ROUTING AND ACTION

MEMORANDUM

ROUTING

TO:(1) Sciences of Extreme Materials Branch (SEM) (Bakas, Michael)

Report is available for review

(2) Proposal Files Report No.: -REP Proposal Number: 78200-SM-REP.9

DESCRIPTION OF MATERIAL

CONTRACT OR GRANT NUMBER: W911NF-21-1-0210

INSTITUTION: California State University - Fresno

PRINCIPAL INVESTIGATOR: Hovannes Kulhandjian

TYPE REPORT: Final Report

DATE RECEIVED: 7/26/23 2:13AM

PERIOD COVERED: 4/26/21 12:00AM through 4/25/23 12:00AM

TITLE: Final Report: Secure Communications and Embedded Systems Laboratory at California State University, Fresno

ACTION TAKEN BY DIVISION

(x) Report has been reviewed for technical sufficiency and IS [x] IS NOT [ ] satisfactory.

( ) Based on my technical review, I have identified no OPSEC or Technology Protection concerns that need to be addressed regarding this report.

(x) Perfomance of the research effort was accomplished in a satisfactory manner and all other technical requirements have been fulfilled.

(x) Based upon my knowledge of the research project, I agree with the patent information disclosed.

Approved by SSL\MICHAEL.BAKAS on 7/26/23 5:27PM

ARO FORM 36-E

REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggesstions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA, 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any oenalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
26-07-2023	Final Report	26-Apr-2021 - 25-Apr-2023

4. TITLE AND SUBTITLE
Final Report: Secure Communications and Embedded Systems Laboratory at California State University, Fresno

5a. CONTRACT NUMBER
W911NF-21-1-0210

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
060122

6. AUTHORS

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
California State University - Fresno
4910 North Chestnut Avenue
Fresno, CA 93726 -1852

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
78200-SM-REP.9

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Hovannes Kulhandjian
UU	UU	UU	UU		19b. TELEPHONE NUMBER 559-278-6297

RPPR Final Report
as of 26-Jul-2023

Agency Code: 21XD

Proposal Number: 78200SMREP **Agreement Number: W911NF-21-1-0210**
INVESTIGATOR(S):

Name: Ph.D. Hayssam El-Razouk
Email: helrazouk@csufresno.edu
Phone Number: 5592782094
Principal: N

Name: Ph.D. Hovannes Kulhandjian
Email: hkulhandjian@csufresno.edu
Phone Number: 5592786297
Principal: Y

Organization: **California State University - Fresno**
Address: 4910 North Chestnut Avenue, Fresno, CA 937261852
Country: USA

DUNS Number: 150837003 EIN: 946003272
Report Date: 25-Jul-2023 Date Received: 26-Jul-2023
Final Report for Period Beginning 26-Apr-2021 and Ending 25-Apr-2023
Title: Secure Communications and Embedded Systems Laboratory at California State University, Fresno
Begin Performance Period: 26-Apr-2021 **End Performance Period:** 25-Apr-2023
Report Term: 0-Other

Submitted By: Ph.D. Hovannes Kulhandjian Email: hkulhandjian@csufresno.edu
Phone: (559) 278-6297

Distribution Statement:

STEM Degrees: 26 **STEM Participants:** 26

Major Goals: 1. Acquisition and installation of required equipment (this grant application).

2. Development of the following new lecture-based courses and laboratory offerings:
a. A new hands-on lab/research targeting the topic of advanced signal processing.
b. A new hands-on lab/research targeting the topic of applied machine learning.
c. A new hands-on lab/research targeting the topic of software-defined communications.
d. A new hands-on lab/research targeting the topic of cryptography and network security.
e. A new course targeting the topic of hardware security including hands-on lab components.
3. Developing undergraduate and graduate applied research in the area of communications systems and embedded systems security.
4. Attracting high school students in our region to the field of communications systems and embedded systems security, through outreach summer camp activities as part of faculty services.

Accomplishments: Goal 1. - We have completed the purchase, testing, and installation of all the equipment for the lab.
Goal 2. - The PI and the Co-PI have been working on developing the new labs and courses. Several labs and course materials have been developed and some samples are provided in the Appendix.
Co-PI will be using the "The Hardware Hacking Handbook - Breaking Embedded Security with Hardware Attacks" by Colin O'Flynn and Jasper van, Woudenberg, November 2021, 512 pp., ISBN-13: 9781593278748
Co-PI included a sample "DPA Attack on AES" exercise.
Goal 3. - The Co-PI has been working with two groups of undergraduate students (5 students total) who used the ChipWhisperer toolkits and completed two final projects (one project for each group) in their cryptography course in Spring 2022.
- The PI has worked with one group composed of 3 females and one male who were using the SDR modules to implement RF-Fingerprinting.
- The PI also worked with another group of senior design students composed of three students, who used SDR to create a multi-users optical communication framework.
- One paper in early access IEEE TCAS-I journal co-authored with his previous grad student.
- One published paper in IEEE CCWC 2023 co-authored with four undergrad students.

- Additionally, four undergrad students have been working on a final project using the ChipWhisperer and ChipShouter tools as part of their cryptography course in spring 2023.
- Co-PI published: - One IEEE TVLSI journal paper as the sole author.
- One paper published in IEEE CCWC 2021 was co-authored with four undergraduate students.
- The PI has two papers accepted at the GNU Radio Conference that was held in Washington DC 2022.
- The PI has submitted with his graduate student a paper to MILCOM 2022 conference, which was accepted and presented, and published.
- The PI has been working with 5 groups of students composed of three students in each group on projects related to Software Defined Radios and Machine Learning. Some of their reports and project Posters are attached.
- The PI has been training and working with a Graduate Student on his MS Project using Software Defined Radios. Goal 4. - The PI has given summer training to Middle School (26 participants) and High School (24 participants) students in which he introduced and motivated students to pursue their studies in wireless communications as well as embedded systems. - The PI also gave a Python Workshop with hands-on activities to Lyles College of Engineering mainly freshmen students he had 39 students participate in that workshop during the 2022 summer.
- In addition to that the PI gave four more workshops on Python coding and Robotics to Middle School, two times to High School, and to Lyles College of Engineering students in the summer of 2023.

RPPR Final Report
as of 26-Jul-2023

Training Opportunities: - The PI spent one year over the course of two semesters and worked with 5 undergraduate students and one graduate student. He has trained them on using the Software-Defined Radio the undergraduate student used it during their Senior Design projects entitled "RF Fingerprinting Combined with Deep Learning" and the other group is working on "Observing Optical Wireless Resource Allocation Via SDR" as well as "Software Defined Visible Light Communications in Underwater", this project was an additional project sponsored by college grants for undergraduate research.
The two groups of senior design students participated in the Beyond 5G Software Defined Radio University Challenge and they were selected to be the finalists and were invited to participate in the final round in person at Wright Brothers Institute (WBI) in Dayton, OH in April 2022. The PI's graduate student, Gregory Dzhezyan, was and is still continuing to work on his Master's Project entitled, "Implementation of Faster-than-Nyquist Signaling using Software Defined Radios".
Based on his work we submitted one research paper entitled, "Implementation of Faster-Than-Nyquist Signaling Using Software Defined Radios" to the GNURadio Conference which was accepted and PI's graduate student Gregory Dzhezyan traveled to Washington, DC to present it in 2022. In addition to that another paper, "Pulse Shaping for Faster-Than-Nyquist to Enable Low-Complexity Detection", was submitted to Military Communications Conference (MILCOM 2022) and it was accepted, we presented the paper successfully.
In addition to that, the PI has one additional excepted paper that was submitted GNU Radio Conference entitled, "SDR Beyond Radio: An OOT GNURadio Library for Simulation and Deployment of Multi-Cell / Multi-User Optical Wireless Communication Systems".
- The PI has also been training another undergraduate student, Nicholas Amely, on Drones and Robotics as well as Machine Learning. After squaring sufficient training, Nicholas Amely has been helping the PI by hosting summer Workshops for Middle and High school students as well as University Freshmen students on Robotics and Coding in 2022.
- In addition to that, the PI is also working with several high school students training them on how to code in Python as well as C++ to program and develop projects on Rasberry PI as well as MicroBit also during the summer of 2022.
- The PI has mentored his graduate student, Gregory Dzhezyan, for over the course of one year and he successfully defended his master's project and graduated with honors, he was also nominated and later awarded as the winner of the Dean's Medalist across the whole Lyles College of Engineering. Currently, he works for Northrop Grumman as a digital signal processing engineer. Gregory mentioned to the PI that the Software Defined Radio skills that he gained during his training thanks to this DOD grant instrumentation had a major role in him getting the job.
- The PI participated in the Farm Robotics Challenge in which 12 universities including our university, Fresno State, UC Davis, UC Merced, UC Sana Cruz, UC Santa Barbra, Perdu University, Carnegie Mellon University, and several other universities. There were 19 teams total, the PI participated in the challenge with 2 graduate students, 6 undergraduate students, and two high school students. The team successfully completed eight challenging tasks that include GPS'less navigation, AI-based pesticides detection, and spraying, extracting soil measurements autonomously, energy harvesting, fruit picking using AI and a robotic arm, people following, collision avoidance, and many other smart farming applications. The research work was accepted to be presented at the latest 2023 FIRA, the Ag Robotics Forum that will take place in Salinas, CA in September.
-The PI is also invited to give an Invited Talk on "Design and Implementation of a Smart Agricultural Robot bullDOG (SARDOG)" at the FIRA conference in Sept. 2023.
-The PI also was invited to give an Invited talk on SARDOG (Smart Agricultural Robot bullDOG) at the Fresno State Innovation & Entrepreneurship Summit. The PI and his student research group also presented a poster and won 2nd place in the student competition. The team also presented a live demo of the robot to the audience including the Fresno State President.
-During spring 2022, the Co-PI mentored five undergraduate senior design projects. Three of these projects are related to cryptography and hardware security. All five groups have successfully completed and presented their projects.
-In addition, the Co-PI trained 8 undergraduate and 1 graduate student on conducting cryptography and hardware security-based projects as part of the ECE 156 class in spring 2022.
-The Co-PI has been training his graduate student on creating Verilog prototypes of hardware-oriented data compressors. The graduate student graduated in Spring 2022.
-The Co-PI mentored 1 graduate project during 2021 - 2022 (graduated in spring 2022). The project is about the hardware design of a new data compressor. Co-PI has been advising one graduate student who is working on side-channel analysis on AES and will be utilizing ChipWhisperer toolkits.

RPPR Final Report
as of 26-Jul-2023

Results Dissemination: Several Papers Accepted for publication:

1. H. Kulhandjian, E. Batz, E. Garcia, S. Vega, S. Velma, M. Kulhandjian, C. D'Amours, B. Kantarci, T. Mukherjee, "AI-based RF-Fingerprinting Framework and Implementation using Software-Defined Radios," in Proc. of IEEE International Conference on Computing, Networking and Communications (ICNC), Honolulu, Hawaii, Feb. 2023.
2. M. Kulhandjian, G. Dzhezyan, H. Kulhandjian, C. D'Amours, "Pulse Shaping for Faster-Than-Nyquist to Enable Low-Complexity Detection," in Proc. of IEEE Military Communication Conference (MILCOM), Rockville, MD, 28 Nov. - 2 Dec., 2022.
3. Marshall, Kyle, Thomas Cantido, Jonathan Case, Long Nguyen, and Hayssam El-Razouk. "An Event-Driven Authentication Approach for Mediation of User Actions." In 2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 0534-0540. IEEE, 2021.
4. El-Razouk, Hayssam. "Input-Latency Free Versatile Bit-Serial GF (2 m) Polynomial Basis Multiplication." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 30, no. 5 (2022): 589-602.
5. G. Dzhezyan, M. Kulhandjian, H. Kulhandjian, C. D'Amours, "Implementation of Faster-than-Nyquist Signaling using Software Defined Radios," in Proc. of the GNU Radio Conference (GRCon), Washington, DC, Sept. 2022.
6. A. Ahmed, G. Dzhezyan, H. Aboutahoun, V. Chu, J. DiViccaro, V. Ohanian, S. Rezaeiboroujerdi, I. Saheb, E. Urban, J. Wu, M. Kulhandjian, H. Kulhandjian, M. Rahaim, "SDR Beyond Radio: An OOT GNU Radio Library for Simulation and Deployment of Multi-Cell/Multi-User Optical Wireless Communication Systems," in Proc. of the GNU Radio Conference (GRCon), Washington, DC, Sept. 2022.
7. Puligunta, Mahidhar, and Hayssam El-Razouk. "Squeezing Area of the Versatile GF (2m) GNB Arithmetic Operators", IEEE Transactions on Circuits and Systems I: Regular Papers (2023).
8. Cabrera, Alejandro, Austin Rider, Jennifer Xiong, and Hayssam El-Razouk. "Implementing Privacy on Public Digital Displays Using Smart Glasses." In 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), pp. 274-279. IEEE, 2023.

Honors and Awards: - The PI received Fresno State Transportation Institute (FSTI) SB1 Research Grant, Project "Smart Robot Design and Implementation to Assist Pedestrian Road Crossing", awarded $14,000, Feb. 2023.
- The PI's research team students received a second-place poster competition for their research work on SARDOG (Smart Agricultural Robot bullDOG) at the Fresno State Innovation & Entrepreneurship Summit Spring 2023.
- The Co-PI was Tenured and Promoted to the Associate Professor Rank in Fall 2022.
- The PI received Fresno State Transportation Institute (FSTI) SB1 Research Grant, Project "AI-based Bridge and Road Inspection Framework using Drones", awarded $12,000, Feb. 2022.
- The PI received USA Army xTechHBCU Competition Award, Project "AI-based Human Detection, and Classi fiction in Zero-Visibility Combat Operations", PI, awarded $5,000, Jan. 2022.
- The PI's two senior design student groups were selected finalists for the Beyond 5G Software Defined Radio University Challenge. The PI along with his two students, one student from each group was invited to participate in-person challenge presentation at Dayton, OH in April 2022. One of the projects was in collaboration with a Professor from UMass Boston.
- The PI's graduate student, Gregory Dzhezyan, won a Travel Award from Air Force Research Laboratory (AFRL), Wright Brothers Institute (WBI) during participation in the Beyond 5G Software Defined Radio University Challenge 2022.
- The Co-PI was awarded the "Best Presenter" distinction for the presented IEEE CCWC 2023 paper "Implementing Privacy on Public Digital Displays Using Smart Glasses".
- The Co-PI was elevated to a Senior IEEE Member in Nov. 2022.
- The PI's graduate student, Gregory Dzhezyan, won the Dean's Medalist Award across the whole college.
- The PI's three groups of senior design students were awarded a DPS Telecom Inc. grant to pursue their research in AI-applied research.
- The PI's undergraduate student, Nicholas Amely, received a LCOE research award to pursue research on AI-based drowning person detection in the channels using Machine Learning and IoT devices. The research was presented as a poster at the annual Projects Day in our school in May 2023
- The PI's graduate student, Gregory Dzhezyan, was awarded a LCOE research award to peruse research on developing "Low-cost Optical Wireless CDMA Transceiver Design for IoT Applications", this research work currently has been submitted to the IEEE Latin-American Conference on Communications 2023.

