International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 11 | Nov 2022 www.irjet.net p-ISSN: 2395-0072

Implementation of CAN on FPGA for Security Evaluation Purpose

Rohit Pawar

Student, Dept. of Electronics Engineering,
Shri Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, India

---***---

Abstract - *Controller Area Network (CAN) is a low-cost communication protocol which is being used in various applications like automotive, medical, military, aviation, etc. Current CAN applications are based on the standard CAN 2.0 protocol. It was not designed for secure communication; although, it offers built-in error detection, robustness, speed and flexibility, the security side of CAN communication is highly underdeveloped. Research on in-vehicle CAN security primitives is difficult as it is hard to get a real vehicle for evaluation and development of these security primitives. The cost of research is considerably high, for some researchers this creates a barrier, acting as a deterrence. This paper presents implementation of CAN on FPGA and evaluation of cryptographic algorithms as a security measure in CAN bus communication.*

Key Words: FPGA, CAN bus, SJA1000, Xilinx Zynq, PetaLinux, ZYBO, Evaluation, Security.

1. INTRODUCTION

If we consider only automobiles, there could be as many as 100 Electronic Control Units (ECUs) all communicating through a single CAN bus system. Despite the functional benefits, the CAN bus system is vulnerable to cyber-attacks like replay attacks, denial-of-service, and man-in-the-middle attacks. Security of any network can be assessed on three attributes: *confidentiality, integrity, and authentication*. Many researchers have demonstrated various successful attacks which showed that CAN bus fails in all these three parameters [2,17,13]. Manufacturers understand these vulnerabilities and have started implementing security measures like, Network Segmentation, Intrusion Detection System, Software-based encryption methods. But all these are computation intensive, have their own vulnerabilities, and are slower, thus reducing the data transfer rate of such a time-critical system. And as pointed out by [3], considering the lifetime of a vehicle, it is possible to find vulnerabilities in above mentioned measures and crack a static encryption key. For high stake applications like military-equipment, vehicles, aircraft such vulnerabilities are not acceptable. A lot of research has been done in securing CAN communications [14-18]; however, these proposed solutions have their own limitations and vulnerabilities. Developing a system that is able to meet resource constraints put by CAN communication system is a real challenge. Modern day ECUs are relatively efficient and fast compared to the ones used a decade ago, hence some of the solutions proposed by some authors could be a viable option with some performance enhancing tweaks. However, a thorough evaluation of these proposed methods is needed in order to explore the areas of improvements. This is a laborious, complicated and time-consuming process, to streamline this process a versatile test-bed is needed that speeds up the entire chain of implementation, evaluation, and development and deployment. FPGAs based solutions can help to address these challenges by enabling true flexibility and scalability to address the security requirements of CAN bus with inherent hardware and software programmability. Rather than creating a bare-metal application to run on FPGA emulated CAN controller, we rely on a test-bed that runs a light-weight Linux kernel on top. Bare-metal application are difficult to debug as there are no fault management and error notifications unless they have been explicitly implemented and validated. Given the scope of our application it is difficult to figure out what exactly could go wrong, hence the need of a test-bed that has error handling and prompting abilities is needed, this allows for speedy debugging and evaluation of the proposed solutions, and speed up the development of new ones. In our experimental test-bed, we have used OpenCores SJA1000 controller, that has been modified to ignore CAN FD frames. This version of SJA1000 is modified by Czech Technical University, Prague (CTU) [1] and allows this controller to co-exist and send data on CAN FD network.

2. SYSTEM DESIGN

The test-bed is built in two parts: hardware and software. The design of hardware is done using Xilinx Vivado Design Suite and is implemented on ZYBO (ZYnq BOard) that uses Zynq All Programmable System on Chip Architecture. The SoC design has two subsystems: Processing System (PS) and Programmable Logic (PL). The PS, is Zynq-7000, has software programmability features and takes over the job of controlling and communicating with PL. PL is Xilinx Artix-7 FPGA with configurable ports and buses, and has hardware modules that are being emulated. The software design is done with the help of PetaLinux SDK to streamline embedded Linux development. This tool helps us to develop a Bootable System Image that consists of a custom CPU-optimized Linux kernel, device drivers and bootloader configurations. The test-bed is now ready for further implementation, evaluation and development of security measures for CAN bus communications. This section further discusses the important elements of hardware and software design.

