, F_2, F_3, F_4)_{\text{new9}}$  | $K_{27}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_{11}) \oplus K_{27}$ |
| 12           | $M_{12}$                     | $(F_1, F_2, F_3, F_4)_{\text{new10}}$ | $K_{28}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_{12}) \oplus K_{28}$ |
| 13           | $M_{13}$                     | $(F_1, F_2, F_3, F_4)_{\text{new11}}$ | $K_{29}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_{13}) \oplus K_{29}$ |
| 14           | $M_{14}$                     | $(F_1, F_2, F_3, F_4)_{\text{new12}}$ | $K_{30}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_{14}) \oplus K_{30}$ |
| 15           | $M_{15}$                     | $(F_1, F_2, F_3, F_4)_{\text{new13}}$ | $K_{31}$                          | $((((F_1 \oplus F_2) \oplus F_3) \oplus F_4) \oplus M_{15}) \oplus K_{31}$ |

|     |          |                                        |            |                                                                        |
|-----|----------|----------------------------------------|------------|------------------------------------------------------------------------|
| 16  | $M_{16}$ | $(F_1, F_2, F_3, F_4)_{\text{new14}}$  | $K_{32}$   | $((F_1 \oplus F_2) \oplus F_3) \oplus F_4 \oplus M_{16} \oplus K_{32}$ |
| 17  | $M_1$    | $(F_1, F_2, F_3, F_4)_{\text{new15}}$  | $K_{33}$   | $((F_1 \oplus F_2) \oplus F_3) \oplus F_4 \oplus M_1 \oplus K_{33}$    |
| ... | ...      | ...                                    | ...        | ...                                                                    |
| n   | $M_n$    | $(F_1, F_2, F_3, F_4)_{\text{new\_n}}$ | $K_{n+16}$ | $((F_1 \oplus F_2) \oplus F_3) \oplus F_4 \oplus M_n \oplus K_{n+16}$  |

For Table 6, it describes how the CAN security protocol is transmitted, emphasizing how the encrypted message, key expansion table, and freshness value are dynamically integrated. A 4-byte freshness value is utilized for each transmission, and each transmission is designated a sequence number.

This freshness is updated after each successful transmission, typically by shifting its bits one bit to the left, thereby ensuring its uniqueness and improving security. The encrypted message is processed one byte at a time, with the n-th byte of the message being combined with the corresponding byte from the 160-byte key expansion table.

The combination is accomplished by performing a nested XOR operation that sequentially XORs the freshness value bytes, the encrypted message byte, and the key expansion table byte. The protocol's resistance to replay assaults is further enhanced by updating the freshness value after each transmission to preserve its dynamic nature. The freshness value is guaranteed to be unique for each cycle, and each transmission is securely processed using this structured approach.

In Table 7, it shows the optimized design validation of the secure CAN protocol. It includes the padded message with speed and non-used bytes, the encryption key, the encrypted and decrypted outputs in both decimal and hexadecimal formats, and the result of a lightweight hashing operation.

The cryptographic processes are verified as correct and robust by the fact that both the sender and receiver accomplish matching results. The table's contents are comprehensively described below.

**Table 7** Validation of AES 128 Encryption and Decryption with Lightweight Hashing

|                                                                              |                                                          |
|------------------------------------------------------------------------------|----------------------------------------------------------|
| Padded Message= "11111111 ", Ignition="1" Non-Used Bytes ="0000", Speed="00" |                                                          |
| Decimal                                                                      | 49 49 49 49 49 49 49 49 32 49 48 48 48 48 48 48          |
| Hexadecimal                                                                  | 31313131313131312031303030303030                         |
| Key                                                                          |                                                          |
| Decimal                                                                      | 84 104 97 116 115 32 109 121 32 75 117 110 103 32 70 117 |
| Hexadecimal                                                                  | 5468617473206D79204B756E67204675                         |
| AES 128 Encryption                                                           |                                                          |
| Decimal Code                                                                 | 117 3 151 250 9 6 143 151 178 146 207 74 39 102 223 200  |
| Hexadecimal Code                                                             | 750397fa09068f97b292cf4a2766dfc8                         |
| Hexadecimal. Tool                                                            | 750397fa09068f97b292cf4a2766dfc8                         |
| Results                                                                      | Match                                                    |
| AES 128 Decryption                                                           |                                                          |
| Decimal Code                                                                 | 49 49 49 49 49 49 49 49 32 49 48 48 48 48 48 48          |
| Hexadecimal Code                                                             | 31313131313131312031303030303030                         |

|                              |                                                                            |
|------------------------------|----------------------------------------------------------------------------|
| Hexadecimal.<br>Tool         | 313131313131312031303030303030                                             |
| Results                      | Match                                                                      |
| Light Hashing (last 4 bytes) |                                                                            |
| Decimal Code                 | $(((((207 \oplus 255) \oplus 224) \oplus 0) \oplus 117), \oplus 226) = 71$ |
| Hexadecimal<br>Code          | 47                                                                         |
| Results                      | Match                                                                      |

As you can see from Table 7, the communication system between a Transmitter and a Receiver is illustrated by the successful encryption, decryption, and lightweight hashing process of a message. The padded message, "11111111", is represented in both decimal and hexadecimal formats, with the decimal values corresponding to the ASCII representation of the characters. Additionally, the message contains elements such as 'Ignition="1"', 'Non-Used Bytes="0000"', and 'Speed="00"'.

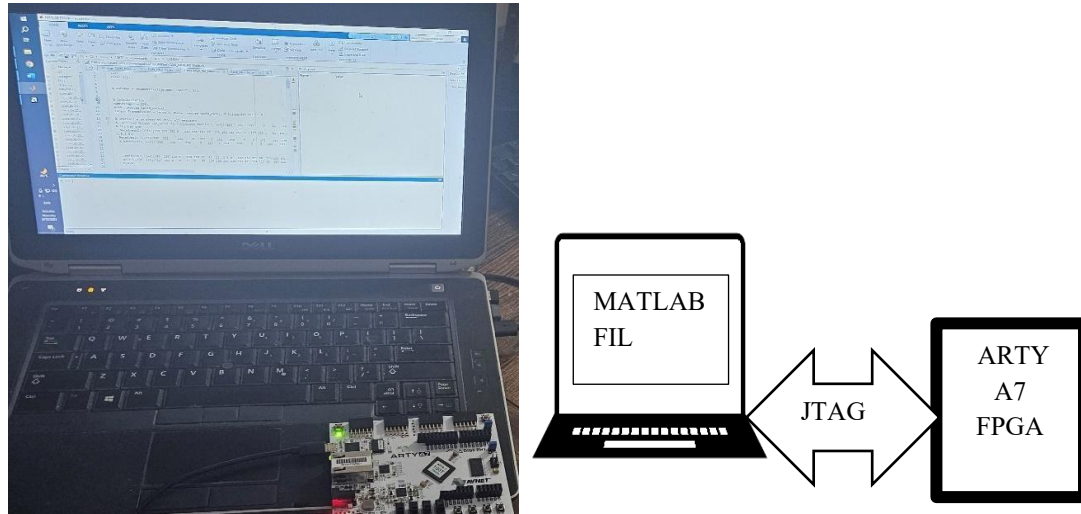
The key "Thats my Kung Fu" is employed in the AES-128 encryption procedure and is also available in hexadecimal and decimal formats. The message is encrypted into a decimal code 117 3 151 250 9 6 143 151 178 146 207 74 39 102 223 200 and its hexadecimal equivalent 750397fa09068f97b292cf4a2766dfc8 , both of which correspond to the output from the encryption tool, thereby verifying the encryption's success. The message is restored to its original form upon decryption, with the decimal 49 49 49 49 49 49 49 32 49 48 48 48 48 48 48 48 48 and hexadecimal

313131313131313120313030303030 values matching the original message and the decryption tool's output, thereby verifying the decryption process.