Protocol Activity Status:

RPPR Final Report
as of 26-Jul-2023

Technology Transfer: Several Papers Accepted for publication:

1. H. Kulhandjian, E. Batz, E. Garcia, S. Vega, S. Velma, M. Kulhandjian, C. D'Amours, B. Kantarci, T. Mukherjee, "AI-based RF-Fingerprinting Framework and Implementation using Software-Defined Radios," in Proc. of IEEE International Conference on Computing, Networking and Communications (ICNC), Honolulu, Hawaii, Feb. 2023.
2. M. Kulhandjian, G. Dzhezyan, H. Kulhandjian, C. D'Amours, "Pulse Shaping for Faster-Than-Nyquist to Enable Low-Complexity Detection," in Proc. of IEEE Military Communication Conference (MILCOM), Rockville, MD, 28 Nov. - 2 Dec., 2022.
3. Marshall, Kyle, Thomas Cantido, Jonathan Case, Long Nguyen, and Hayssam El-Razouk. "An Event-Driven Authentication Approach for Mediation of User Actions." In 2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 0534-0540. IEEE, 2021.
4. El-Razouk, Hayssam. "Input-Latency Free Versatile Bit-Serial GF (2 m) Polynomial Basis Multiplication." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 30, no. 5 (2022): 589-602.
5. G. Dzhezyan, M. Kulhandjian, H. Kulhandjian, C. D'Amours, "Implementation of Faster-than-Nyquist Signaling using Software Defined Radios," in Proc. of the GNU Radio Conference (GRCon), Washington, DC, Sept. 2022.
6. A. Ahmed, G. Dzhezyan, H. Aboutahoun, V. Chu, J. DiViccaro, V. Ohanian, S. Rezaeiboroujerdi, I. Saheb, E. Urban, J. Wu, M. Kulhandjian, H. Kulhandjian, M. Rahaim, "SDR Beyond Radio: An OOT GNU Radio Library for Simulation and Deployment of Multi-Cell/Multi-User Optical Wireless Communication Systems," in Proc. of the GNU Radio Conference (GRCon), Washington, DC, Sept. 2022.
7. Puligunta, Mahidhar, and Hayssam El-Razouk. "Squeezing Area of the Versatile GF (2m) GNB Arithmetic Operators", IEEE Transactions on Circuits and Systems I: Regular Papers (2023).
8. Cabrera, Alejandro, Austin Rider, Jennifer Xiong, and Hayssam El-Razouk. "Implementing Privacy on Public Digital Displays Using Smart Glasses." In 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), pp. 274-279. IEEE, 2023.

Several other papers are under preparation and submission.

PARTICIPANTS:

Participant Type: PD/PI
Participant: Hovannes Kulhandjian
Person Months Worked: 12.00 **Funding Support:**
Project Contribution:
National Academy Member: N

Participant Type: Co PD/PI
Participant: Hayssam El-Razouk
Person Months Worked: 12.00 **Funding Support:**
Project Contribution:
National Academy Member: N

ARTICLES:

Publication Type: Journal Article Peer Reviewed: Y **Publication Status:** 1-Published
Journal: IEEE Transactions on Circuits and Systems I: Regular Papers
Publication Identifier Type: DOI Publication Identifier: 10.1109/TCSI.2023.3253797
Volume: 70 Issue: 6 First Page #: 2425
Date Submitted: 7/21/23 12:00AM Date Published: 6/1/23 7:00AM
Publication Location: IEEE, USA
Article Title: Squeezing Area of the Versatile GF (2m) GNB Arithmetic Operators
Authors: Mahidhar Puligunta, Hayssam El-Razouk
Keywords: Gaussian processes , Inverters , Computer architecture , Finite element analysis , Arithmetic , Galois fields , Field programmable gate arrays
Abstract: Cryptography primitives have a prominent role in securing applications that may require low-area realizations, for example portable devices and other resource constrained devices. A given system may require support for different cryptography based protocols/ primitives. Many standardized and/or published primitives rely on arithmetic operations over GF(2m) that occupy major area footprint. Therefore, versatile operators have been of interest to reduce the area penalty, in particular bit-serial multipliers. This paper introduces a novel scheme for versatile multiplication by the normal element in the Gaussian Normal Basis (GNB) leading to new low-area versatile GNB multiplier and inverter architectures that are presented for the first time, as far as we know. Specifically, the proposed inverters are the first versatile GNB inversion in open literature, to the best of our knowledge. Field Programmable Gate Arrays (FPGA) implementation results demonstrate that the proposed versatile multi
Distribution Statement: 1-Approved for public release; distribution is unlimited.
Acknowledged Federal Support: **Y**

Publication Type: Journal Article Peer Reviewed: Y **Publication Status:** 1-Published
Journal: IEEE Transactions on Very Large Scale Integration (VLSI) Systems
Publication Identifier Type: DOI Publication Identifier: 10.1109/TVLSI.2022.3155611
Volume: 30 Issue: 5 First Page #: 589
Date Submitted: 7/22/23 12:00AM Date Published: 5/1/22 7:00AM
Publication Location:
Article Title: Input-Latency Free Versatile Bit-Serial GF(2m) Polynomial Basis Multiplication
Authors: Hayssam El-Razouk
Keywords: Hardware , Gaussian processes , Clocks , Loading , Cryptography , Throughput , Ciphers
Abstract: Cryptography and error correction codes are widely used for information security and data integrity services in modern digital computing and communications systems. A number of standardized and published cryptography, and error correction algorithms utilize arithmetic operations over GF(2m). The polynomial basis (PB) representation of GF(2m) elements is suitable for the support of multiple GF(2m) fields (that is, versatility). This article considers the problem of reduced hardware efficiency of versatile GF(2m) PB multipliers as a result of increased loading latency of the inputs and irreducible polynomials. In this context, to the best of our knowledge, this article proposes the first input-latency free versatile bit-serial GF(2m) multiplier using PB. The superiority of the proposed versatile PB multiplier is demonstrated for the case where the multiplication’s inputs arrive serially in terms of higher output throughput and hardware efficiency compared to existing versatile PB and Gau
Distribution Statement: 1-Approved for public release; distribution is unlimited.
Acknowledged Federal Support: **Y**

CONFERENCE PAPERS:

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: 2023 International Conference on Computing, Networking and Communications (ICNC)
Date Received: 20-Jul-2023 Conference Date: 20-Feb-2023 Date Published:
Conference Location: Honolulu, HI, USA
Paper Title: AI-based RF-Fingerprinting Framework and Implementation using Software-Defined Radios
Authors: Hovannes Kulhandjian, Elizabeth Batz, Eduardo Garcia, Selena Vega, Sanjana Velma, Michel Kulhandji
Acknowledged Federal Support: **Y**

RPPR Final Report
as of 26-Jul-2023

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)
Date Received: 20-Jul-2023 Conference Date: 28-Nov-2022 Date Published:
Conference Location: Rockville, MD, USA
Paper Title: Pulse Shaping for Faster-than-Nyquist to Enable Low-Complexity Detection
Authors: Michel Kulhandjian, Gregory Dzhezyan, Hovannes Kulhandjian, Claude D'Amours
Acknowledged Federal Support: **Y**

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: GNU Radio Conference
Date Received: 21-Jul-2023 Conference Date: 27-Sep-2022 Date Published: 29-Sep-2022
Conference Location: Washington, D.C.
Paper Title: Implementation of Faster-than-Nyquist Signaling using Software Defined Radios
Authors: Gregory Dzhezyan, Michel Kulhandjian, Hovannes Kulhandjian
Acknowledged Federal Support: **Y**

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: GNU Radio Conference
Date Received: 21-Jul-2023 Conference Date: 27-Sep-2022 Date Published: 26-Sep-2022
Conference Location: Washington, D.C.
Paper Title: SDR Beyond Radio: An OOT GNU Radio Library for Simulation and Deployment of Multi-Cell/Multi-User Optical Wireless Communication Systems
Authors: Arsalan Ahmed, Gregory Dzhezyan, Hussam Aboutahoun, Victor Chu, Jordan DiViccaro, Vahae Ohania
Acknowledged Federal Support: **Y**

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC)
Date Received: 21-Jul-2023 Conference Date: 08-Mar-2023 Date Published:
Conference Location: Las Vegas, NV, USA
Paper Title: Implementing Privacy on Public Digital Displays Using Smart Glasses
Authors: Alejandro Cabrera, Austin Rider, Jennifer Xiong, Hayssam El-Razouk
Acknowledged Federal Support: **Y**

Publication Type: Conference Paper or Presentation **Publication Status:** 1-Published
Conference Name: 2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)
Date Received: 22-Jul-2023 Conference Date: 27-Oct-2021 Date Published:
Conference Location: Vancouver, BC, Canada
Paper Title: An Event-Driven Authentication Approach for Mediation of User Actions
Authors: Kyle Marshall, Thomas Cantido, Jonathan Case, Long Nguyen, Hayssam El-Razouk
Acknowledged Federal Support: **Y**

RPPR Final Report
as of 26-Jul-2023

Partners

,

I certify that the information in the report is complete and accurate:
Signature: Hovannes Kulhandjian
Signature Date: 7/26/23 2:13AM

Secure Communications and Embedded Systems Laboratory
at California State University, Fresno.

PI: Hovannes Kulhandjian, Co-PI: Hayssam El-Razouk
California State University, Fresno

DoD Grant W911NF2110210

Final Report

7/25/2023

In an effort to improve science, technology, engineering, and mathematics (STEM) education at California State University, Fresno (Fresno State), by incorporating the missing field of secure communications, software defined radios, cryptography and research, we requested funding that was provided by the department of defense (DOD) to purchase vital equipment that will help to advance STEM education at Fresno State and give our students the hands-on skills and research experience to be successful in competitive careers. The developed new laboratory will be used for conducting undergraduate and graduate research, in addition to complementing existing/ new courses through providing state-of-art hands-on training facilities for our students.

Our plan meant to address four items:

1. Acquisition and installation of required equipment (this grant application).
2. Development of the following new lecture-based courses and laboratory offerings:
    a. A new hands-on lab/research targeting the topic of advanced signal processing.
    b. A new hands-on lab/research targeting the topic of applied machine learning.
    c. A new hands-on lab/research targeting the topic of software-defined communications.
    d. A new hands-on lab/research targeting the topic of cryptography and network security.
    e. A new course targeting the topic of hardware security including hands-on lab components.
3. Developing undergraduate and graduate applied research in communications systems and embedded systems security.
4. Attracting high school students in our region to the field of communications systems and embedded systems security, through outreach summer camp activities as part of faculty services and encourage them to join the STEM fields.

Accomplishments Under Goals

Goal 1. - We have completed the purchase, testing, and installation of all the equipment for the lab.
Goal 2. - The PI and the Co-PI have been working on developing the new labs and courses. Several labs and course materials have been developed and some samples are provided in the Appendix.
Co-PI will be using the "The Hardware Hacking Handbook - Breaking Embedded Security with Hardware Attacks" by Colin O'Flynn and Jasper van, Woudenberg, November 2021, 512 pp., ISBN-13: 9781593278748

Co-PI included a sample "DPA Attack on AES" exercise.
Goal 3. - The Co-PI has been working with two groups of undergraduate students (5 students total) who used the ChipWhisperer toolkits and completed two final projects (one project for each group) in their cryptography course in Spring 2022.
- The PI has worked with one group composed of 3 females and one male who were using the SDR modules to implement RF-Fingerprinting.
- The PI also worked with another group of senior design students composed of three students, who used SDR to create a multi-users optical communication framework.
- One paper in early access IEEE TCAS-I journal co-authored with his previous grad student.
- One published paper in IEEE CCWC 2023 co-authored with four undergrad students.
- Additionally, four undergrad students have been working on a final project using the ChipWhisperer and ChipShouter tools as part of their cryptography course in spring 2023.
- Co-PI published: - One IEEE TVLSI journal paper as the sole author.
- One paper published in IEEE CCWC 2021 was co-authored with four undergraduate students.
- The PI has two papers accepted at the GNU Radio Conference that was held in Washington DC 2022.
- The PI has submitted with his graduate student a paper to MILCOM 2022 conference, which was accepted and presented, and published.
- The PI has been working with 5 groups of students composed of three students in each group on projects related to Software Defined Radios and Machine Learning. Some of their reports and project Posters are attached.
- The PI has been training and working with a Graduate Student on his MS Project using Software Defined Radios.
Goal 4. - The PI has given summer training to Middle School (26 participants) and High School (24 participants) students in which he introduced and motivated students to pursue their studies in wireless communications as well as embedded systems. - The PI also gave a Python Workshop with hands-on activities to Lyles College of Engineering mainly freshmen students he had 39 students participate in that workshop during the 2022 summer.
- In addition to that the PI gave four more workshops on Python coding and Robotics to Middle School, two times to High School, and to Lyles College of Engineering students in the summer of 2023.

Equipment Purchased

Some of the main equipment that we acquired through the DOD grant includes software defined radios from National Instruments (NI), Embedded Security FPGA/Side Channel Advanced Kit from NewAE Technology Inc., Dell laptops/computers, instrumentation measuring tools including educational hardware kits for student hands-on training on embedded systems. Some of the major equipment that we were planning to purchase, software defined radios USRP N320 had gone up in prices considerably therefore we had to make some changes to the SDR equipment. Also, after consulting with the National Instrument engineers we were advised on what equipment would best suit our educational and research purposes given the budget that we had. We were also advised on acquiring some equipment that was essential for the software defined radios. The major equipment that we have purchased and installed in the lab are listed below. There are many miscellaneous equipment, e.g., cables, power supplies, connectors, tools, wires, educational kits, etc., that were not listed in the table.