2.1 Hardware Design

The hardware consists of SJA1000 controller, TJA1050 CAN trans-receiver, Zynq 7000 SoC, and other periphery circuits required for implementation of test-bed. Multiple instances of this test-bed can be used to connect and communicate on a CAN bus line for analysis and evaluation of different security measures. For the demonstration of communication between two CAN nodes; Node$_1$ and Node$_2$ over the CAN bus, we perform a simple signal detection experiment. When input is given to any one of the nodes by pressing the switch, for instance consider Node$_1$, CAN communication is triggered and a data-frame corresponding to the signal is transmitted over the CAN bus, this signal data-frame is picked up by Node$_2$, it interprets it and the LED assigned to the switch, turns on or off. i.e., the LED of one node is controlled by another node. Fig. 1 shows the block diagram of a CAN node.

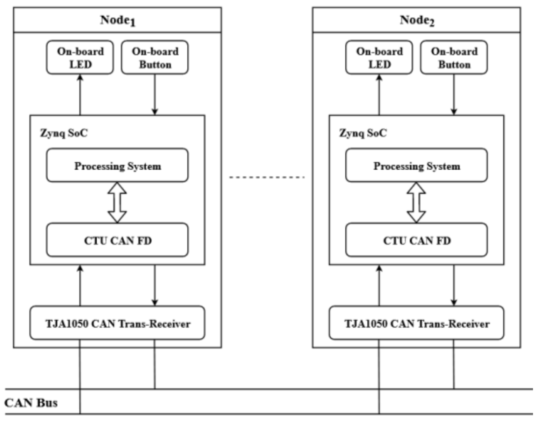


Fig -1: Block Diagram of Test-bed.

2.1.1 SJA1000

It is a stand-alone controller for the Controller Area Network (CAN), mostly used within automotive and general industrial environments. It has both CAN 2.0A (Standard CAN) and 2.0B (Extended CAN) protocol support, with bitrates up to 1 Mb/s. The SJA1000 that is used in this project is CAN FD tolerant and is written in Verilog language [1]. It contains transmission buffer (TXB), receiving buffer (RXB), bit timing logic (BTL), acceptance filter (ACF), bit stream processor (BSP), error management logic (EML), and interface management logic (IML). It uses an APB interface for on chip communication with master (i.e., the processing system). SJA1000 appears to a master as a memory-mapped I/O device. Independent operation of both devices is guaranteed by a RAM-like implementation of its registers. Fig. 2 shows schematic diagram of the implemented SJA1000 CAN controller.

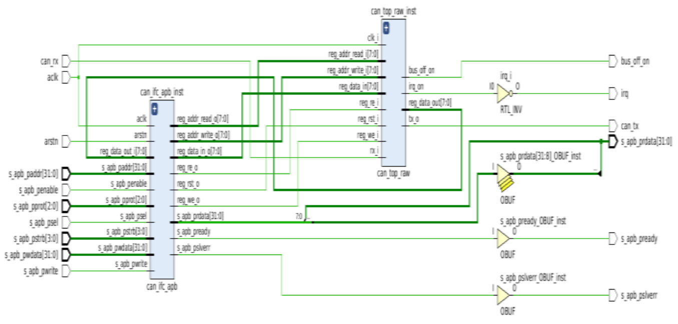


Fig -2: Schematic of Implemented SJA1000 CAN Controller

2.1.2 TJA1050 CAN transreceiver

It is an interface between CAN controller and physical bus. It provides differential transmit and differential receive capability to the CAN controller. It converts the Tx and Rx signal of controller to CANH and CANL signal. It offers good performance when it comes to optimal matching of output signals (CANH, CANL). The figure below shows block diagram of TJA1050.