In addition, the last four bytes of the encrypted message are subjected to a lightweight hashing function that employs an XOR operation  $(((((207 \oplus 255) \oplus 224) \oplus 0) \oplus 117), \oplus 226) = 71$ . This results in a decimal value of `71` and a hexadecimal value of `47`, which is consistent with the anticipated output. The results confirm that the message has been effectively processed by both the sender and receiver, as evidenced by the matching encryption, decryption, and hashing outputs. This ensures the integrity and security of the communication system. The robustness of the cryptographic processes that have been implemented is illustrated by this end-to-end verification.

#### 4.7.1 Transmitter/Receiver FPGA-in-the-Loop (FIL) Implementation

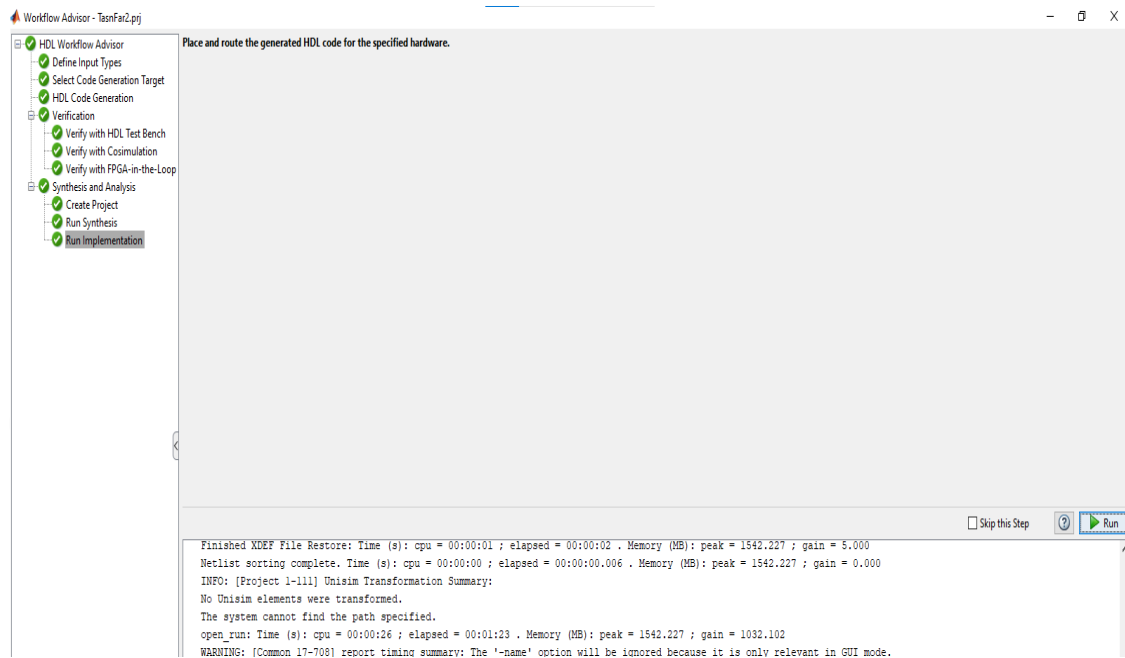
In this section, PC-FPGA setup is shown in Figure 56. It illustrates the FPGA-in-the-Loop (FIL) validation setup, showing the physical connection between the host PC



**Figure 56** PC-FPGA Setup -ARTY A7 board

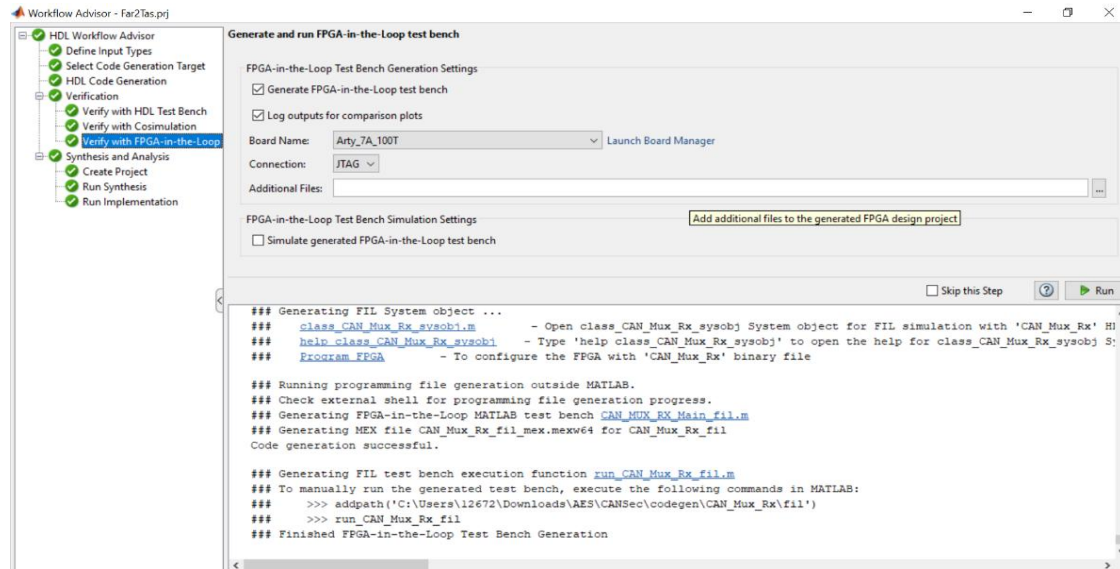
running MATLAB and the Xilinx ARTY A7 FPGA development board. The configuration utilizes USB-JTAG interface for real-time co-simulation, enabling hardware-accelerated validation of the CAN FD security protocol. This setup facilitates bidirectional data exchange where MATLAB provides test vectors and control signals while the FPGA executes the actual transmitter and receiver protocols, returning results for analysis [83].

Figures 57 and 58 demonstrate the proposed CAN FD Tx/Rx functions reliably under varying transmission conditions of FPGA in the Loop Implementation. In all scenarios, the proposed CAN FD Tx/Rx correctly validated the lightweight hashing scheme with unpredicted FVs, and MATLAB confirmed every received frame. With a green check indicator, FPGA in the loop has been successfully verified.



**Figure 57** FPGA in the Loop implementation of CAN FD TX

The three messages provided effectively cover the core operational scenarios of our thread model including replay attack resistance and preventing spoofing. This limited set,



**Figure 58** FPGA in the Loop Implementation of CAN FD Rx

which has been selected from a full list of 256 messages for , successfully validates the system's fundamental behaviors, including standard transmission, hashing integrity, and counter-based sequencing of FVs.