Vendor	Tool Function	Equipment Brand/ Reference	Quantity	Price	Total Cost
National Instruments	Software defined radios	USRP X310	10	$5,904.00	$59,040.00
	Software defined radios	USRP N210	4	$2,800.00	$12,000.00
	Software defined radios	USRP B210	4	$1,800.00	$7,200.00
	Software defined radios	USRP B205min	4	$1,500.00	$6,000.00
	Software defined radios	USRP Air-T by DeepWave Digital	2	$5,892.25	$11,784.50
	Software defined radios	ADALM-PLUTO	18	$150.00	$2,700.00
	USRP Daughterboards, antennas, and GPS, etc.	DBSRX2	16	$182.00	$2,912.00
		UBX-40	8	$1,091.70	$8,733.60
		CBX-40	16	$573.00	$9,168.00
		UBX-40	16	$1,056.00	$16,896.00
		Vert2450	16	$43.00	$688.00
		3V GPS	16	$62.00	$992.00
		OctoClockCDA-2990	2	$2,097.90	$4,195.80
		GPSDO Kit	4	$772.20	$3,088.80
NewAE Technology Inc.	Embedded Security FPGA/Side Channel Advanced Kits	NAE-CW1200 ChipWhisperer-Pro	16	$5,007.00	$80,112.00

RIGOL Innovation or nothing	Electric measurement equipment	DS2302A, DG1022Z, DP832, MSO8204, RSA5065-TG	18	$2,222.00	$40,521.57
Dell	Computing	Dell Latitude 7400 Laptops	10	$1,612.83	$25,805.28
		Dell Precision 5570	2	$3,147.01	$6,294.02
		Dell Precision 7560 Laptops	2	$2,404.20	$9,616.8
		Dell Precision 3650 Tower	2	$1,397.26	$2,794.52
		Dell Optiplex 7090 Desktop	2	$2,314.00	$4,682.00
		CLX - SET Gaming Desktop	4	$4,019.99	$16,079.96
		Samsung 27” monitors	4	$229.99	$919.96
TI	Hardware	Data capture boards	4	$700.00	$2,800.00
		Digilent JTAG-HS3 Programming	12	$59.00	$708

Table 1: The major equipment purchased to develop the instrumentation laboratory.

Results Dissemination

Several Papers Accepted for publication:

1. **H. Kulhandjian**, E. Batz, E. Garcia, S. Vega, S. Velma, M. Kulhandjian, C. D'Amours, B. Kantarci, T. Mukherjee, "AI-based RF-Fingerprinting Framework and Implementation using Software-Defined Radios," in Proc. of IEEE International Conference on Computing, Networking and Communications (ICNC), Honolulu, Hawaii, Feb. 2023.

2. M. Kulhandjian, G. Dzhezyan, **H. Kulhandjian**, C. D'Amours, "Pulse Shaping for Faster-Than-Nyquist to Enable Low-Complexity Detection," in Proc. of IEEE Military Communication Conference (MILCOM), Rockville, MD, 28 Nov. - 2 Dec., 2022.

3. Marshall, Kyle, Thomas Cantido, Jonathan Case, Long Nguyen, and **Hayssam El-Razouk**. "An Event-Driven Authentication Approach for Mediation of User Actions." In 2021 IEEE 12th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), pp. 0534-0540. IEEE, 2021.

4. **El-Razouk, Hayssam**. "Input-Latency Free Versatile Bit-Serial GF (2 m) Polynomial Basis Multiplication." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 30, no. 5 (2022): 589-602.

5. G. Dzhezyan, M. Kulhandjian, **H. Kulhandjian**, C. D'Amours, "Implementation of Faster-than-Nyquist Signaling using Software Defined Radios," in Proc. of the GNU Radio Conference (GRCon), Washington, DC, Sept. 2022.

6. A. Ahmed, G. Dzhezyan, H. Aboutahoun, V. Chu, J. DiViccaro, V. Ohanian, S. Rezaeiboroujerdi, I. Saheb, E. Urban, J. Wu, M. Kulhandjian, **H. Kulhandjian**, M. Rahaim, "SDR Beyond Radio: An OOT GNU Radio Library for Simulation and Deployment of Multi-Cell/Multi-User Optical Wireless Communication Systems," in Proc. of the GNU Radio Conference (GRCon), Washington, DC, Sept. 2022.

7. Puligunta, Mahidhar, and **Hayssam El-Razouk**. "Squeezing Area of the Versatile GF (2m) GNB Arithmetic Operators", IEEE Transactions on Circuits and Systems I: Regular Papers (2023).

8. Cabrera, Alejandro, Austin Rider, Jennifer Xiong, and **Hayssam El-Razouk**. "Implementing Privacy on Public Digital Displays Using Smart Glasses." In 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), pp. 274-279. IEEE, 2023.

Honors and Awards

- The PI received Fresno State Transportation Institute (FSTI) SB1 Research Grant, Project "Smart Robot Design and Implementation to Assist Pedestrian Road Crossing", awarded $14,000, Feb. 2023.

- The PI's research team students received a second-place poster competition for their research work on SARDOG (Smart Agricultural Robot bullDOG) at the Fresno State Innovation & Entrepreneurship Summit Spring 2023.

- The Co-PI was Tenured and Promoted to the Associate Professor Rank in Fall 2022.

- The PI received Fresno State Transportation Institute (FSTI) SB1 Research Grant, Project "AI-based Bridge and Road Inspection Framework using Drones", awarded $12,000, Feb. 2022.

- The PI received USA Army xTechHBCU Competition Award, Project "AI-based Human Detection, and Classi fiction in Zero-Visibility Combat Operations", PI, awarded $5,000, Jan. 2022.

- The PI's two senior design student groups were selected finalists for the Beyond 5G Software Defined Radio University Challenge. The PI along with his two students, one student from each group was invited to participate in-person challenge presentation at Dayton, OH in April 2022. One of the projects was in collaboration with a Professor from UMass Boston.

- The PI's graduate student, Gregory Dzhezyan, won a Travel Award from Air Force Research Laboratory (AFRL), Wright Brothers Institute (WBI) during participation in the Beyond 5G Software Defined Radio University Challenge 2022.

- The Co-PI was awarded the "Best Presenter" distinction for the presented IEEE CCWC 2023 paper "Implementing Privacy on Public Digital Displays Using Smart Glasses".

- The Co-PI was elevated to a Senior IEEE Member in Nov. 2022.

- The PI's graduate student, Gregory Dzhezyan, won the Dean's Medalist Award across the whole college.

- The PI's three groups of senior design students were awarded a DPS Telecom Inc. grant to pursue their research in AI-applied research.

- The PI's undergraduate student, Nicholas Amely, received a LCOE research award to pursue research on AI-based drowning person detection in the channels using Machine Learning and IoT devices. The research was presented as a poster at the annual Projects Day in our school in May 2023.

- The PI's graduate student, Gregory Dzhezyan, was awarded a LCOE research award to peruse research on developing "Low-cost Optical Wireless CDMA Transceiver Design for IoT Applications", this research work currently has been submitted to the IEEE Latin-American Conference on Communications 2023.

Training Opportunities

- The PI spent one year over the course of two semesters and worked with 5 undergraduate students and one graduate student. He has trained them on using the Software-Defined Radio the undergraduate student used it during their Senior Design projects entitled "RF Fingerprinting Combined with Deep Learning" and the other group is working on "Observing Optical Wireless Resource Allocation Via SDR" as well as "Software Defined Visible Light Communications in Underwater", this project was an additional project sponsored by college grants for undergraduate research.

The two groups of senior design students participated in the Beyond 5G Software Defined Radio University Challenge and they were selected to be the finalists and were invited to participate in the final round in person at Wright Brothers Institute (WBI) in Dayton, OH in April 2022. The PI's graduate student, Gregory Dzhezyan, was and is still continuing to work on his Master's Project entitled, "Implementation of Faster-than-Nyquist Signaling using Software Defined Radios".

Based on his work we submitted one research paper entitled, "Implementation of Faster-Than-Nyquist Signaling Using Software Defined Radios" to the GNURadio Conference which was accepted and PI's graduate student Gregory Dzhezyan traveled to Washington, DC to present it in 2022. In addition to that another paper, "Pulse Shaping for Faster-Than-Nyquist to Enable Low-Complexity Detection", was submitted to Military Communications Conference (MILCOM 2022) and it was accepted, we presented the paper successfully.

In addition to that, the PI has one additional excepted paper that was submitted GNU Radio Conference entitled, "SDR Beyond Radio: An OOT GNURadio Library for Simulation and Deployment of Multi-Cell / Multi-User Optical Wireless Communication Systems".

- The PI has also been training another undergraduate student, Nicholas Amely, on Drones and Robotics as well as Machine Learning. After squaring sufficient training, Nicholas Amely has been helping the PI by hosting summer Workshops for Middle and High school students as well as University Freshmen students on Robotics and Coding in 2022.

- In addition to that, the PI is also working with several high school students training them on how to code in Python as well as C++ to program and develop projects on Rasberry PI as well as MicroBit also during the summer of 2022.

- The PI has mentored his graduate student, Gregory Dzhezyan, for over the course of one year and he successfully defended his master's project and graduated with honors, he was also nominated and later awarded as the winner of the Dean's Medalist across the whole Lyles College of Engineering. Currently, he works for Northrop Grumman as a digital signal processing engineer. Gregory mentioned to the PI that the Software Defined Radio skills that he gained during his training thanks to this DOD grant instrumentation had a major role in him getting the job.

- The PI participated in the Farm Robotics Challenge in which 12 universities including our university, Fresno State, UC Davis, UC Merced, UC Sana Cruz, UC Santa Barbra, Perdu University, Carnegie Mellon University, and several other universities. There were 19 teams total, the PI participated in the challenge with 2 graduate students, 6 undergraduate students, and two high school students. The team successfully completed eight challenging tasks that include GPS'less navigation, AI-based pesticides detection, and spraying, extracting soil measurements autonomously, energy harvesting, fruit picking using AI and a robotic arm, people following, collision avoidance, and many other smart farming applications. The research work was accepted to be presented at the latest 2023 FIRA, the Ag Robotics Forum that will take place in Salinas, CA in September.

-The PI is also invited to give an Invited Talk on "Design and Implementation of a Smart Agricultural Robot bullDOG (SARDOG)" at the FIRA conference in Sept. 2023.

-The PI was also invited to give an Invited talk on SARDOG (Smart Agricultural Robot bullDOG) at the Fresno State Innovation & Entrepreneurship Summit. The PI and his student research group also presented a poster and won 2nd place in the student competition. The team also presented a live demo of the robot to the audience including the Fresno State President.

-During spring 2022, the Co-PI mentored five undergraduate senior design projects. Three of these projects are related to cryptography and hardware security. All five groups have successfully completed and presented their projects.

-In addition, the Co-PI trained 8 undergraduate and 1 graduate student on conducting cryptography and hardware security-based projects as part of the ECE 156 class in spring 2022.

-The Co-PI has been training his graduate student on creating Verilog prototypes of hardware-oriented data compressors. The graduate student graduated in Spring 2022.

-The Co-PI mentored 1 graduate project during 2021 - 2022 (graduated in spring 2022). The project is about the hardware design of a new data compressor. Co-PI has been advising one graduate student who is working on side-channel analysis on AES and will be utilizing ChipWhisperer toolkits.

I (PI) would like to share a quote sent to me by one of my Senior Design students, Vahae Ohanian. *Your guidance in our project, titled "5G and Beyond Challenge: Observing Optical Wireless Resources Allocation via Software Defined Radios", has led to an incredibly enriching and enlightening experience for me and my colleagues. The foundations of communication engineering, the principles of Visible Light Communication, and the intricacies of research and industry-grade technology like Software Defined Radios all make up a shortlist of the numerous topics that I have become very familiar with over these past few months through your mentorship. You have always pushed me to strive for the best, and when the best has been achieved, to strive for even better, and only for that reason my group and I are now working with another research group from the University of Massachusetts, Boston, where we are participating in a challenge presented by the Air Force Research Lab. In no other capacity would I have been able to work with a team of peers from across the country, and I now have that experience thanks to you.*

Conclusion:

We received all equipment we purchased, and our lab construction work is almost completed. Many students have already been trained in how to use the state-of-the-art equipment and we have had a dozen projects already completed by our students working on the DOD equipment. We expect an increasing number of both undergraduate and graduate projects to be done in areas related to software defined radios, secure communications, artificial intelligence, machine learning, cryptography, and embedded systems. The PI and Co-PI has offered advanced courses that has heavily utilized the equipment in the student projects. We have offered several workshops for middle school, high school, and our freshmen students, which has provided training for over 300 students over the course of two years. We have developed several labs and

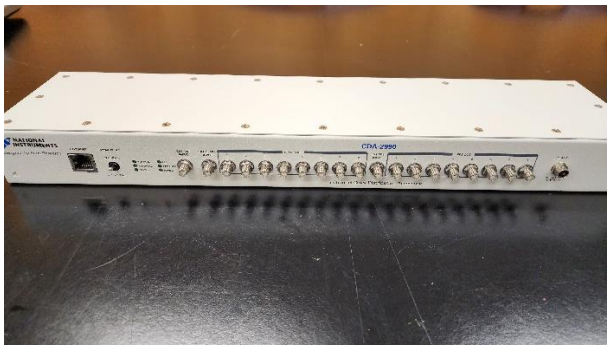
courses that teach the students how to use the equipment and are hands on. The skills gained by the students during the training are valuable to allow them to be competent when they join the workforce. This significant grant the DoD has given us is bringing a significant change to our department and college in advanced teaching and research, and we expect students to continuously benefit from it. We are grateful to DOD for the opportunity to enhance our research and teaching laboratory capabilities at California State University, Fresno.