2.1.3 Processing System

Zynq-7000 processing system (PS) is the master and an instance of SJA1000 in PL is the slave. Communication between master and slave on the SoC is done through AXI and APB buses respectively. AXI is designed for communication between blocks of IP on FPGA. AXI interconnect IP is used to connect one or more AXI memory-mapped master devices to one or more memory-mapped slave devices. The SJA1000 IP has APB interface, suitable for low-speed communication. An AXI-APB bridge is needed to connect this APB slave to the AXI master. Interrupts coming from the CAN controller is connected to the PL-PS interrupt port of the processing system. Tx and Rx signals of the CAN controller are mapped to IO pins connected to the high speed *Pmod* (peripheral module) ports on the board, it is an open standard defined by Digilent Inc. for connecting peripheral modules to FPGA. The Tx-Rx pair of the controller is converted to CANH and CANL signal by the transceiver module connected to *Pmod* port. Configuration and control of CAN controller is done by the PS using a driver code which accesses the controller using the virtual memory address. Driver program is required for mode selection, reading and writing messages to the controller, etc. A top-level application is required to read sensor values, manage keys and construct messages. A standalone application could be created for this purpose but as discussed earlier our application requires a different approach. The custom Linux kernel in the system has support for Python, C, and C+, this allows for rapid development and deployment of application program. The interface circuit diagram of our system is shown below.

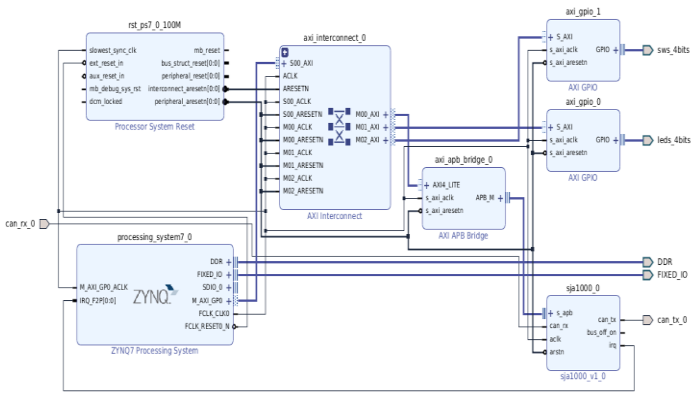


Fig -3: Interface Diagram of the Test-bed not using transreceiver

2.2 Software Design

The processor in the Processing System boots first, this allows for a software centric approach for Programmable Logic configuration which is managed by a program running on the CPU. After it starts, the Linux kernel probes the devices in the system, it gets to know about the devices present in the system from the Device Tree. A device's existence in the system, its location on the bus from which it may be accessed, and its configuration—including its registers, interrupts, and other settings—are all specified in the device tree entry. Device tree entry of SJA1000 is shown below. Field called 'compatible' is used to specify the name of the driver associated to the device, 'interrupts-' field specifies on which pin number the interrupt of device is connected and reg specifies the address of device.

Code Snippet:

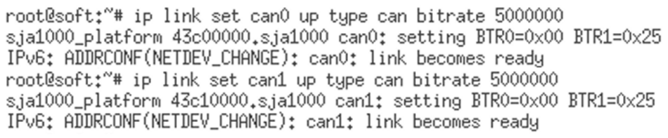
```
sja1000_0: sja1000@43c00000 {
compatible = "nxp, sja1000";
interrupt-names="irq";
interrupt-parent = <&intc>;
interrupts = <0 29 4>;
reg = <0x43c00000 0x10000>;
reg-to-width = <4>;
status = "okay";
nxp, external-clock-frequency = <100000000>;
};
```

The software realization of CAN node communication is done in two parts: Initializing the SJA1000 CAN controller and creating an application program that can access the controller for establishing communication with other nodes on CAN bus.

2.2.1 Initializing SJA1000 CAN controller

The system at this stage is ready to send all data and receive all data using default arbitration ID (0x001), without filtering any messages, owing to the default filter register values of the SJA1000 controller set by the SocketCAN device driver. Without changing the default filter register values, we can directly access the controller through command line interface (CLI) to send/receive data to/from the bus. Another way is to set the values of filter register before initializing the CAN controller. Both of these methods are discussed below.