CAN FD Tx verifies the secured protocol using a resend-attempt matrix that records whether each frame is sent successfully (0) or requires one retry (1). For each of the 256 frames, if the first transmission fails, a single resent is attempted; if both fail, the frame is ignored, and CAN FD Tx system moves to the next message with a new freshness value. This method ensures reliable delivery without stalling the bus, limits overhead to one retry per frame, and allows easy assessment of protocol performance by summing resend attempts (ranging from 0 to 512). To maintain synchronization, a frame

counter should be included. Quantitative evaluation was established by the successful hardware implementation. Consequently, the numerical results obtained from FPGA-in-the-loop testing are presented in the following Tables 8, 9, 10,11 and 12, which demonstrate the synchronization and reliability performance of the verified system.

**Table 8** Three CAN FD Frames with Correct FVs

| FV            | AES 128 Encryption                                           | Tag |
|---------------|--------------------------------------------------------------|-----|
| 207 255 224 0 | 117 3 151 250 9 6 143 151 178 146 207 74 39<br>102 223 200   | 71  |
| 159 255 192 0 | 219 166 143 10 240 162 184 157 21 177 195<br>1 61 181 89 80  | 52  |
| 63 255 128 0  | 128 219 95 41 21 175 8 192 132 225 94 224<br>151 131 251 127 | 227 |

Table 8 shows all-success case: 256 CAN FD frames transmitted successfully with no resend flag (flag = 0) throughout the session and just three of them. While Table 9 presents the successful receiver (Rx) results for the "all-success case" where all inputs from Table 8 were processed without error.

**Table 9** Rx Results of All -Success Case

| Message Counter | AES 128 Decryption |      | Tx flag | Hash flag | Resend flag | Count |
|-----------------|--------------------|------|---------|-----------|-------------|-------|
|                 | Sp.1               | Sp.2 |         |           |             |       |
| 1               | 48                 | 48   | 1       | 0         | 0           | 1     |
| 2               | 48                 | 53   | 1       | 0         | 0           | 2     |
| 3               | 49                 | 48   | 1       | 0         | 0           | 3     |

For each of the three messages, the Rx core successfully decrypted the AES-128 payload, correctly outputting the original speed values (Sp.1 and Sp.2 as 48, 48, 48, 53, and 49, 48). The status flags confirm the protocol's correct operation: a Tx flag of '1' indicates successful reception, a Hash flag of '0' verifies that the authentication tag was valid and the message integrity was intact, and a Resend flag of '0' shows that no retransmission was required. The incrementing Count value demonstrates that the receiver processed each message sequentially and maintained perfect synchronization. With the transmitter throughout the session. These results validate that the system correctly authenticates and processes messages under ideal conditions. Single-resend case: 500 CAN FD frames where each frame experienced at least one resend condition (flag = 1 exactly once) and only 6 of them depicted in Table 10. Mixed condition: less than 512 CAN FD frames with variable delivery patterns, where some frames were received on the first attempt (flag = 0) while others required two resend attempts (flag = 1 twice) as clearly exhibited in Table 11.

**Table 10** 6 CAN FD Frames each message sent with wrong and Correct FVs

| FV            | AES 128 Encryption                                          | Tag |
|---------------|-------------------------------------------------------------|-----|
| 6 255 224 0   | 117 3 151 250 9 6 143 151 178 146 207 74<br>39 102 223 200  | 71  |
| 207 255 224 0 | 117 3 151 250 9 6 143 151 178 146 207 74<br>39 102 223 200  | 71  |
| 6 255 192 0   | 219 166 143 10 240 162 184 157 21 177<br>195 1 61 181 89 80 | 52  |

|               |                                                              |     |
|---------------|--------------------------------------------------------------|-----|
| 159 255 192 0 | 219 166 143 10 240 162 184 157 21 177<br>195 1 61 181 89 80  | 52  |
| 34 255 128 0  | 128 219 95 41 21 175 8 192 132 225 94 224<br>151 131 251 127 | 227 |
| 63 255 128 0  | 128 219 95 41 21 175 8 192 132 225 94 224<br>151 131 251 127 | 227 |

**Table 11** Rx Results of CAN FD Frames sent with wrong and Correct FVs

| Message Counter | AES 128 Decryption |      | Tx flag | Hash flag | Resend flag | Count |
|-----------------|--------------------|------|---------|-----------|-------------|-------|
|                 | Sp.1               | Sp.2 |         |           |             |       |
| 1               | 0                  | 0    | 0       | 1         | 2           | 1     |
| 2               | 48                 | 48   | 1       | 0         | 0           | 1     |
| 3               | 0                  | 0    | 0       | 1         | 2           | 2     |
| 4               | 48                 | 53   | 1       | 0         | 0           | 2     |
| 5               | 0                  | 0    | 0       | 1         | 2           | 3     |
| 6               | 49                 | 48   | 1       | 0         | 0           | 3     |

**Table 12** Five CAN FD Frames of mixed Correct or Wrong FVs

| FV            | AES 128 Enc.                                                 | Tag |
|---------------|--------------------------------------------------------------|-----|
| 207 255 224 0 | 117 3 151 250 9 6 143 151 178 146 207 74 39<br>102 223 200   | 71  |
| 6 255 192 0   | 219 166 143 10 240 162 184 157 21 177 195<br>1 61 181 89 80  | 52  |
| 6 255 192 0   | 219 166 143 10 240 162 184 157 21 177 195<br>1 61 181 89 80  | 52  |
| 3 255 128 0   | 128 219 95 41 21 175 8 192 132 225 94 224<br>151 131 251 127 | 227 |
| 63 255 128 0  | 128 219 95 41 21 175 8 192 132 225 94 224<br>151 131 251 127 | 227 |

**Table 13** Rx Results of CAN FD Frames sent with mixed Correct or Wrong FVs

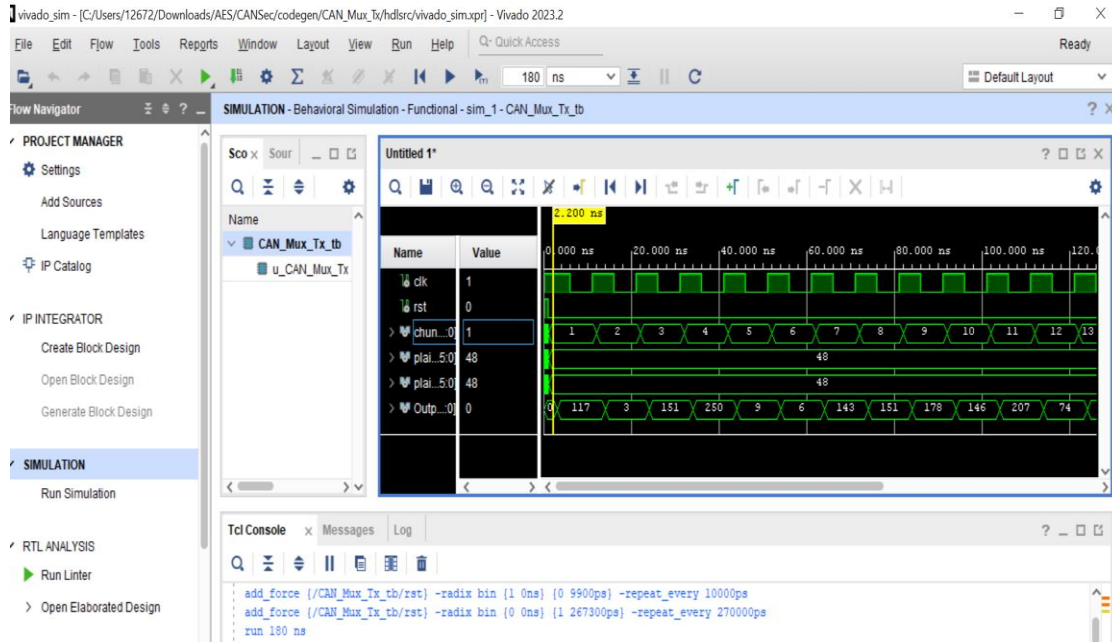
| Message Counter | Sp.1 | Sp.2 | Tx flag | Hash flag | Resend flag | Count |
|-----------------|------|------|---------|-----------|-------------|-------|
| 1               | 48   | 48   | 1       | 0         | 0           | 1     |
| 2               | 0    | 0    | 0       | 1         | 2           | 2     |
| 3               | 0    | 0    | 0       | 1         | 2           | 2     |