APPENDIX I: Some of the NI (Ettus Research), Deepwave Digital RF SDR and Analog Devices SDR equipment acquired for the *Secure Communications and Embedded Systems Laboratory* we are establishing at Fresno State are listed below.

(a) USRP X310

(b) USRP X310 back side

(c) USRP N210



(d) OctoClock CDA-2990

(e) USRP B210

(f) USRP Air-T by DeepWave Digital

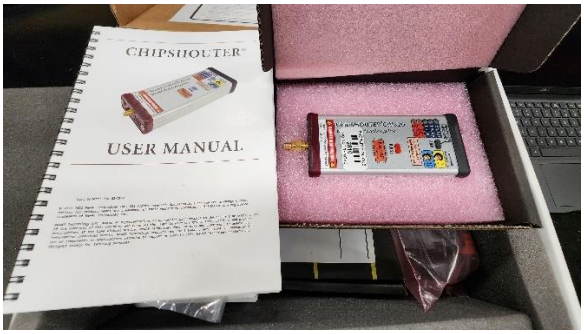
(g) USRP B205mini

(h) ADALM-PLUTO

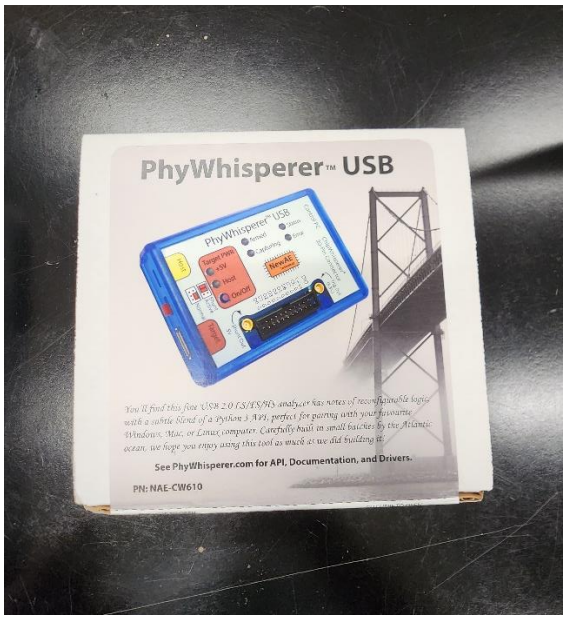
(i) ChipWisperer CW1200 kit

(j) ChipWisperer CW 305

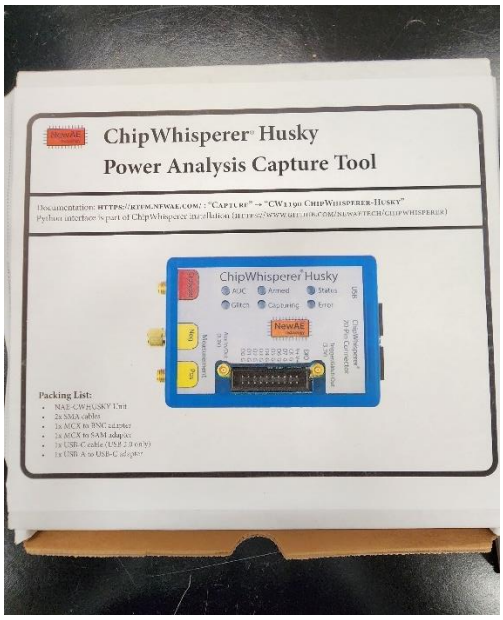
(k) ChipWhisper Bergen Board CW310



(l) ChipSHOUTER NAE-CW520-kit



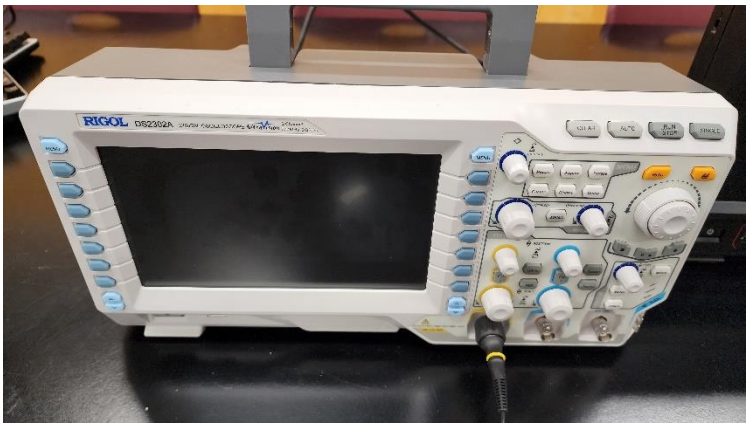
(m) PhyWisper USB



(n) ChipWisper Hksky Power Analysis Capture Tool

(o) RIGOL Spectrum Analyzer RSA5065

(p) RIGOL Digital Osciloscope MSO8204



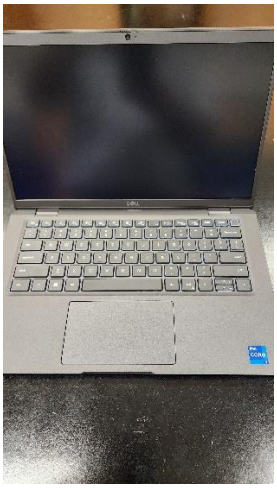
(q) RIGOL Oscilloscope DS2302A

(r) RIGOL Waveform Generator DG 1022

(s) RIGOL Waveform Generator SDG 1032X

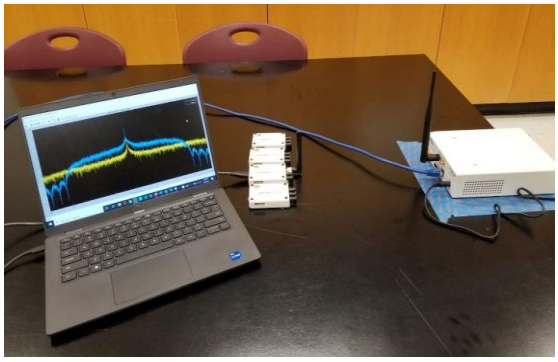
(t) RIGOL Power Supply SPD3303X

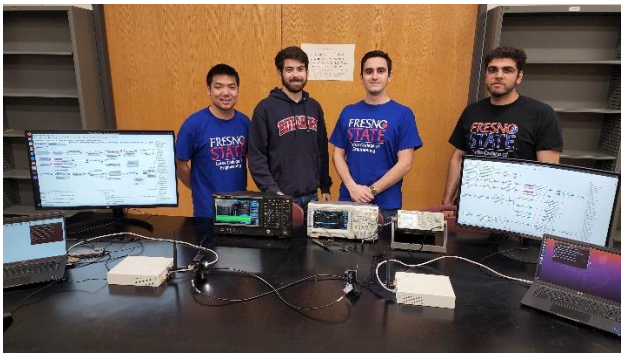
(u) RIGOL DC Power Supply DP832



(v) Dell Laptop

Our students presenting capstone project during Projects Day on Multi-User Visible Light Communications using Software Defined Radios.

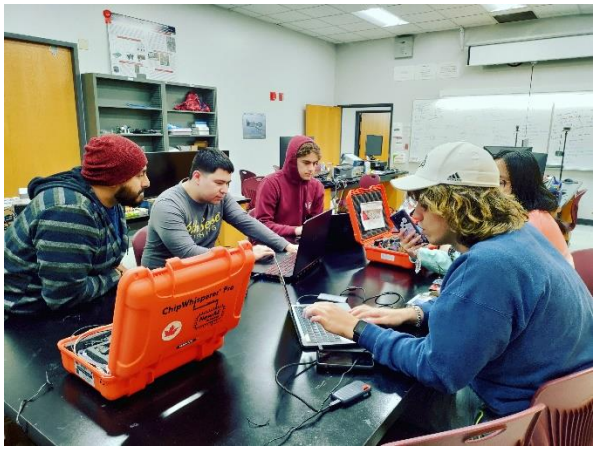


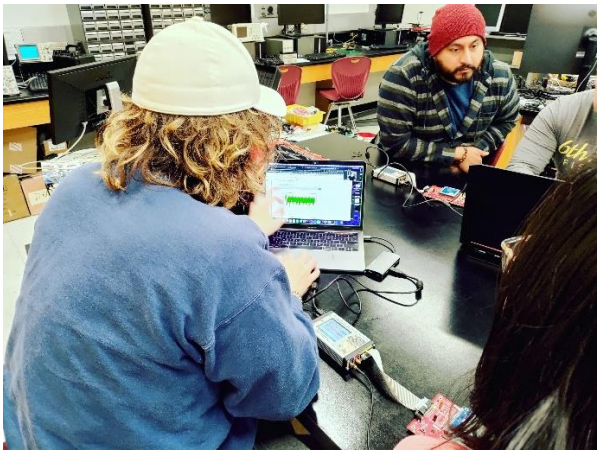


The PI along with his students is testing out the two projects that he is participaitng in the SDR challenge in Dayton Ohaio and presented two projects on on SDR based Visibel light communications the other on RF Fingerpringing using SDR.

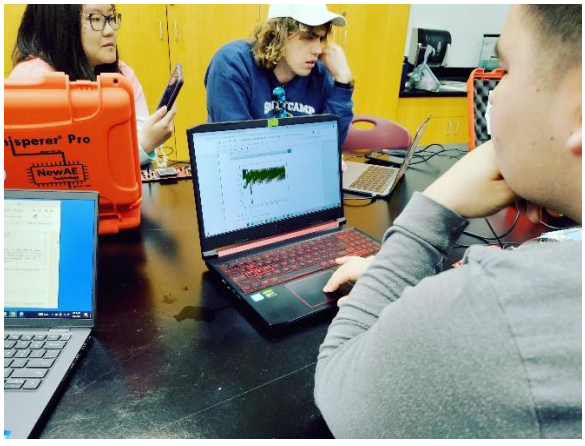
The PI traveled with his students to compete in the SDR challenge in Dayton Ohaio and presented two projects on on SDR based Visibel light communications the other on RF Fingerpringing using SDR.

Our two teams lead by the PI are being judged by AFRL and WBI judges at the SDR challenge in Dayton Ohaio and presented two projects on on SDR based Visibel light communications the other on RF Fingerpringing using SDR.

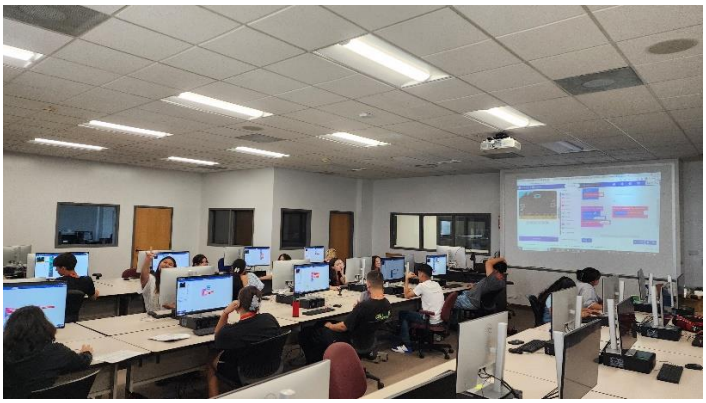




Students working in the lab on Cryptography and Side Channel Analysis projects.



Students working in the lab on Cryptography and Side Channel Analysis projects.



Coding and Robotics Workshop given by PI to Summer Camp High School and Middle School Students.

ECE 156 - Fundamentals of Cryptography and Computer Network Security

DPA Attack on AES Labs

Dr. Hayssam El-Razouk
Electrical and Computer Engineering Department
California State University, Fresno

Acknowledgement: this lab is based on the open-source tutorial material available through the following link:

https://rtfm.newae.com/Capture/ChipWhisperer-Pro/

6. AES DPA Attack

Supported setups:

SCOPES:

- OPENADC
- CWNANO

PLATFORMS:

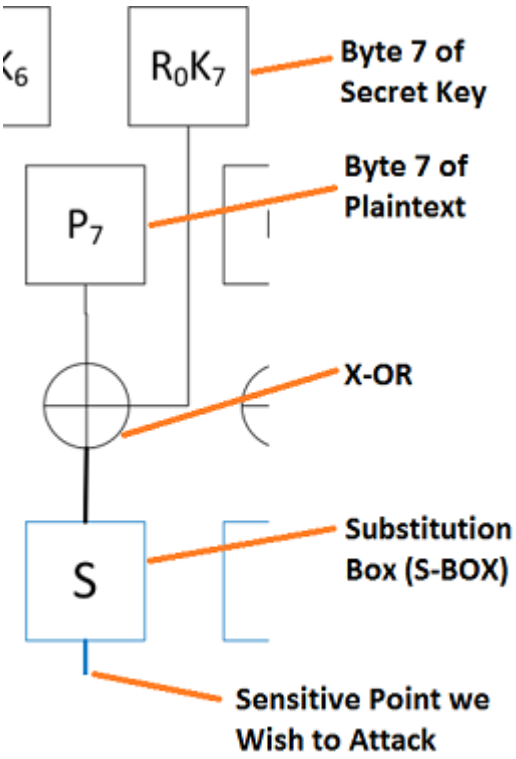
- CWLITEARM
- CWLITEXMEGA
- CWNANO

Before starting this tutorial, it's recommended that you first complete the earlier PA_DPA tutorials since these will familiarize you with the concept of Differental Power Analysis. With that out of the way, let's look at how this attack works.

DPA Attack Theory

As we explored in the earlier DPA tutorials, the Hamming Weight of the result of the SBox operation in AES has a measurable effect on the power consumed by the microcontroller. It turns out that just this effect (and not anything stronger, such as its linearity) is enough information to break an AES key. There's a few different ways we could go about this, but for this tutorial, we'll be looking at difference of means. With this technique, the goal is to separate the traces by a bit in the result of the SBox output (it doesn't matter which one): if that bit is 1, its group of traces should, on average, have higher power consumption during the SBox operation than the other set.

Whether or not we get a large difference in the means between these two groups depends on whether they were properly sorted into these groups. If not, there should be, on average, little difference between the two and therefore a low difference of means. Recall the SBox operation:



The SBox output depends on the subkey, which we don't know (and the plaintext, which we do). However, since there's a large difference of means for the correct key and small ones for the rest of the possible subkeys, we have a method of checking whether a given subkey is correct. If we calculate the difference of means for each subkey, the correct one will have the largest difference of means.