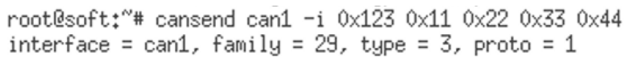
A. Direct access to CAN controller: This can be done at OS level using CLI as well as at application program level. For simplicity, we show this process done at OS level, by executing the following command.



```
root@soft:~# ip link set can0 up type can bitrate 5000000
sja1000_platform 43c00000.sja1000 can0: setting BTR0=0x00 BTR1=0x25
IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
root@soft:~# ip link set can1 up type can bitrate 5000000
sja1000_platform 43c10000.sja1000 can1: setting BTR0=0x00 BTR1=0x25
IPv6: ADDRCONF(NETDEV_CHANGE): can1: link becomes ready
```

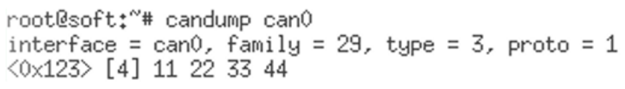
Fig -4: Initializing two instances of SJA1000 CAN controller.

After initialization, the CAN controller can be now be directly accessed to send and receive message, along with a custom or default arbitration ID. The direct access method is helpful when we wish to receive and analyse all the data broadcasted over the CAN bus. The figure below shows the execution.



```
root@soft:~# cansend can1 -i 0x123 0x11 0x22 0x33 0x44
interface = can1, family = 29, type = 3, proto = 1
```

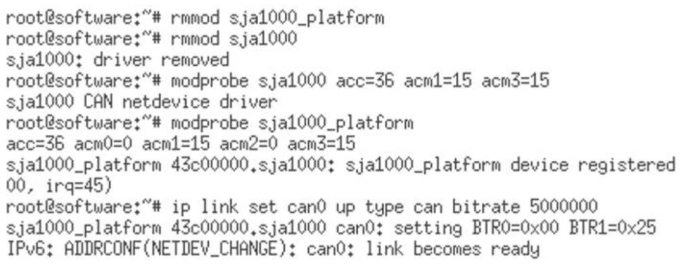
Fig -5: can1 (Node$_2$) sending data over CAN bus using ID (0x123)



```
root@soft:~# candump can0
interface = can0, family = 29, type = 3, proto = 1
<0x123> [4] 11 22 33 44
```

Fig -6: can0 (Node$_1$) receiving data over CAN bus from CAN node having ID (0x123)

B. By changing filter register values: SocketCAN driver does not have support for writing the filter registers of the CAN controller by the user. Since the registers are written at the time of insertion of device driver module and can't be changed frequently, the values of all four Acceptance Code Register (ACR) and all four Acceptance Mask Register (ACR) are hard coded as all zeros and all ones respectively, which means all the messages on the bus will be received and sent in FIFO manner. In the SocketCAN driver, filtering is done at the kernel layer which doesn't make sense for our application since we have actual CAN protocol controller IP on the chip. We have modified he kernel module code, which allows us to change values of filter registers. The shell command below shows the process of changing filter register values after kernel module code modification.



```
root@software:~# rmmod sja1000_platform
root@software:~# rmmod sja1000
sja1000: driver removed
root@software:~# modprobe sja1000 acc=36 acm1=15 acm3=15
sja1000 CAN netdevice driver
root@software:~# modprobe sja1000_platform
acc=36 acm0=0 acm1=15 acm2=0 acm3=15
sja1000_platform 43c00000.sja1000: sja1000_platform device registered
00, irq=45)
root@software:~# ip link set can0 up type can bitrate 5000000
sja1000_platform 43c00000.sja1000 can0: setting BTR0=0x00 BTR1=0x25
IPv6: ADDRCONF(NETDEV_CHANGE): can0: link becomes ready
```

Fig -7: Shows process to change filter register values

2.2.2 Program for sending and receiving message

After the initialization of CAN controller, the system is ready to take part in communication. Apart from directly sending data through shell command, writing an application-based program helps in realizing autonomous communication. The message that is written to or read from CAN data-frame depends on the purpose of application.