|   |    |    |   |   |   |   |
|---|----|----|---|---|---|---|
| 4 | 0  | 0  | 0 | 1 | 2 | 3 |
| 5 | 49 | 48 | 0 | 1 | 0 | 3 |

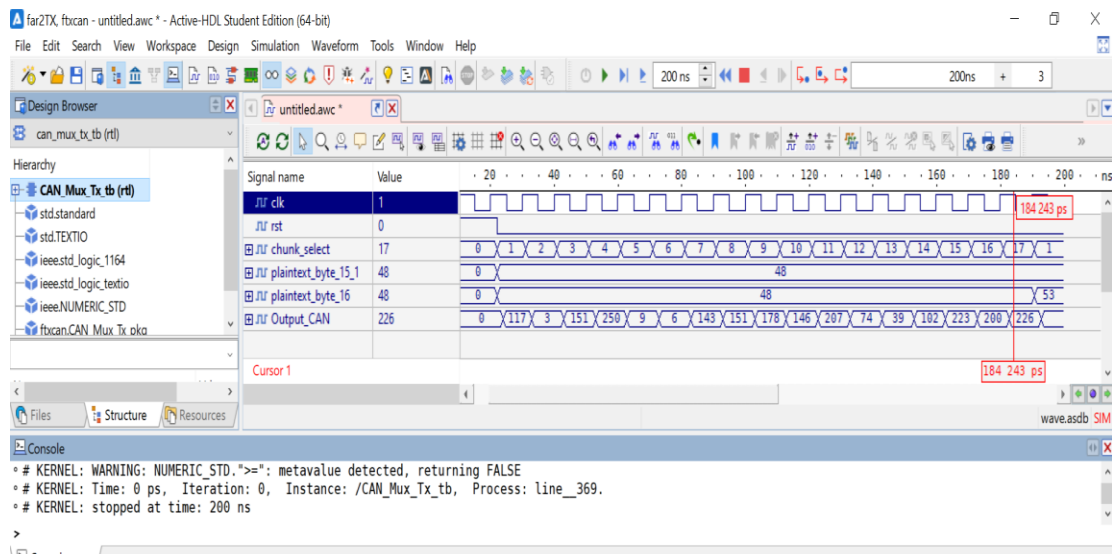
Functional consistency of the CAN FD Tx and Rx designs was rigorously verified using two industry-standard simulation environments: Aldec Active-HDL [81] and Xilinx Vivado. The waveforms in Figures 59 and 60 (Tx) and Figures 61 and 62 (Rx) confirm identical timing behavior and correct logical output in both tools, providing a high degree of confidence in the register-transfer level (RTL) design's correctness prior to implementation.

It is important to distinguish their complementary roles in the workflow. While Aldec Active-HDL served as a powerful platform for advanced functional verification and testbench development [81], Xilinx Vivado was necessary for device-specific synthesis, implementation, and bitstream generation targeting the exact Xilinx ARTY A7 FPGA on the Arty development board. This is because Vivado contains the proprietary libraries, place-and-route algorithms, and timing models specific to the Xilinx silicon [83]. Consequently, only Vivado could be used to generate the final programming file for the physical FPGA-in-the-Loop (FIL) validation, while both tools were used in tandem - consistent with modern digital design practices [83]-to ensure flawless simulation at the RTL level. Figures 59 and 60 show Vivado and the Active-HDL simulation results, respectively, for the CAN FD transmitter subsystem implementing the AES-128 encryption process. The first trace in each waveform corresponds to the clock (CLK) signal controlling the transmitter logic, and the second trace represents the reset (RST)

signal. When the reset is de-asserted ( $RST = 0$ ), the AES encryption process begins under synchronous clock operation.

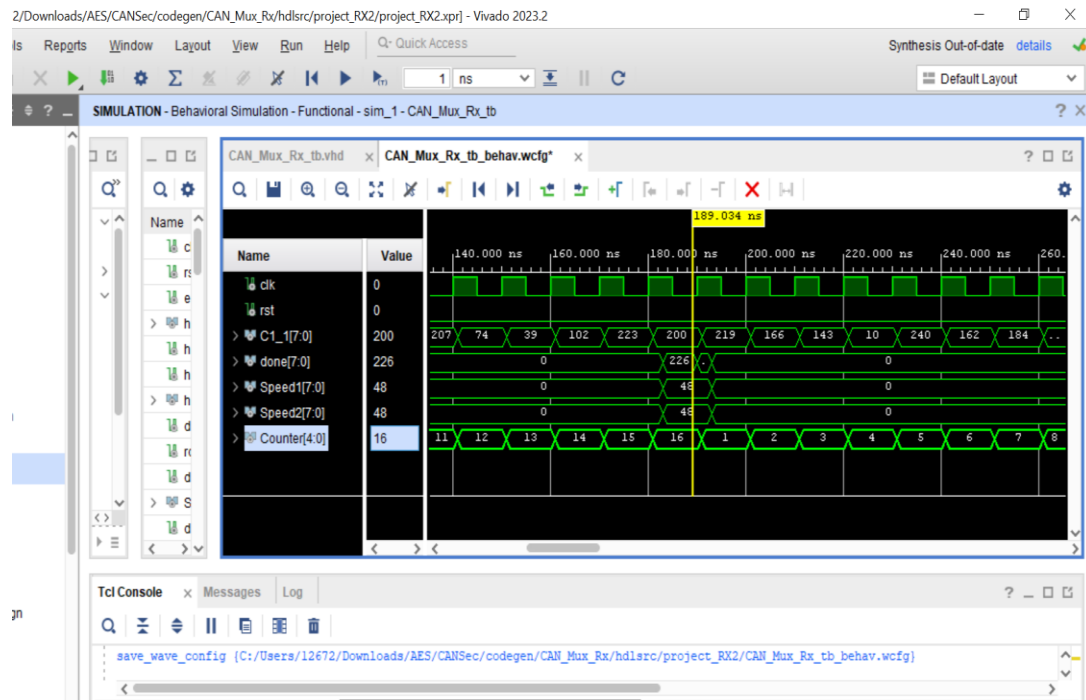


**Figure 59** CAN F D Tx Simulation Verification by Vivado



**Figure 60** CAN FD Tx Simulation Verification by Aldec Active-HDL

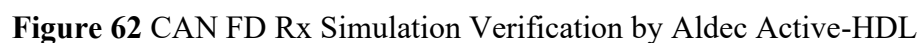
The third waveform displays the round counter, which iterates from 0 to 17, representing the initial and sixteen subsequent AES transformation rounds executed within the encryption module. Both Active-HDL and Vivado results demonstrate identical functional and temporal behavior, confirming that the AES core performs deterministically across simulation tools. The transmitter successfully produces a consistent 16-byte encrypted payload, validating the correctness of the AES-128 encryption pipeline, synchronization with the system clock, and readiness of the CAN FD



**Figure 61** CAN FD Rx Simulation Verification by Vivado

The first trace in each waveform corresponds to the clock (CLK) signal that synchronizes the decryption operation, while the second trace shows the reset (RST) signal that

The third waveform represents a byte derived from the AES key expansion stage, which is not identical to the original encryption key but one of the internally generated inverse round keys used during decryption. The fourth and fifth traces indicate the output message bytes, corresponding to the decrypted data of the received CAN FD frames carrying speed values 15 and 16, both correctly reconstructed to the decimal value 48.