Our plan looks as follows

1. Capture a bunch of power traces with varying plaintext
2. Group each trace by the value of their SBox output's lowest bit for a given key guess
3. Calculate the difference of means
4. Repeat for each possible subkey
5. Select the largest difference of means -> this is the correct subkey
6. Repeat for each subkey in the key

At the end, we should get a correct AES key!

```
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEXMEGA'
CRYPTO_TARGET = 'TINYAES128C'
```

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../../hardware/victims/firmware/simpleserial-aes
make PLATFORM=$1 CRYPTO_TARGET=$2
```

Capturing Power Traces

Capture and setup is similar to earlier tutorials. We'll have to capture a fair number of traces (usually a few thousand here) since Difference of Means isn't a super trace efficient method. As you'll find during the CPA tutorials, CPA is much better in this regard - it can often break AES implementations such as these in under 50 traces.

You may also find that you need to modify gain settings and the number of traces you capture - this attack is much more sensitive to gain settings and noise than a CPA attack would be.

Setup

```
%run "Helper_Scripts/Setup_Generic.ipynb"
```

```
fw_path = "../../hardware/victims/firmware/simpleserial-aes/simpleserial-aes-{}.hex".fo
```

```
cw.program_target(scope, prog, fw_path)
```

Capture

```
%run "Helper_Scripts/plot.ipynb"
plot = real_time_plot(plot_len=3000)
```

```
#Capture Traces
from tqdm import tnrange, trange
import numpy as np
import time

ktp = cw.ktp.Basic()

traces = []
N = 2500  # Number of traces

if PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
    scope.adc.samples = 4000
elif PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
    scope.gain.db = 20
    scope.adc.samples = 1700 - 170
    scope.adc.offset = 500 + 700 + 170
    N = 5000

print(scope)
for i in trange(N, desc='Capturing traces'):
    key, text = ktp.next()  # manual creation of a key, text pair can be substituted he

    trace = cw.capture_trace(scope, target, text, key)
    if trace is None:
        continue
    traces.append(trace)
    plot.send(trace)

#Convert traces to numpy arrays
```

```
trace_array = np.asarray([trace.wave for trace in traces])
textin_array = np.asarray([trace.textin for trace in traces])
known_keys = np.asarray([trace.key for trace in traces])  # for fixed key, these keys a
```

Analysis

As we discussed above, our goal here is to find the biggest difference of means out of the possible subkey values we could have. First, we'll get some values and functions that will be useful for our calculations. We'll be using `intermediate()` later to get the output of the SBox from a plaintext and key input.

```
numtraces = np.shape(trace_array)[0] #total number of traces
numpoints = np.shape(trace_array)[1] #samples per trace

sbox = (
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54,

def intermediate(pt, keyguess):
    return sbox[pt ^ keyguess]
```

Our first step here will be separating our traces into different groups based on the SBox's output. As mentioned earlier, we're separating based on the least significant bit, but really any bit would work (as a test, you can change this and see if the attack still works):

```
one_list = []
zero_list = []
for tnum in range(numtraces):
    if (intermediate(textin_array[tnum][subkey], kguess) & 1):
        one_list.append(trace_array[tnum])
    else:
        zero_list.append(trace_array[tnum])
```

Then calculate the difference of means:

```
one_avg = np.asarray(one_list).mean(axis=0)
zero_avg = np.asarray(zero_list).mean(axis=0)
mean_diffs[kguess] = np.max(abs(one_avg - zero_avg))
```

We'll need to repeat this with each possible key guess and then pick the one with the highest difference of means:

```
guess = np.argsort(mean_diffs)[-1]
key_guess.append(guess)
print(hex(guess))
print(mean_diffs[guess])
```

Finally, altogether and attacking all of the subkeys:

```
from tqdm import tnrange
import numpy as np
mean_diffs = np.zeros(255)
key_guess = []
known_key = known_keys[0]
plots = []
for subkey in tnrange(0, 16, desc="Attacking Subkey"):
    for kguess in tnrange(255, desc="Keyguess", leave=False):
        one_list = []
        zero_list = []

        for tnum in range(numtraces):
            if (intermediate(textin_array[tnum][subkey], kguess) & 1): #LSB is 1
                one_list.append(trace_array[tnum])
            else:
                zero_list.append(trace_array[tnum])
        one_avg = np.asarray(one_list).mean(axis=0)
        zero_avg = np.asarray(zero_list).mean(axis=0)
        mean_diffs[kguess] = np.max(abs(one_avg - zero_avg))
        if kguess == known_key[subkey]:
            plots.append(abs(one_avg - zero_avg))
    guess = np.argsort(mean_diffs)[-1]
    key_guess.append(guess)
    print(hex(guess) + "(real = 0x{:02X})".format(known_key[subkey]))
    #mean_diffs.sort()
    print(mean_diffs[guess])
    print(mean_diffs[known_key[subkey]])
```

With that done, we should now have the correct key:

```
print(key_guess)
print(known_key)
```

We can also plot the difference of means for a few of the correct subkey bytes:

```
%matplotlib notebook
import matplotlib.pylab as plt

plt.plot(plots[0], 'g')
plt.plot(plots[1], 'r')
plt.plot(plots[15], 'b')
```

The Ghost Peak Problem

Depending on your hardware setup (typically the XMEGA), the previous analysis may have failed to break the full key. This is actually due to a well known problem with DPA attacks: "ghost peaks". We won't worry too much about why this happens (if you're interested, you can find a paper about the

issue here), but luckily this issue isn't too difficult to solve. First, let's collect difference data for both the best key guess, as well as the correct key:

```
from tqdm import tnrange
import numpy as np
mean_diffs = np.zeros(255)
known_key = known_keys[0]
best_guess_plots = []
correct_guess_plots = []
for subkey in tnrange(0, 16, desc="Attacking Subkey"):
    for kguess in tnrange(255, desc="Keyguess", leave=False):
        one_list = []
        zero_list = []

        for tnum in range(numtraces):
            if (intermediate(textin_array[tnum][subkey], kguess) & 1): #LSB is 1
                one_list.append(trace_array[tnum])
            else:
                zero_list.append(trace_array[tnum])
        one_avg = np.asarray(one_list).mean(axis=0)
        zero_avg = np.asarray(zero_list).mean(axis=0)
        mean_diffs[kguess] = np.max(abs(one_avg - zero_avg))
        if kguess == known_key[subkey]:
            correct_guess_plots.append(abs(one_avg - zero_avg))
        if kguess == key_guess[subkey]:
            best_guess_plots.append(abs(one_avg - zero_avg))
```

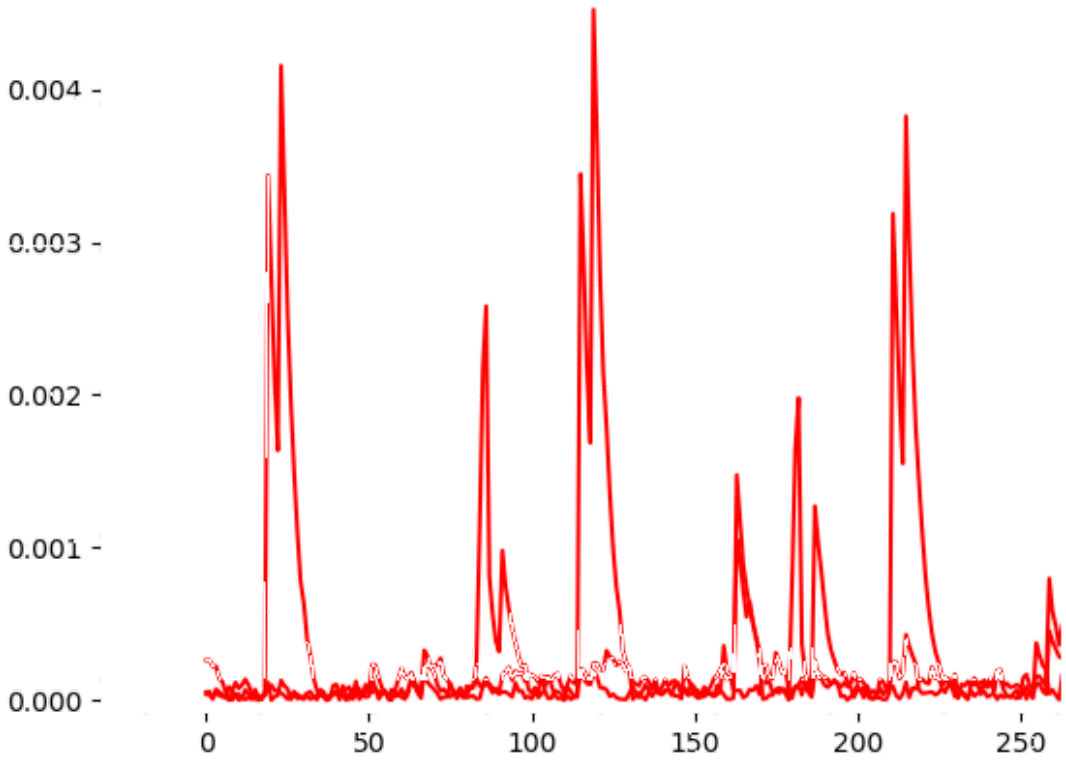
Next, let's take a look at both plots for one of the subkeys that didn't break. If subkey 4 broke for you here, replace it with one that didn't.

```
%matplotlib notebook
import matplotlib.pylab as plt

subkey = 8

plt.plot(correct_guess_plots[subkey], 'r')
plt.plot(best_guess_plots[subkey], 'g')
```

You'll probably find that the "ghost peak" trails slightly behind where the actual operation is taking place (for the XMEGA, this is roughly a difference of 52). Therefore, by moving the starting point of the trace data with the peak, we can eliminate the ghost peak! To get the offset we should use, let's plot the correct plots for the first three subkeys:

```
%matplotlib notebook

for i in range(3):
    plt.plot(correct_guess_plots[i], 'r')
```

From these plots, we can see that the peak moves some constant amount. For example, in the following plot, these three peaks corresponded to x=23, x=119, and x=215, an offset of 96.



One thing to notice here is that the difference between the ghost peak and the correct peak is less than the difference between the first correct peak and the starting point of x=0. This means we can simply move the starting point by 96 to eliminate the ghost peak. Let's retry the analysis, moving the starting point of the trace data up by 96 each time (or whatever you found to be best):

```
from tqdm import tnrange
import numpy as np
mean_diffs = np.zeros(255)
key_guess = []
known_key = known_keys[0]
plots = []
offset = 0

if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
    #offset = 96
    offset = 263

for subkey in range(0, 1):#, desc="Attacking Subkey"):
    for offset in range(0,265):#,desc="offsets", leave=False):
        for kguess in range(255):#, desc="Keyguess", leave=False):
            one_list = []
            zero_list = []

            for tnum in range(numtraces):
                if (intermediate(textin_array[tnum][subkey], kguess) & 1): #LSB is 1
                    one_list.append(trace_array[tnum][subkey+offset:])
                else:
                    zero_list.append(trace_array[tnum][subkey+offset:])
```

```
            one_avg = np.asarray(one_list).mean(axis=0)
            zero_avg = np.asarray(zero_list).mean(axis=0)
            mean_diffs[kguess] = np.max(abs(one_avg - zero_avg))
            if kguess == known_key[subkey]:
                plots.append(abs(one_avg - zero_avg))
        guess = np.argsort(mean_diffs)[-1]
        key_guess.append(guess)
        #print("offset = " + str(offset) + ":\n")
        print(hex(guess) + "(real = 0x{:02X})".format(known_key[subkey]) + ", offset =
        if hex(guess) == known_key[subkey]:
            #print(hex(guess) + "(real = 0x{:02X})".format(known_key[subkey]) + "offset
            #mean_diffs.sort()
            print(mean_diffs[guess])
            print(mean_diffs[known_key[subkey]])
```

```
from tqdm import tnrange
import numpy as np
mean_diffs = np.zeros(255)
key_guess = []
known_key = known_keys[0]
plots = []
offset = 0

if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
    #offset = 96
    offset = 263

for subkey in tnrange(0, 16, desc="Attacking Subkey"):
    for kguess in tnrange(255, desc="Keyguess", leave=False):
        one_list = []
        zero_list = []

        for tnum in range(numtraces):
            if (intermediate(textin_array[tnum][subkey], kguess) & 1): #LSB is 1
                one_list.append(trace_array[tnum][subkey*offset:])
            else:
                zero_list.append(trace_array[tnum][subkey*offset:])
        one_avg = np.asarray(one_list).mean(axis=0)
        zero_avg = np.asarray(zero_list).mean(axis=0)
        mean_diffs[kguess] = np.max(abs(one_avg - zero_avg))
        if kguess == known_key[subkey]:
            plots.append(abs(one_avg - zero_avg))
    guess = np.argsort(mean_diffs)[-1]
    key_guess.append(guess)
    print(hex(guess) + "(real = 0x{:02X})".format(known_key[subkey]))
    #mean_diffs.sort()
    print(mean_diffs[guess])
    print(mean_diffs[known_key[subkey]])
```

As you can see, the DPA attack now has no problem breaking the key!

Conclusion

Congratulations, you have broken AES using a DPA attack! As you might have discovered during this tutorial, there can be quite a few issues with the difference of means method for breaking AES keys:

- It's quite susceptible to noise

- The attack can easily pick up other parts of the AES operation
- The ghost peak problem
- The attack typically requires a lot of traces. These software AES implementations are pretty weak against power analysis, but they still required thousands of traces to break

Nevertheless, using a difference of means attack can still be very useful. For example, a later tutorial, PA_Multi_1, uses a difference of means attack similar in concept to this one to break the signature of an AES256 bootloader.

Tests