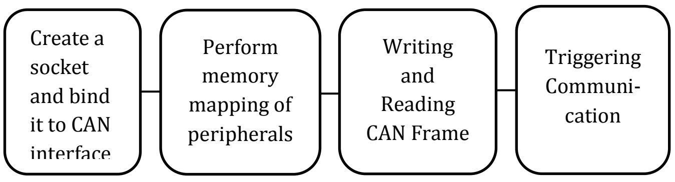

Create a socket and bind it to CAN interface
Perform memory mapping of peripherals
Writing and Reading CAN Frame
Triggering Communi-cation

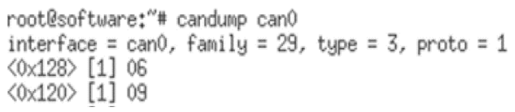
Fig -8: Program design flow

3. SYSTEM TEST

To ensure the functionality and viability of test-bed we carried out following tests.

3.1 Switch status transfer using CAN bus between two nodes

Communication tests were performed with and without using transreceiver for further evaluation. In this test discussed here, instead of using transceiver modules we can emulate the transceiver operation simply by ANDing the Tx signals from the two controllers. For this purpose, we use a two input AND gate. Tx signal of both the controllers are given as inputs to the AND gate and its output is connected to the Rx of both the controllers. This allows for successful arbitration. After the initialization of the system, we run our script in Linux shell of our test-bed. Python application running at the top level accesses the memory mapped GPIOs. It reads sensor values, decides message identifier and attaches the corresponding count value to construct the message according to the protocol. Then it sends the message to the controller through SocketCAN and reads the received message from the controller. The program enables two CAN nodes to communicate. When input is given to Node$_1$ by flipping switch, this input signal is sent over CAN bus and is received by Node$_2$, Node$_2$ then extracts the signal message from CAN data-frame and turns on its LED that is assigned to the switch of Node$_1$. In short, the LEDs of Node$_2$ is controlled by Node$_1$ and vice-versa. Fig.9 shows the demonstration. For this test, acceptance filter register values were changed so that each node accepts data frame from another node and filters out its own data frame.



```
root@software:~# candump can0
interface = can0, family = 29, type = 3, proto = 1
<0x128> [1] 06
<0x120> [1] 09
```

Fig -9. A: Message sent over CAN bus

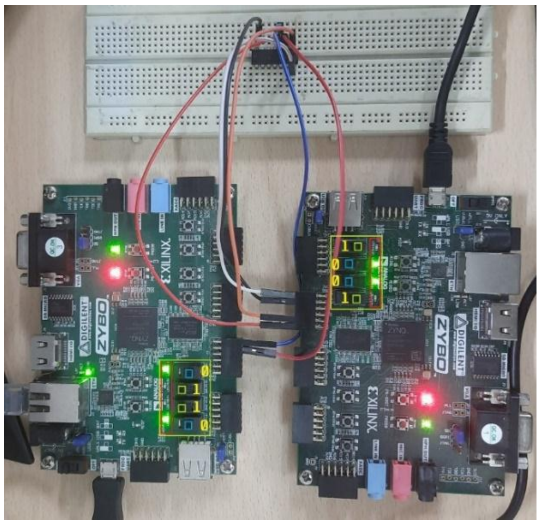
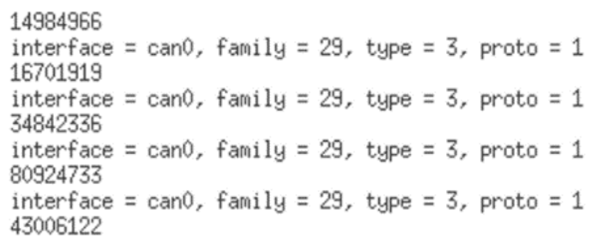


Fig -9. B: Switch status transfer between two CAN nodes

In Fig. 9.A and 9.B CAN nodes on left (Node$_1$) with identifier <0x128> sends message 0x06 to Node$_2$ and CAN nodes on right (Node$_2$) with identifier <0x120> sends message 0x09 to Node$_1$. The received messages contain the status of the switch and depending upon the status, corresponding LEDs are turned on. The switches highlighted in blue are in off state and the ones highlighted in green are in on state, their status can be seen other board.