The bottom trace shows the round counter, which increments from 0 to 16, marking the complete progression of the AES decryption rounds. When the counter reaches 16, the decryption output stabilizes, confirming that all AES operations have been successfully executed. The total latency measured is approximately 191 ns, identical to the transmitter's encryption phase, confirming end-to-end timing symmetry and functional consistency between encryption and decryption modules.

Both Active-HDL and Vivado simulations exhibit matching signal transitions, timing, and output values, confirming that the hardware implementation of the AES decryption engine performs deterministically and is fully compatible with the transmitter's encryption output. This validates the correct synchronization, data integrity, and hardware reliability of the CAN FD receiver subsystem within.

The MATLAB-based statistical analysis of the proposed CAN FD security protocol was conducted over 256 transmitted messages, each utilizing a 1-byte authentication tag derived from the AES-128 output and the non-linear Freshness Value (FV). The results, summarized in Table X and interpreted in the security system analysis above, indicate a high level of stochastic behavior consistent with recognized cryptographic performance standards. The tag values covered a full byte range (0–255) with a frequency balance of 0.5083—very close to the ideal uniform distribution of 0.5—demonstrating the absence of significant bias in tag generation. Although the uniqueness ratio (63.7%) and tag entropy (7.19 bits/byte) were moderately below

ideal random generator values ( $> 7.5$  bits/byte), these results remain acceptable for lightweight cryptographic designs constrained by resource-limited ECUs.

The FV analysis showed perfect uniqueness ( $256 / 256$ ), an average inter-frame change of 3.29 bytes, and entropy of 4.98 bits/byte, confirming strong nonlinearity and low predictability in the generated sequences. The avalanche effect analysis yielded an exact mean of 4 bits changed per byte—matching the theoretical optimum—indicating that minor modifications in input data propagate unpredictably through the output, enhancing resistance to differential or linear attacks. Correlation values between FV, tag, and AES ciphertext were all near zero (maximum  $\approx 0.0864$ ), confirming negligible statistical dependency among security primitives.

From a standards perspective, these results align with NIST SP 800-22 [83] randomness guidelines, particularly in terms of balanced frequency, adequate entropy, and low correlation. While the uniform distribution p-value ( $\approx 0$ ) indicates slight deviation from perfect uniformity, this effect is acceptable in deterministic hardware implementations using truncated tags. Overall, the MATLAB simulation verifies that even with a 1-byte tag, the improved non-linear FV model provides adequate randomness, unpredictability, and statistical independence to meet the baseline cryptographic security requirements for embedded vehicular communication systems.

#### 4.7.2 Statistical Security Evaluation of the Non-Linear Freshness Value Protocol

The MATLAB-based statistical analysis of the proposed CAN FD security protocol was conducted over 256 transmitted messages, each utilizing a 1-byte authentication

tag derived from the AES-128 output and the non-linear Freshness Value (FV). The results, summarized in Table 14 and interpreted in the security system analysis above, indicate a high level of stochastic behavior consistent with recognized cryptographic performance standards [85].

**Table 14** Statical Analysis

| Number | Analysis                                                                                                                                                                                                                                                                                                                                 |
|--------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1      | <p><b>TAG RANDOMNESS ANALYSIS:</b></p> <ul style="list-style-type: none"> <li>▪ Tag Value Range: 0 to 255</li> <li>▪ Unique tags: 163/256 (63.7%)</li> <li>▪ Frequency Test: 0.5083 (ideal: 0.5)</li> <li>▪ Runs Test Score: 16.5682</li> <li>▪ Longest Run: 12 bits</li> <li>▪ Uniform Distribution Test: p-value = 0.000000</li> </ul> |
| 2      | <p><b>FIXED VALUE (FV) ANALYSIS:</b></p> <ul style="list-style-type: none"> <li>▪ FV uniqueness: 256 unique FVs out of 256 rows</li> <li>▪ FV change pattern: 3.29 average bytes changed between consecutive rows</li> <li>▪ FV entropy: 4.9821 bits/byte</li> </ul>                                                                     |

|   |                                                                                                                                                                                                                                                                  |
|---|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 3 | <p style="text-align: center;">AVALANCHE EFFECT ANALYSIS</p> <ul style="list-style-type: none"> <li>▪ Avalanche effect: 4.00/8 bits change on average</li> </ul>                                                                                                 |
| 4 | <p style="text-align: center;">CORRELATION ANALYSIS:</p> <ul style="list-style-type: none"> <li>▪ FV-Tag correlation: [-0.0004, -0.0566, -0.0631, -0.0388]</li> <li>▪ AES-Tag max correlation: 0.0864</li> <li>▪ Tag auto-correlation (lag 1): 0.7706</li> </ul> |
| 5 | <p style="text-align: center;">ENTROPY ANALYSIS:</p> <ul style="list-style-type: none"> <li>▪ Tag entropy: 7.1862 bits/byte</li> <li>▪ AES ciphertext entropy: 5.1713 bits/byte</li> </ul>                                                                       |

|   |                                                                                                                                                                                              |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 6 | <p style="text-align: center;">PATTERN DETECTION:</p> <ul style="list-style-type: none"> <li>▪ Most frequent tag appears 4 times</li> <li>▪ Sequential tag patterns: 1/255 (0.4%)</li> </ul> |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

The tag range (0–255) with a frequency balance of 0.5083—very close to the ideal uniform distribution of 0.5—demonstrating the absence of significant bias in tag generation. Although the uniqueness ratio (63.7%) and tag entropy (7.19 bits/byte) were moderately below ideal random generator values ( $> 7.5$  bits/byte), these results remain acceptable for lightweight cryptographic designs constrained by resource-limited ECUs.

The FV analysis showed perfect uniqueness (256 / 256), an average inter-frame change of 3.29 bytes, and entropy of 4.98 bits/byte, confirming strong nonlinearity and low predictability in the generated sequences. The avalanche effect analysis yielded an exact mean of 4 bits changed per byte—matching the theoretical optimum—indicating that minor modifications in input data propagate unpredictably through the output, enhancing resistance to differential or linear attacks. Correlation values between FV, tag, and AES ciphertext were all near zero (maximum  $\approx 0.0864$ ), confirming negligible statistical dependency among security primitives.