```
assert (known_key == key_guess).all(), "Failed to break key.\nGot: {}\nExp: {}".format(
```

FRESNO STATE
Electrical and Computer Engineering

SOFTWARE DEFINED RADIO

COMMUNICATIONS LABORATORY

LABORATORY INSTRUCTION SHEET

Course Code	ECE 135L/ECE 235L
Course Name	Software Defined Radio Communications Lab
Experiment No	6
Experiment Title	Modelling 5G PHY channel coding using polar codes

EXPERIMENT 6: Modelling 5G PHY channel coding using polar codes.

AIM: To gain inside into principles of polar codes using SDR.

MOTIVATION:

The aim of this experiment is to explain the principle of polar codes, which are used in 5G L1 signalling. The great advantage of polar codes is achieving channel capacity and at the same time relative simple implementation. For these reasons polar code were chosen as channel coding to protect signalling communication against channel impairments in the 5G mobile communication systems. GNU Radio enables a simple implementation of polar coding in communication system by using already existing blocks. This experiment also enables to measure BER of the modelled communication system. This feature allows students to observe the influence of the change of polar code parameters on the system BER. The exact quantitative evaluation of these changes on the variations of communication system BER is rather complex and therefore is on this place omitted. In order to focus on the properties of polar codes a simple baseband communication system utilizing BPNRZ signalling is modelled.

Using TIMS SDR module we will model a baseband communication system employing polar codes for channel coding with block diagram shown in Figure 1.

OBJECTIVES:

1. To deepen the principles of GNU Radio concept.
2. To obtain hands-on experience with GNU Radio.
3. To obtain hands-on experience with polar codes using GNU Radio and SDR.

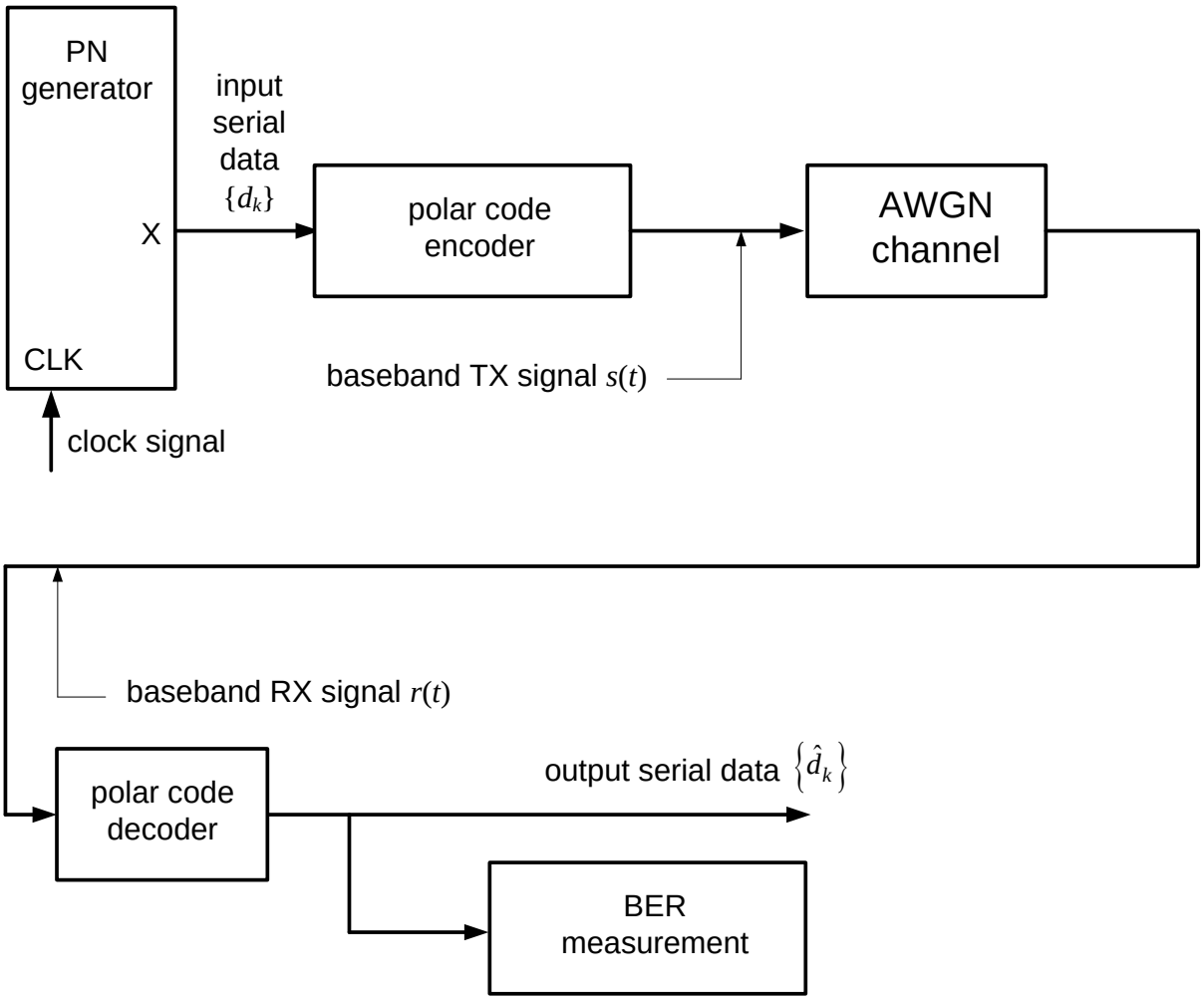
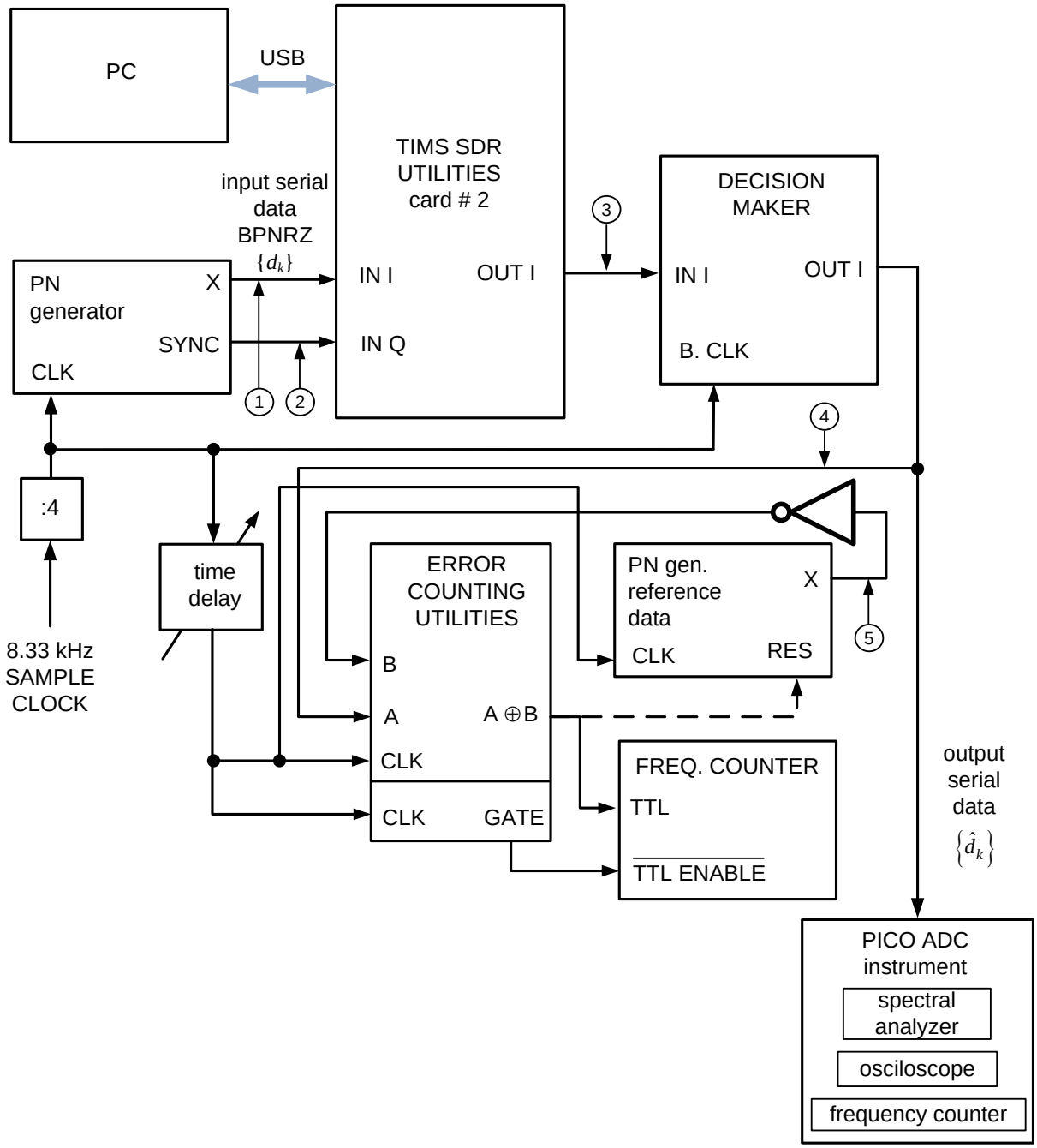


Figure 1. Block diagram of baseband communication system with polar codes.

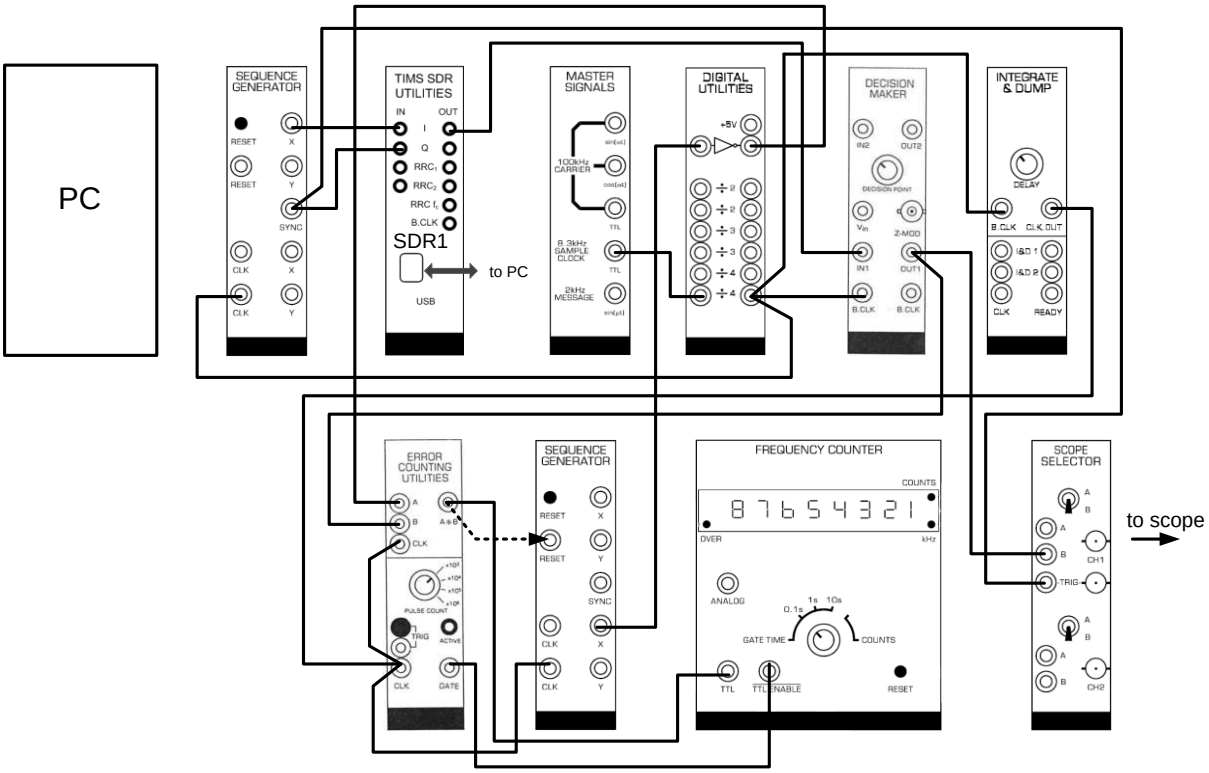
EQUIPMENT LIST:

1. SEQUENCE GENERATOR × 2
2. TIMS SDR UTILITIES
3. DECISION MAKER
4. ERROR COUNTING UTILITIES
5. DIGITAL UTILITIES
6. INTEGRATE & DUMP

BLOCK DIAGRAM of measurement setup:



WIRING DIAGRAM of measurement setup:



MODULE SETTINGS:

1. SEQUENCE GENERATORS: set short sequence: both DIP switches up
2. DECISION MAKER: WAVEFORM FORMAT SLECTOR: NRZ-TTL
   set DECISION POINT: to the middle of bit interval
3. ERROR COUNTING UTILITIES:
   TRIG: HI GATE: LO COUNT. MULT.: x 4
4. INTEGRATE & DUMP: SW3: A,B : OFF

Note: The following screenshots were taken using PicoScope 4424 device.

A BRIEF DESCRIPTION OF GNU RADIO COMPANION:

GNU Radio is a free, open source software development toolkit that provides signal processing blocks to implement SDR. GNU Radio applications are primarily written using Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ . To simplify interaction with GNU Radio a graphical user interface - GNU Radio Companion (GRC) was created. All necessary information regarding GNU Radio can be found on: https://www.gnuradio.org/ .

GNU Radio concept in short:

- All signal processing in GNU Radio is done through flow graphs.
- A flow graph consists of blocks. A block performs signal processing operations, such as filtering, adding signals, transforming, decoding, hardware access, etc.
- Data passes between block in various formats (complex, real, integers,...).
- Every flow graph requires at least one sink and one source.

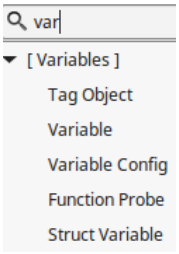
A brief tutorial about GRC is described in "FM reception using SDR" experiment.

Experiment setup:

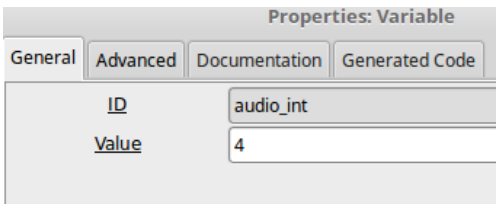
Transmitted data are supplied from PN generator using BPNRZ output. PN data are input to A/D converter of TIMS SDR UTILITIES module. Sampled data signal is fed to polar code encoder within GRC environment. The encoded signal is summed together with AWGN generated within GRC environment. Channel output is fed to the polar code decoder input. Decoded output is fed to the input of D/A converter of TIMS SDR UTILITIES module. Decoded signal is reshaped using TIMS DECISION MAKER module. BER measurement is performed using standard TIMS BER measurement procedure. Time alignment of the decoded and reference signal is obtained using delay circuit from TIMS INTEGRATE & DUMP module. Comparison of transmitted and received data is also performed visually using GRC GUI and the oscilloscope.