3.2 Ciphertext over CAN Bus

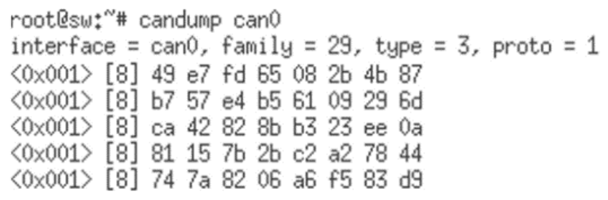
Cryptographic algorithms were evaluated on test-bed for evaluation of its performance. Algorithm were implemented on application layer. A time-delay analysis was performed to see the delay caused by inclusion of these algorithms in communication. This delay included time taken for sending and receiving message over CAN bus, time needed for encryption and decryption of message. An overview of this operation is shown below. For this test a bit-rate 125Kb/s was used for reference and of multiple iterations of test were performed with 1000 random messages in each test.



```
14984966
interface = can0, family = 29, type = 3, proto = 1
16701919
interface = can0, family = 29, type = 3, proto = 1
34842336
interface = can0, family = 29, type = 3, proto = 1
80924733
interface = can0, family = 29, type = 3, proto = 1
43006122
```

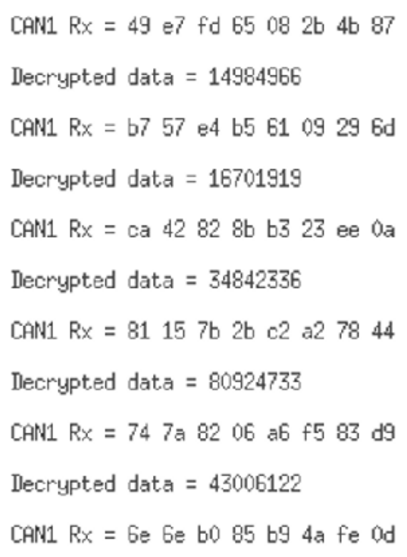
Fig -10. A: Line1 - Message to encrypt

Line 2 - CAN controller of Node₁ invoked after encrypted message passed to it



```
root@sw:~# candump can0
interface = can0, family = 29, type = 3, proto = 1
<0x001> [8] 49 e7 fd 65 08 2b 4b 87
<0x001> [8] b7 57 e4 b5 61 09 29 6d
<0x001> [8] ca 42 82 8b b3 23 ee 0a
<0x001> [8] 81 15 7b 2b c2 a2 78 44
<0x001> [8] 74 7a 82 06 a6 f5 83 d9
```

Fig -10. B: Encrypted message sent on CAN bus by Node₁



```
CAN1 Rx = 49 e7 fd 65 08 2b 4b 87

Decrypted data = 14984966

CAN1 Rx = b7 57 e4 b5 61 09 29 6d

Decrypted data = 16701919

CAN1 Rx = ca 42 82 8b b3 23 ee 0a

Decrypted data = 34842336

CAN1 Rx = 81 15 7b 2b c2 a2 78 44

Decrypted data = 80924733

CAN1 Rx = 74 7a 82 06 a6 f5 83 d9

Decrypted data = 43006122

CAN1 Rx = 6e 6e b0 85 b9 4a fe 0d
```

Fig -10. C: Encrypted message picked up by Node₂ and decrypted to reveal the original message

Fig. 10.A shows the plaintext i.e., the message to be sent to Node₂ by Node₁, this plaintext in converted into ciphertext by cryptographic algorithms and is then sent to CAN controller of Node₁. Fig. 10.B shows the encrypted message (plaintext) or ciphertext on CAN bus sent buy Node₁. Fig. 10.C shows the ciphertext received by Node₂ and is decrypted to reveal the original message (plaintext). The charts below give an overview of the results.