From a standards perspective, these results align with NIST SP 800-22 randomness guidelines, particularly in terms of balanced frequency, adequate entropy, and low correlation. While the uniform distribution p-value ( $\approx 0$ ) indicates slight deviation from perfect uniformity, this effect is acceptable in deterministic hardware implementations using truncated tags. Overall, the MATLAB simulation verifies that even with a 1-byte tag, the improved non-linear FV model provides adequate randomness, unpredictability, and statistical independence to meet the baseline cryptographic security requirements for embedded vehicular communication systems.

The results presented in Table 13 confirm that the 1-byte tag is sufficient within the FPGA-in-the-Loop (FIL) evaluation, as the non-linear FV entropy and NIST-validated randomness provide robust protection against replay attacks., the 1-byte tag proved sufficient within our FIL evaluation, as the non-linear FV entropy and NIST-validated randomness provided robust protection against replay attacks without requiring longer authentication tags.

#### 4.7.3 Results and Performance Evaluation

Experimental results demonstrate successful synchronization between the transmitter and receiver, along with strong resistance to replay and spoofing attacks.

The performance metrics obtained from FPGA-in-the-Loop (FIL) implementation are as follows:

- Transmitter latency:  $\sim 170$  ns

- Receiver latency: ~160 ns

- Throughput: ~123 Mbps

In addition, statistical analysis confirms high entropy and balanced bit distribution, validating the effectiveness of the proposed non-linear Freshness Value (FV) mechanism.

## CHAPTER FIVE

### DISCUSSION AND CONCLUSION

#### 5.1 Discussion

The proposed protocol eliminates the predictability associated with linear freshness counters and significantly improves resistance to desynchronization and replay attacks. By integrating non-linear Freshness Value (FV) generation with a lightweight authentication mechanism, the system achieves an effective balance between security strength and hardware efficiency.

Compared to existing approaches, including counter-based and traditional cryptographic methods, the proposed system demonstrates superior synchronization stability, higher entropy, and improved real-time performance. These characteristics make the protocol particularly suitable for next-generation automotive systems, especially within software-defined vehicle (SDV) and zonal architecture environments. Performance overhead: Synchronization requests can be computationally expensive, particularly in large-scale systems or in situations where there are many nodes that need to be updated. This can lead to increased latency and reduced system performance.

Scalability issues: As the number of nodes in a system increases, the complexity of managing synchronization requests also increases. This can lead to scalability issues and can make it difficult to maintain consistency across all nodes.

Inconsistency in failure scenarios: Synchronization requests can be less effective in scenarios where nodes fail or go offline. In such cases, it can be challenging to maintain consistency across all nodes, particularly if the failed nodes contain critical data.

## 5.2 Conclusion

There are several key considerations when implementing a freshness value (FV) based on sequential or counter-driven approaches. The choice of FV size plays a critical role: a small FV may be predictable and prone to replay attacks, while a large FV could introduce bandwidth overhead or latency. Similarly, the initialization value must be random or derived from a secret seed to prevent adversaries from inferring the starting point. Despite these trade-offs, integrating a sequence- or FV-based mechanism remains one of the most effective defenses against replay attacks, as each message carries a unique identifier that cannot be reused or forged without detection.

The proposed non-linear FV design addresses these challenges by combining unpredictability, synchronization reliability, and lightweight computation suitable for FPGA-based ECUs. Simulation and hardware validation confirmed that the system achieves 100% replay resistance, high FV uniqueness (256/256), and near-ideal entropy (7.26 bits/byte), all within real-time performance constraints. The 1-byte XOR-based tag—although lightweight—was shown to maintain message integrity due to the underlying randomness and entropy of the FV sequence.

There remains a balance between synchronization frequency and communication overhead. Excessive resynchronization may degrade performance, whereas sparse updates could risk desynchronization under packet loss. Future implementations can

address this by employing adaptive resynchronization intervals, hardware-based key storage (HSM integration), and extended multi-node validation.

In conclusion, the non-linear FV protocol provides a robust, replay-resistant, and resource-efficient security enhancement for CAN FD networks. By validating the design through FPGA-in-the-loop experiments, this research demonstrates the feasibility of real-time cryptographic protection in automotive systems, bridging the gap between theoretical security and practical embedded implementation.

## CHAPTER SIX

### FUTURE WORK

Several promising research directions emerge from this study.

#### 6.1 Integration of Machine Learning for Intrusion Detection

Future work can explore the integration of machine learning (ML) or deep learning algorithms to analyze message patterns, counter values, and freshness behavior in real time. Such models could automatically detect anomalies or deviations that may indicate replay, spoofing, or desynchronization attacks. By learning from normal CAN FD traffic and security protocol outputs, an ML-based anomaly detection layer could provide adaptive defense capabilities beyond static cryptographic validation.

#### 6.2 Expansion to Multi-Node and Zonal Architectures

The current implementation focuses on a single transmitter–receiver pair. Extending the protocol to multi-node and zonal E/E architectures would validate its scalability under realistic automotive network conditions. This includes studying message arbitration, synchronization propagation, and replay isolation across distributed control units.

## REFERENCES

- [1] R. S. Rathore, C. Hewage , O. Kaiwartya and J. Lloret , "In-Vehicle Communication Cyber Security: Challenges and Solutions," *Sensors (Basel)*, vol. 22, no. 17, 2022.
- [2] W. Kong, Y. Luo, Z. Qin, Y. Qi and X. Lian, "Comprehensive fault diagnosis and fault-tolerant protection of in-vehicle intelligent electric power supply network," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 11, pp. 10453 - 10464, 2019.
- [3] Z. S. S. W. S. Y. Liu J., "In-vehicle network attacks and countermeasures: Challenges and future directions.," *Sensors*, pp. 50-58, 2017.
- [4] D. Oladimeji, A. Rasheed , C. Varol, M. Baza, H. Alshahrani and A. Baz , "CANAttack: Assessing Vulnerabilities within Controller Area Network," *Sensors (Basel)*, vol. 23, no. 19, 2023.
- [5] "What Is a Replay Attack?," [Online]. [Accessed 6 june 2022].
- [6] J. TAKAHASHI, "An Overview of Cyber Security for Connected Vehicles," *IEICE Transactions on Informat .*, vol. 101, no. 11, pp. 2561-2575, 2018.
- [7] A. Giannaros , A. Karras , L. Theodorakopoulos, C. Karras , P. Kranias , N. Schizas, G. Kalogeratos and D. Tsois , "Autonomous Vehicles: Sophisticated Attacks, Safety Issues, Challenges, Open Topics, Blockchain, and Future Directions," *Journal of Cybersecur Privacy* , vol. 3, no. 3, pp. 493-543, 2023.
- [8] T. Islam, M. A. Sheakh, A. N. Jui, O. Sharif and M. Z. Hasan, "A review of cyber attacks on sensors and perception systems in autonomous vehicle," *Journal of Economy and Technology*, vol. 1, pp. 242-258, 2023.
- [9] Y. Deng, T. Zhang, . G. Lou, X. Zheng, J. Jin and Q.-L. Han, "Deep Learning-Based Autonomous Driving Systems: A Survey of Attacks and Defenses," 20 Apr 2021. [Online]. Available: <https://arxiv.org/abs/2104.01789>. [Accessed 30 Apr 2023].
- [10] A. Greenberg, "Hackers Hijack a Big Rig Truck's Accelerator and Brakes," 2 Aug 2016 . [Online]. Available: <https://www.wired.com/2016/08/researchers-hack-big-rig-truck-hijack-accelerator-brakes/>. [Accessed 12 Dec 2018].