TX side:

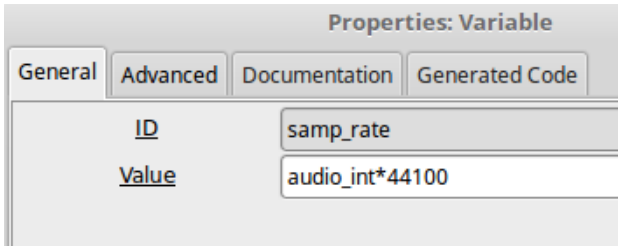
T1 Setting the global sampling rate. In the opened new untitled GRC window place **Variable** block in the flow graph. Double click on the icon: placed in the GRC menu toolbar. In the block search dialog type: `var` and place block **Variable** in the canvas:



Name this variable as *audio_int* and set its value to 4:



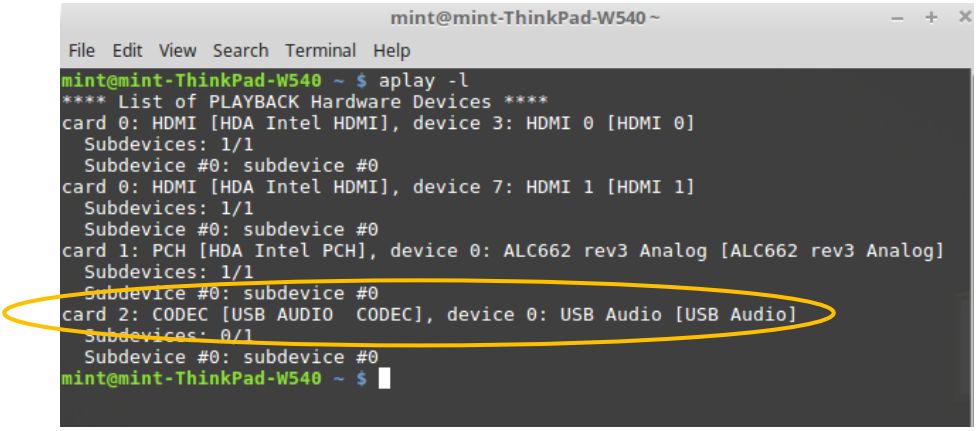
Set the global sampling rate to be 4 times audio card sampling rate:



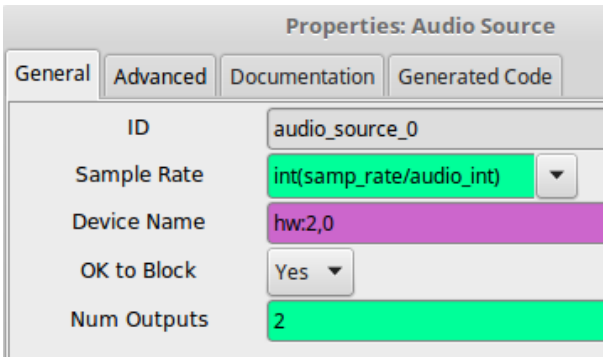
T2 As an interface between TIMS SEQUENCE GENERATOR module and GRC we will use A/D converter of the TIMS SDR UTILITY module. The signal input for the GRC software is realized with **Audio Source** block. In the block search dialog type: `aud` and place block **Audio Source** block in the canvas:

T3 After TIMS SDR UTILITY module is plugged in it has assigned a unique device name. Identify TIMS SDR UTILITIES module in PC hardware. In command prompt type:

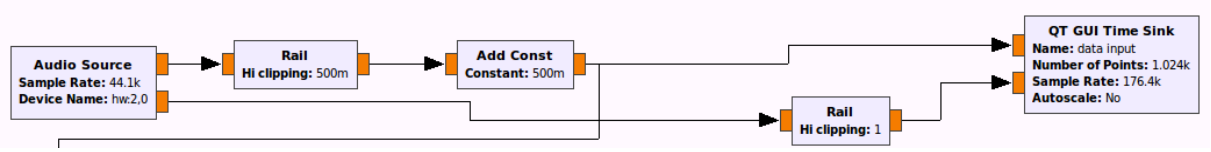
`aplay -l:`



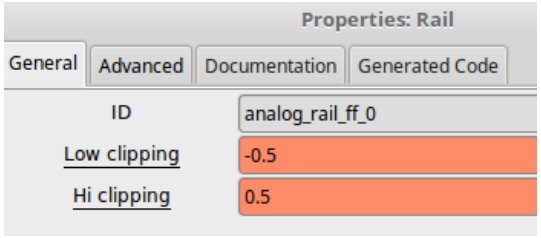
The TIMS SDR UTILITIES audio device is detected as card # 2, and the subdevice # 0. These values create a unique **Device Name** of **Audio Source** block. This device name is input in the format: `hw:card#,device#`. In this case it would be: `hw:2,0`.



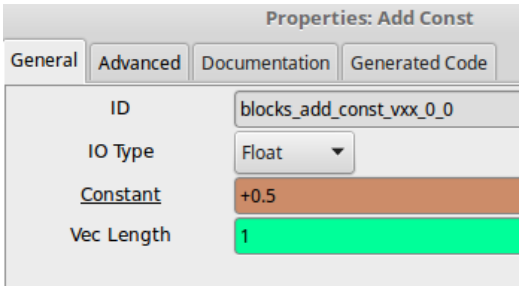
T3 Observing input data signal. Place on the canvas GRC "oscilloscope" - **QT GUI Time Sink** block and the following blocks:



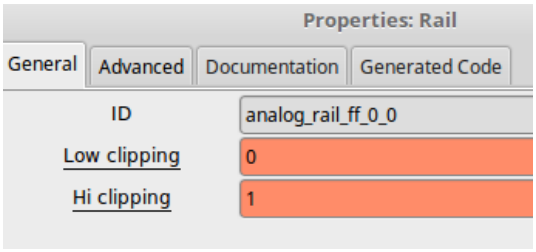
Connect output **0** of the **Audio Source** block to **Rail** block input. The **Rail** block clips the level of the output of the **Audio Source** block input **0**. Set the clipping parameters of the Rail block to ± 500 mV :



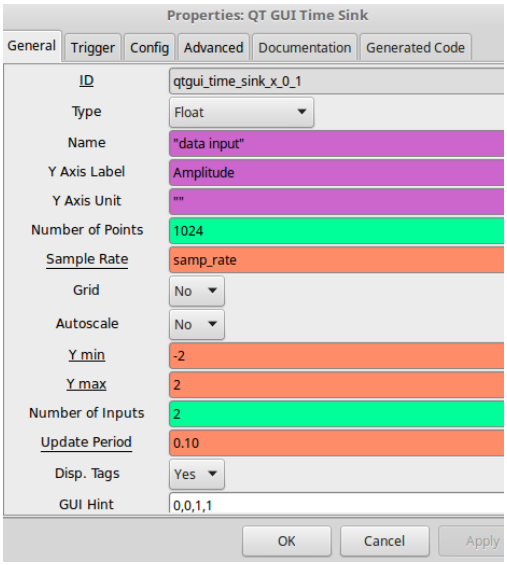
This way the BPNRZ output of the TIMS SEQUENCE GENERATOR module is level shifted to ± 500 mV in order to match the following processing blocks. To change the BPNRZ signal to UNRZ a level shift of +500 mV is performed using **Add Const** block. Connect the **Rail** block output to the **Add Const** block input. Set the parameters of the **Add Const** block:

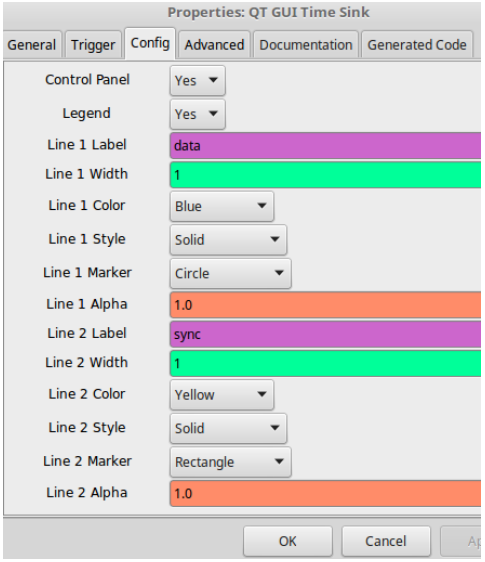


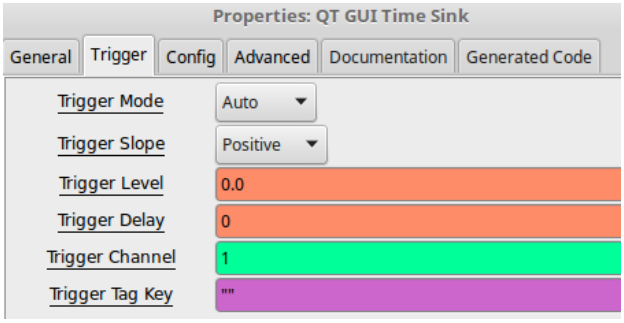
This way we obtained 1V UNRZ signal. We could not use TTL output of the **PN GNERATOR** module since this will overload TIMS **SDR** module input. Therefore two GNU Radio blocks: **Rail** and **Add Const** were used. Connect the output of the **Add Const** block to the input **0** of the **QT GUI Time Sink** block. Since we want to observe PN sequence we have to synchronise the oscilloscope. The second input of the **QT GUI Time Sink** block is displaying the sync signal of the SEQUENCE GENERATOR module. Sync signal is already an UNRZ signal therefore only level adjustment is necessary. This level adjustment is performed by using the second **Rail** block. Set its clipping level to 1V :



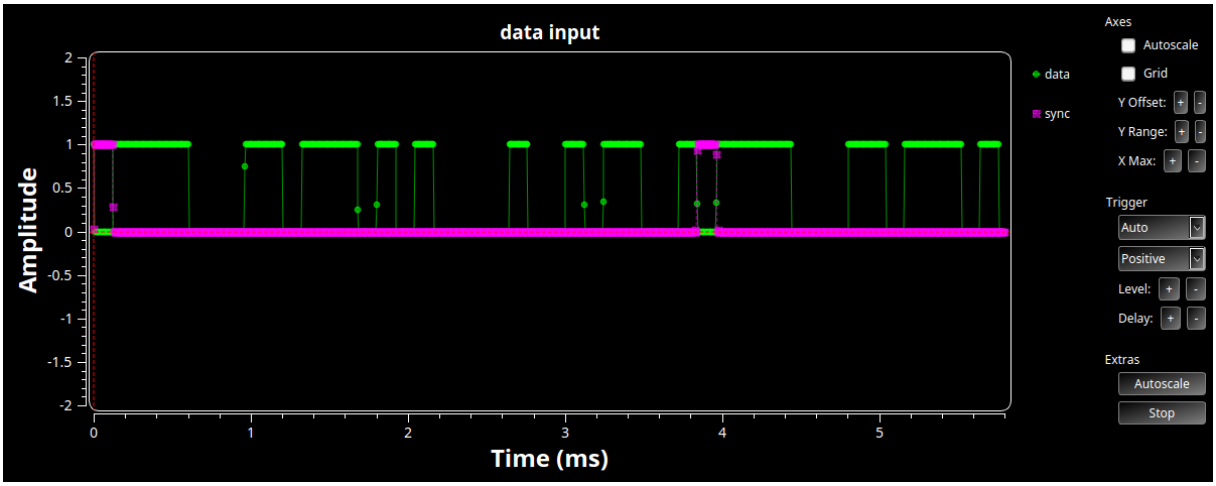
Connect this Rail block output to the input **1** of the **QT GUI Time Sink** block. Set the following **QT GUI Time Sink** block parameters:







After saving the grc file in the GRC menu toolbar click on the icon: generate the flow graph and then on the icon: execute the flow graph. If all processing went well **Top Block** window should appear showing input TX data and the PN synchronization pulse:



Verification of the GRC "oscilloscope" can be done using external oscilloscope, see Figure 2.

Oscilloscope setting (PicoScope 4424):
number of samples: 500k, resolution 14bits
trig.: CHD, auto
time base: 2 ms/div, horizontal zoom: x1
vertical scale: CHA, CHD: +/- 20V DC

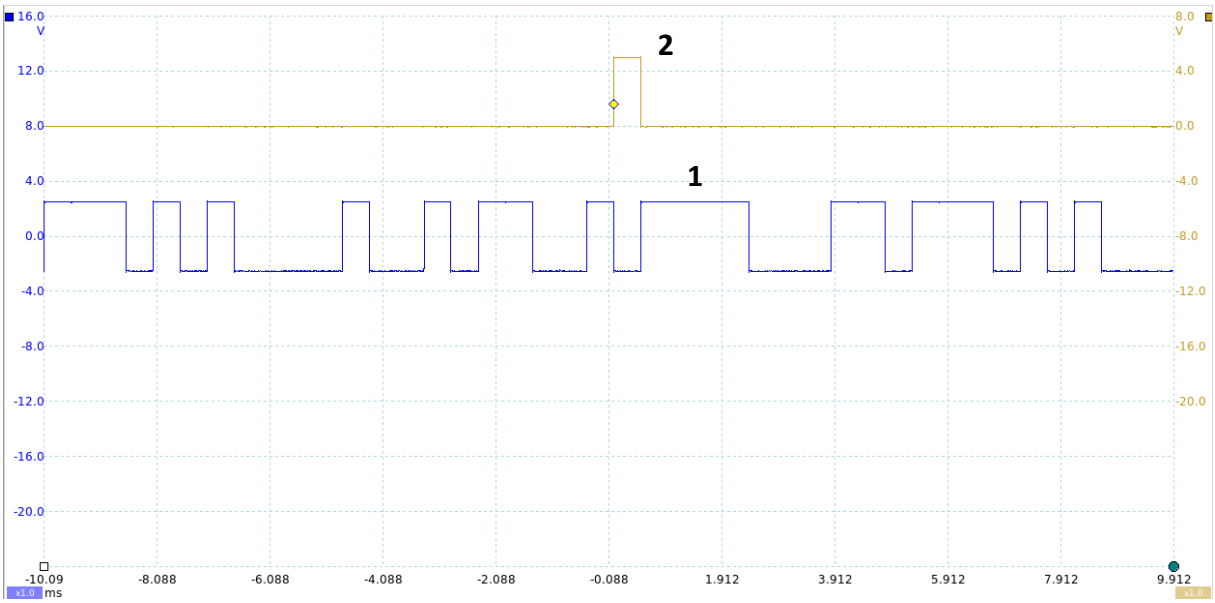
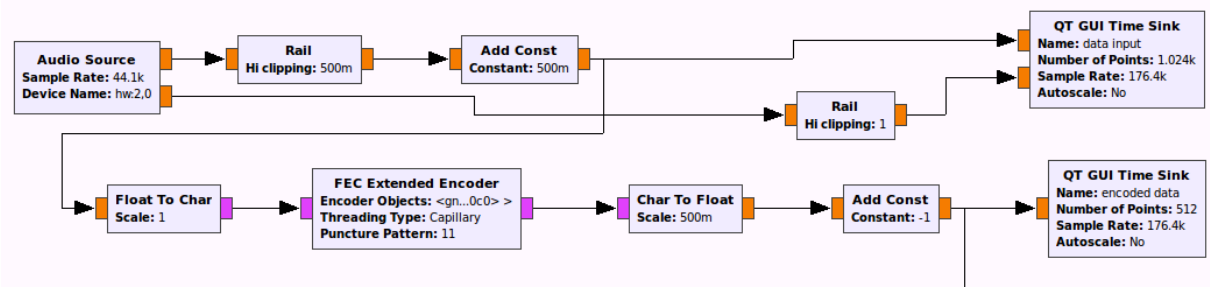
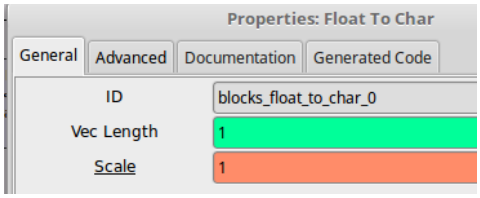


Figure 2. BPNRZ data signal and synchronisation signal