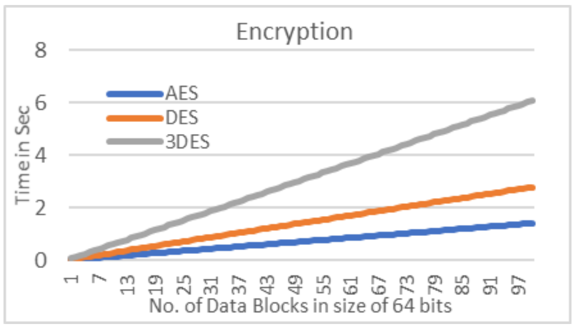


Fig -11. A: Performance analysis during Encryption

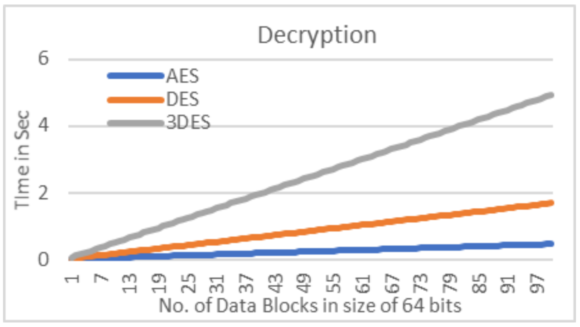


Fig -11. A: Performance analysis during Decryption.

We have tested three different algorithms: Advance Encryption Standard (AES), Data Encryption Standard (DES), Triple Data Encryption Standard (3DES). We are running the algorithms in (Cipher Block Chaining) CBC mode, in this mode block of plaintext are XORed with ciphertext of pervious of previous block and is then encrypted. The difference in encryption and decryption time of these algorithms is because in CBC mode encryption is done sequentially, where each cypher block is dependent on previous block. However, in case of decryption, to decrypt next cypher block there is no need of previous plaintext so decryption can be completely parallelized and is much faster in concurrent systems, like FPGA. The execution time and latency induced in pure software implementation of these algorithms is shown table below.

Table -1 Comparison of overhead because of use of Cryptographic Algorithms.

Sr. No.	Method	Network Capacity	Penalty	Execution Time	Delay Induced
1	No Algorithm	115 Kb/s	8%	10 ms	2 ms
2	AES-128	82 Kb/s	28.57%	14 ms	4 ms
3	DES	38 Kb/s	66.66%	30 ms	20 ms
4	3DES	19 Kb/s	83.60%	61 ms	51 ms

In Table 1, entry 1 shows the actual network load achieved without use of any cryptographic algorithm. AES being the fastest uses more computation power, followed by DES and then 3DES. These implemented algorithms cause an increase in network load because of the limited data size of standard CAN frame which is overcome by sending multiple CAN frames for one message. This may not be a problem for low traffic networks however with the advancement in modern vehicles there is more integration of sophisticated technology (Ex: ADAS) the number of ECUs are only going to go up. Hence, it is imperative to look for a solution that satisfies the time-critical needs of CAN bus communication.

4. PRACTICAL IMPLICATIONS

The test-bed can be used to evaluate different proposed security measures and carry out analysis for areas of improvement. We have demonstrated test-bed's functionality and analysed the performance of cryptographic algorithms in CAN communication. This test-bed is a versatile option that facilitates fast paced development and deployment. Application program can be developed in other system and then can be directly copied into the memory of the test-bed. For any fine tuning or changes needed in program, it can be done directly by accessing the program through the Linux shell in test-bed. This is helpful when iterative changes are made to program. The system does not need to restart every time, all the changes can be made directly to program while the test-bed is online. Moreover, this system gives the ability to debug a program which is not possible when bare-metal application is created, the error handling and prompting ability is just another advantage of this test-bed. This test-bed is intended for evaluation and development of security measures. Once a satisfactory analysis is carried out, later a bare-metal application or RTOS (Real Time Operating System) based solution can be developed to improve execution speed.

5. CONCLUSION & FUTURE SCOPE

The functionality and viability of the test-bed has been verified through different experiments. The test-bed is simple, stable and practical, and thus can be used for further studies and evaluations. Hardware-based solutions are better when it comes to performance and security, but are not versatile. Software based solutions are versatile but effective only when high performance CPUs are available and they induce communication delay. Taking advantage of high level of parallelism offered by FPGA and flexibility of Embedded Systems it is possible to come with a versatile and high-performance security solutions. Based on the findings of subsequent tests, the primary goal would be to concentrate on creating a workable solution for safeguarding CAN Bus communications.