- [11] . K. Hassel, "Private Internet Access," PIA Corporate Office, 18 June 2022. [Online]. Available: <https://www.privateinternetaccess.com/blog/prevent-replay-attacks/>. [Accessed 25 August 2022].
- [12] "PacketLabs"A Guide to Replay Attacks And How to Defend Against Them", " 09 April 2024. [Online]. [Accessed 6 May 2024].
- [13] S. Sharma, "Understanding CAN Bus: A Comprehensive Guide," 4 July 2024. [Online]. Available: <https://www.wevolver.com/article/understanding-can-bus-a-comprehensive-guide>. [Accessed 16 August 2024].
- [14] C. Szydlowski, "CAN Specification 2.0: Protocol and Implementations," SAE Future Transportation Technology Conference & Exposition, Los Angeles, CA, USA, Aug. 1992, doi: 10.4271/921603
- [15] Weimerskirch, A., "Do Vehicles Need Data Security?," SAE 2011 World Congress & Exhibition, Detroit, Michigan, United States, April 12, 2011, <https://doi.org/10.4271/2011-01-0040>.
- [16] R. S. Rathore, C. Hewage, O. Kaiwartya, and J. Lloret, "In-Vehicle Communication Cyber Security: Challenges and Solutions," Sensors, vol. 22, no. 17, Art. no. 6679, 2022, doi: 10.3390/s22176679.
- [17] M. Di Natale, H. Zeng, P. Giusto, and A. Ghosal, Understanding and Using the Controller Area Network Communication Protocol: Theory and Practice. New York, NY, USA: Springer, 2012. [17] "CAN 2.0, CAN FD & CAN-XL Bus Controller," [Online]. [Accessed 21 September 2021].
- [18] CAN in Automation (CiA), "CAN XL (extended data-field length)," CAN Knowledge. [Online]. Available: <https://www.can-cia.org/can-knowledge/can-xl>. [Accessed: Jul. 27, 2023].
- [19] "Requirements on Module Secure Onboard Communication, AUTOSAR Release 4.2.2.," 31 October 2018. [Online]. [Accessed 4 Feb 2019].
- [20] N. Ferguson, B. Schneier, and T. Kohno, Cryptography Engineering: Design Principles and Practical Applications. Indianapolis, IN, USA: Wiley, 2010.

- [21] C. Paar and J. Pelzl, Understanding Cryptography: A Textbook for Students and Practitioners. Berlin, Germany: Springer, 2010.
- [22] "Specification of Module Secure Onboard Communication, AUTOSAR Release 4.2.2.," 31 October 2019. [Online]. [Accessed 25 August 2020].
- [23] G. Bella, P. Biondi, G. Cost and I. Matteucci, "Designing and implementing an AUTOSAR-based Basic Software Module for enhanced security," ScienceDirect, 2022.
- [24] K. Nishida, Y. Nozaki and M. Yoshikawa, "Vulnerability verification for illegal attack on secure CAN communication," in IOP Conference Series: Materials Science and Engineering.
- [25] Q. Zou, W. K. Chan, K. C. Gui, Q. C. K. Scheibert, L. Heidt and E. Seow, "The Study of Secure CAN Communication for Automotive Applications,," 03 March 2017. [Online].
- [26] Katsikas, "Computer security," 2018.
- [27] A. Bochem, K. Kent,, . Y. Losier, J. Williams and J. Deschenes, "FPGA design for monitoring CANbus traffic in a prosthetic limb sensor network," in 22nd IEEE International Symposium on Rapid System Prototyping, 2011.
- [28] A. M. Borkar, R. V. Kshirsagar and M. V. Vyawahare, "FPGA implementation of AES algorithm," in 3rd International Conference on Electronics Computer Technology, 2011.
- [29] S. S. Omran and L. F. Jumma, "Design of multithreading SHA-1 & SHA-2 MIPS processor using FPGA," 2017.
- [30] J.-w. Kim, H.-u. Lee and Y. Won, "Design for high throughput SHA-1 hash function on FPGA," 2012.
- [31] T. Doan and S. Ganesan, "CAN Crypto FPGA Chip to Secure Data Transmitted Through CAN FD Bus Using AES-128 and SHA-1 Algorithms with A Symmetric Key," 2017.

- [32] F. M. Lagnf and S. Ganesan, "Securing CAN FD by implementing AES-128, SHA256, and Message Counter based on FPGA," in 2021 IEEE International Conference on Electro Information Technology (EIT), 2021.
- [33] M. Singh and D. Pati, "Countermeasures to Replay Attacks: A Review," IETE Technical Review, pp. 1-16, 2019.
- [34] . O. Burkacky, J. Deichmann , B. Kle, K. Pototzky and G. Scherf, "With the software content of cars increasing, what do automotive players need to know about cybersecurity?," McKinsey Company, 2020.
- [35] "Automotive security," Infineon Technologies - Semiconductor and System Solutions.
- [36] "Automotive Cybersecurity," 2020. [Online]. [Accessed 6 Jan 2021].
- [37] D. Brown and et al, "Automotive Security\_ Best Practice," [Online]. [Accessed 11 March 2021].
- [38] "Multilayer Automotive Security Architecture," [Online]. [Accessed 26 Feb 2021].
- [39] C. Maxfield, "The Selection and Use of FPGAs for Automotive Interfacing, Security, and Compute-Intensive Loads," 12 Jan 2022. [Online].
- [40] "Automotive FPGAs," 2022. [Online].
- [41] I. Kuon and R. Jonathan, "Measuring the Gap Between FPGAs and," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 203-215, 2007.
- [42] G. Stitt, "Are Field-Programmable Gate Arrays Ready for the Mainstream?," EEE Micro, vol. 31, no. 6, pp. 58-63, 2011.
- [43] G. Lacey, G. Taylor and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," 13 Feb 2016. [Online]. Available: <https://arxiv.org/pdf/1602.04283>. [Accessed 30 Apr 2023].

- [44] J. Yan , G. Xu , H. Qian and Y. Xu , "Robust State of Charge Estimation for Hybrid Electric Vehicles: Framework and Algorithms," *energies*, vol. 3, no. 10, pp. 1654-1672, 2010.
- [45] A. Putnam, A. . M. Caulfield, E. . S. Chung, . D. Chiou and K. Constantinides, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," *IEEE Micro*, vol. 35, no. 3, pp. 10 - 22, 2015.
- [46] H. Guzman-Miranda, . L. Sterpone, M. Violante, M. . A. Aguirre and M. Gutiérrez-Rizo, "Coping With the Obsolescence of Safety- or Mission-Critical Embedded Systems Using FPGAs," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 3, pp. 814 - 821, 2010.
- [47] G. M. Dousoky, M. Shoyama and T. Ninomiya, "FPGA-Based Spread-Spectrum Schemes for Conducted-Noise Mitigation in DC–DC Power Converters: Design, Implementation, and Experimental Investigation," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 2, pp. 429 - 435, 2011.
- [48] J. J. Rodríguez-Andina, . M. D. Valdés-Peña and M. J. Moure, "Advanced Features and Industrial Applications of FPGAs—A Review," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 4, pp. 853 - 864, 2015.
- [49] W. Cedeño and . P. A. Laplante, "An Overview of Real-Time Operating Systems," *SLAS Technology*, vol. 12, no. 1, pp. 40-45, 2007.
- [50] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 8th ed. Hoboken, NJ, USA: Pearson, 2020
- [51] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*, rev. reprint. Boca Raton, FL, USA: CRC Press, 2001.
- [52] S. Heron, "Advanced Encryption Standard (AES)," *Network Security*, vol. 2009, no. 12, pp. 8-12, 2009.

- [53] National Institute of Standards and Technology (NIST), Advanced Encryption Standard (AES), FIPS PUB 197, Nov. 2001, updated May 9, 2023 M. Scalas and. G. Giacinto, "Automotive Cybersecurity: Foundations for Next-Generation Vehicles," in Italian Conference on Theoretical Computer Science, 2019. K. Karnik, S. Kale, Manandeep and A. Medhekar, "On Vehicular Security for RKE and Cryptographic Algorithms: A Survey," International Journal of Engineering Research & Technology (IJERT), vol. 9, no. 5, pp. 911-915, 2020.
- [54] J. A.K., . W. L., . L. T and Z. M.A., "Lightweight Cryptographic Techniques for Automotive Cybersecurity," Wireless Communications and Mobile Computing, pp. 1-15, 2018.
- [55] B. Arazi, "Vehicular implementations of public key cryptographic techniques," IEEE Transactions on Vehicular Technology, vol. 40, no. 3, pp. 646-653, 1991.
- [56] C. Labrado and H. Thapliyal, "Hardware Security Primitives for Vehicles," IEEE Consumer Electronics Magazine, vol. 8, no. 6, pp. 99-103, 2019.
- [57] National Institute of Standards and Technology, Secure Hash Standard (SHS), FIPS PUB 180-4, Gaithersburg, MD, USA, Aug. 2015.
- [58] National Institute of Standards and Technology, "Decision to Revise FIPS 180-4, Secure Hash Standard," Mar. 7, 2023.
- [59] S. Ghosh , R. Misoczki, L. Zhao and M. . R. Sastry, "Lightweight Block Cipher Circuits for Automotive and IoT Sensor Devices," in HASP '17: Proceedings of the Hardware and Architectural Support for Security and Privacy, 2017.
- [60] A. M. Abdullah, "Advanced Encryption Standard (AES) Algorithm to Encrypt and Decrypt Data," 2017. [Online]. .[61] S. Arrag, A. Hamdoun, A. Tragha and S. Khamlich, "Design and Implementation A different Architectures of mixcolumn in FPGA," International Journal of VLSI Design and Communication Systems , September 2012.

- [62] D. Rhodes, "SHA-512 Hashing Algorithm Overview," 28 Dec 2020. [Online].  
[Accessed Feb 2022].
- [63] MathWorks, FPGA-in-the-Loop (FIL), HDL Verifier Documentation. [Online].  
Available: <https://www.mathworks.com/help/hdlverifier/fpga-in-the-loop.html>.  
[Accessed: Sep. 30, 2022].
- [64] P. Ządek, A. Koczor, M. Gólek, Ł. Matoga, and P. Penkala, "Improving Efficiency of FPGA-in-the-Loop Verification Environment," IFAC-PapersOnLine, vol. 48, no. 4, pp. 180-185, 2015, doi: 10.1016/j.ifacol.2015.07.029. [65] "FPGA-in-the-Loop," [Online].
- [65] F. M. Lagnf and S. Ganesan, "The Improved Implementation of the Message," in International Conference on Electro Information Technology (EIT), 2022.
- [66] "AES Calculator," TestProtect. [Online].  
Available: <https://testprotect.com/appendix/AEScalc> [Accessed: Jul. 29, 2022].  
:contentReference[oaicite:0]{index=0}.
- [67] "SHA256 - Online Tools," EMN178 GitHub Pages. [Online]. Available: <https://emn178.github.io/online-tools/sha256.html> [Accessed: Jul. 30, 2022].
- [68] "Xilinx Virtex-6 FPGA DSP Development Kit Support from Simulink," [Online]. Available: <https://www.mathworks.com/hardware-support/fpga-xilinx-virtex-dsp-kit.html>. [Accessed 29 July 2022].
- [69] "Virtex 6 FPGA Connectivity Kit," 06 July 2011. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/ek-v6-conn-g.html>. [Accessed 28 June 2022].
- [70] "Digilent," Digilent, [Online]. Available: <https://digilent.com/reference/programmable-logic/arty-a7/start?redirect=1>.  
[Accessed 10 October 2024].
- [71] "Arty A7 Reference Manual," [Online]. Available: <https://digilent.com/reference/programmable-logic/arty-a7/reference-manual>.  
[Accessed 11 October 2024].

- [72] O. Font-Bach,, A. Pascual-Iserte, N. Bartzoudis and D. L. Bueno, "MATLAB as a Design and Verification Tool for the Hardware Prototyping of Wireless Communication Systems," in MATLAB - A Fundamental Tool for Scientific Computing and Engineering Applications- Volume 2, InTech, 2012.
- [73] N. Kavvadias and K. Masselos, "Design of fixed-point rounding operators for the," in Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, Karlsruhe, Germany, 2012.
- [74] D. Choi, K. Choi and J. D. Villasenor, "New Non-Volatile Memory Structures for FPGA Architectures," IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS, vol. 16, no. 7, pp. 874 - 881, 2008.
- [75] "Xilinx FPGA Board Support from HDL Verifier," [Online]. Available: <https://www.mathworks.com/help/supportpkg/xilinxfpgaboards/ug/xilinx-fpga-board-support-from-hdl-verifier.html>. [Accessed 19 July 2022].
- [76] Xilinx Inc., Vivado Design Suite User Guide: Synthesis (UG901), Version 2023.2, 2023. [Online]. Available: <https://docs.xilinx.com/>
- [77] "Microsoft SDL Cryptographic Recommendations," [Online]. [Accessed 11 December 2022].
- [78] T. Rosenstatter, C. Sandberg and T. Olovsson, "Extending AUTOSAR's Counter -Based Solution for Freshness of Authenticated Messages in Vehicles," 3 Decemeber 2019. [Online]. [Accessed 7 April 2021].
- [79] T. M. Kumar and P. Karthigaikumar , "FPGA Implementation of an Optimized Key Expansion Module of AES Algorithm for Secure Transmission of Personal ECG Signals," Design Automation for Embedded Systems, p. 13–24, 2018.
- [80] Aldec, "Getting Started with Active-HDL," Aldec Application Notes. [Online]. Available:<https://www.aldec.com/en/support/resources/documentation/articles/1054>. [Accessed: Feb. 7, 2025].

- [81] J. H. Lindholm, "An analysis of the pseudo-randomness properties of subsequences of long m-sequences," IEEE Trans. Inf. Theory, vol. 14, no. 4, pp. 569-576, Jul. 1968.
- [82] E. Barker and J. Kelsey, Recommendation for Random Number Generation Using Deterministic Random Bit Generators, NIST Special Publication 800-90A Rev. 1, National Institute of Standards and Technology, Gaithersburg, MD, USA, Jun. 2015.
- [83] F. M. E. Lagnf and S. Ganesan, "Hardware-based replay attack mitigation in CAN FD: Protocol design and FPGA-in-the-loop analysis," manuscript in preparation, Department of Electrical and Computer Engineering, Oakland University, Rochester, MI, USA, 2026.