REFERENCES

[1] OpenCores SJA1000 controller. Available at: https://canbus.pages.fel.cvut.cz/

[2] Koscher, Karl, et al. "Experimental security analysis of a modern automobile." *2010 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010.

[3] Bozdal, Mehmet, et al. "Evaluation of can bus security challenges." *Sensors* 20.8 (2020): 2364

[4] Lin, Chung-Wei, and Alberto Sangiovanni-Vincentelli. "Cyber-security for the controller area network (CAN) communication protocol." *2012 International Conference on Cyber Security*. IEEE, 2012.

[5] Szilagy, Chris, and Philip Koopman. "A flexible approach to embedded network multicast authentication." (2008).

[6] Huang, Tianxiang, et al. " ATG: An Attack Traffic Generation Tool for Security Testing of In-vehicle CAN Bus"(2018)

[7] Payne, Bryson R. (2019) "Car Hacking: Accessing and Exploiting the CAN Bus Protocol," *Journal of Cybersecurity Education, Research and Practice*: Vol. 2019 : No. 1 , Article 5

[8] K. Koscher *et al.*, ``Experimental security analysis of a modern automobile,''in *Proc. IEEE Symp. Secur. Privacy*, Oakland, CA, USA, May 2010,pp. 447_462.

[9] (2014). *The Lockheed Martin Cyber Kill Chain*. [Online]. Available :http://cyber.lockheedmartin.com/hubfs/Gaining_the_Advantage_Cyber_Kill_Chain.pdf

[10] H. Gustavsson and J. Axelsson, ``Evaluating _exibility in embedded automotive product lines using real options,'' in *Proc. 12th Int. Softw. Product Line Conf.*, Sep. 2008, pp. 235_242.

[12] K. Koscher *et al.*, ``Experimental security analysis of a modern automobile,'' in *Proc. IEEE Symp. Secur. Privacy*, Oakland, CA, USA, May 2010, pp. 447_462.

[13] S. Woo, H. J. Jo, and D. H. Lee, ``A practical wireless attack on the connected car and security protocol for in-vehicle CAN,'' *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 2, pp. 993_1006, Apr. 2015.

[14] J. Schmandt, A. T. Sherman, and N. Banerjee, ``Mini-MAC: Raising the bar for vehicular security with a lightweight message authentication protocol,'' *Veh. Commun.*, vol. 9, pp. 188_196, Jul. 2017.

[15] H. Abdulmalik and B. Luo, ``Using ID-hopping to defend against targeted DoS on CAN,'' in *Proc. 1st Int. Workshop Safe Control Connected Auton. Vehicles*, 2017, pp. 19_26.

[16] L. Martin, P. Mundhenk, and S. Steinhorst, ``Security-aware obfuscated priority assignment for automotive CAN platforms,'' *ACM Trans. Des Automat. Electron. Syst.*, vol. 21, no. 2, pp. 1_27, 2016.

[17] P.-S. Murvay and B. Groza, ``DoS attacks on controller area networks by fault injections from the software layer,'' in *Proc. 12th Int. Conf.Availability, Rel. Secur.*, 2017, p. 71.

[19] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Keccak Specifications, 2009.

[18] S. Woo, D. Moon, T. -Y. Youn, Y. Lee and Y. Kim, "CAN ID Shuffling Technique (CIST): Moving Target Defense Strategy for Protecting In-Vehicle CAN," in IEEE Access, vol. 7, pp. 15521-15536, 2019, doi: 10.1109/ACCESS.2019.2892961.

[20] Bellare, M., Rogaway, P. (1994). Entity Authentication and Key Distribution. In: Stinson, D.R. (eds) Advances in Cryptology — CRYPTO' 93. CRYPTO 1993. Lecture Notes in Computer Science, vol 773. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-48329-2_21

[21] Elinux.org. Available at: https://elinux.org/index.php?title=Python_Can&action=edit

[22] Digilent, Inc. Available at: https://digilent.com/reference/programmable-logic/zybo/start