- check points **1** and **2** on the block diagram.

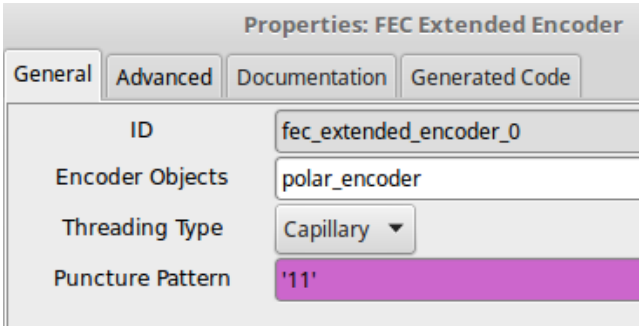
T4 Encoding data. Data are encoded using FEC polar code encoder. Place the following blocks on the canvas:



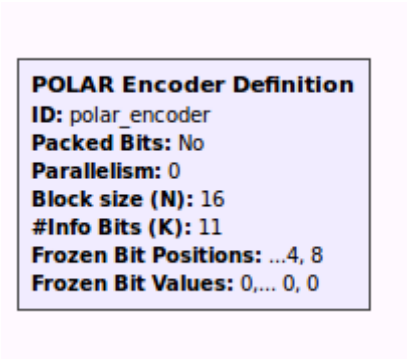
The input NRZ bit stream is first transformed to `char` format using **Float To Char** block with the following setup:



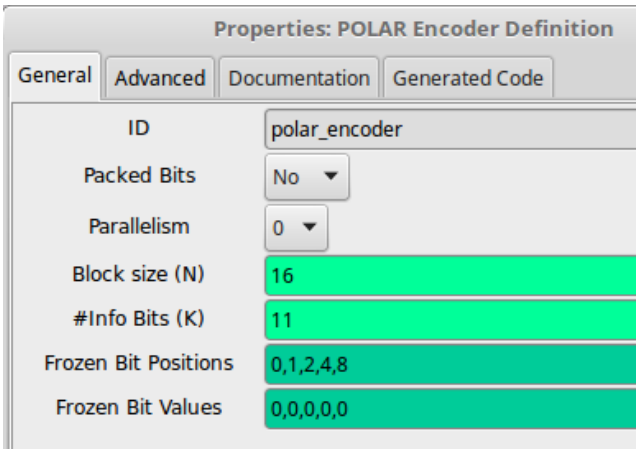
Transformed data to `char` format are encoded using **FEC Extended Encoder** block. In the block properties as `Encoder Objects` select `polar_encoder`:



In order to define the properties of the FEC encoder another block: **Polar Encoder Definition** has to be placed on the canvas:



This block sets the parameters of the selected FEC encoder. We will start with the polar code of length 16:

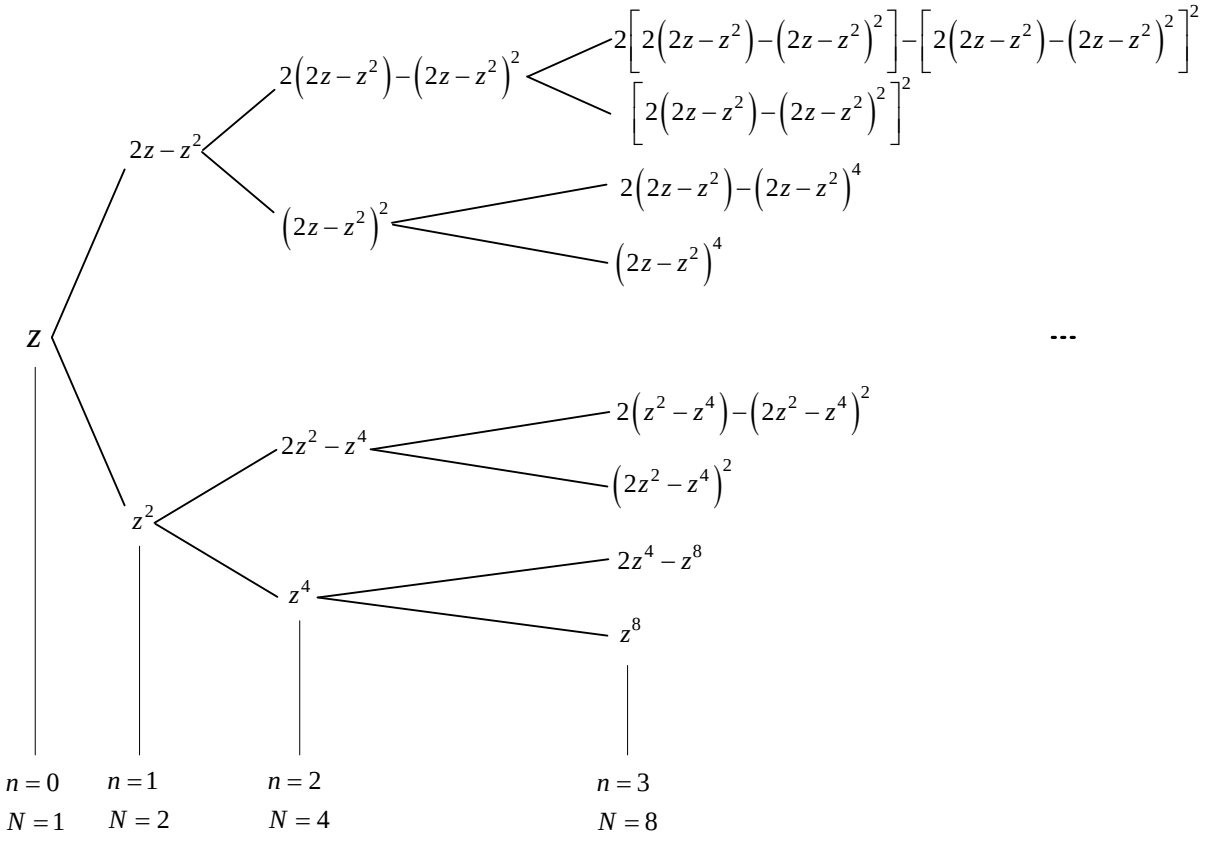


The theory of polar codes is exceeding the scope of this lab therefore at this place only a very brief description of polar codes parameters in order to setup the experiment is given. The length of polar codes (block size N) is: $N = 2^n$, where n is an positive integer. Usually: $n \geq 2$. At the same time the number of frozen bits plus the number of information bits has to be equal N. In the process of channel polarization the capacity of some virtual channels are approaching to 1 while the capacity of others is approaching to 0. Channels with capacity approaching to 0 do not convey an

information, therefore bit values assigned for the outputs of these virtual channels can be set to a fixed value e.g. 0 - which is known to the receiver. Bits at such a positions are denoted as *frozen bits*. The indices of frozen bits can be calculated using Bhattacharyya parameter. In case of AWGN channel with noise variance $N_0/2$ and BPNRZ signalling with signal constellation $\left\{\pm\sqrt{E_c}\right\}$ the initial value of Bhattacharyya parameter *z* is:

$$z = e^{-\frac{E_c}{N_0}} \qquad n = 0 \qquad N = 1 \tag{1}$$

where $\frac{E_c}{N_0}$ is *designed* SNR. Calculation of z for other block lengths follows this recursion:



The recursion stops when at the leaf level are *N* values, indexed from the top: $0,\ldots,N-1$. In the process of the encoder definition for a given block length N and the value of designed SNR based on the above scheme is calculated the set Bhattacharyya parameters. Using this notation, the values of Bhattacharyya parameters represents the error probabilities of particular virtual channels. From this set is evident that some values are approaching to 1 - "bad" channels with capacity approaching to 0. The other values are approaching to 0 - "good" channels with capacity approaching to 1. As a threshold between those two extreme cases a value 0.5 can be chosen. Therefore bit positions with Bhattacharyya parameter < 0.5 can

be marked as an information bits. Bit positions with Bhattacharyya parameter > 0.5 can be marked as a frozen bits and they can have assigned fixed value of 0. In our case for: $n=4;\quad N=16;\ \frac{E_c}{N_0}=0\,[dB]$ the set of Bhattacharyya parameters (*BP*) is graphically represented in Figure 3.

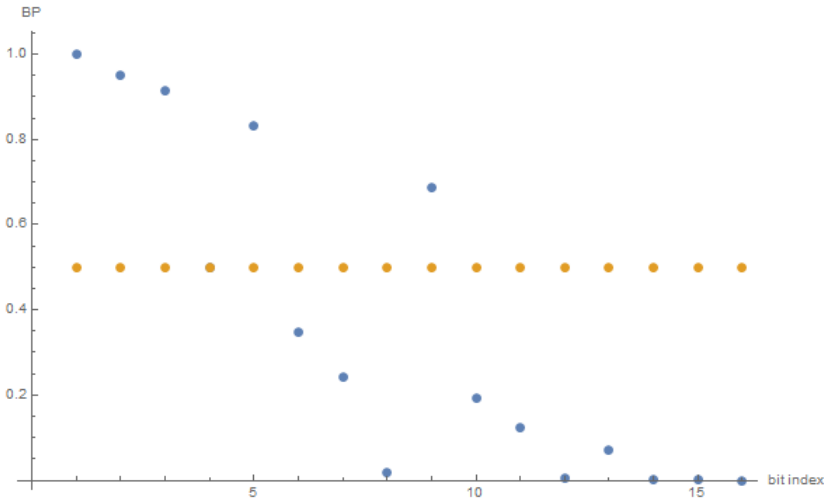


Figure. 3 Set of Bhattacharyya parameters for [16,11] polar code.

The values of Bhattacharyya parameters are summarized in Table 1. From Figure 3 and Table 1 is evident that bits with indices: 1,2,3,5,9 can be frozen since the corresponding $BP>0.5$. Important note: in GNU Radio frozen bits are indexed starting from 0, therefore the list of frozen bits is: 0,1,2,4,8 in the **POLAR Encoder Definition** block! For this case: $N=16$, the number of frozen bits is 5 therefore $K=16-5=11$ and the code rate is: $R=K/N=5/16\cong 0.3125$. Obviously such a low code rate is impractical in real applications.

n	BP
1	9,994E-01
2	9,497E-01
3	9,137E-01
4	4,987E-01
5	8,328E-01
6	3,493E-01
7	2,430E-01
8	1,689E-02
9	6,875E-01
10	1,945E-01
11	1,233E-01
12	4,056E-03
13	7,127E-02
14	1,317E-03
15	6,708E-04
16	1,125E-07

Table 1. Bhattacharyya parameters for [16,11] polar code.

In order to better see which bits have to be frozen a bar chart (see Figure 4) from Table 1. can be constructed:

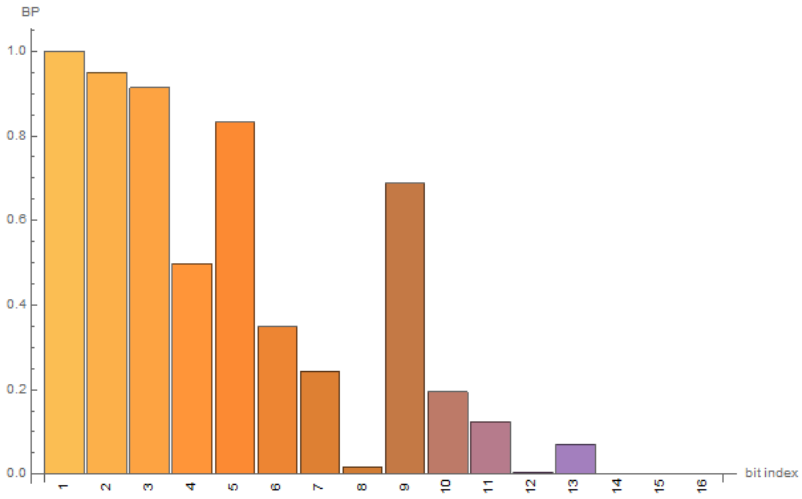
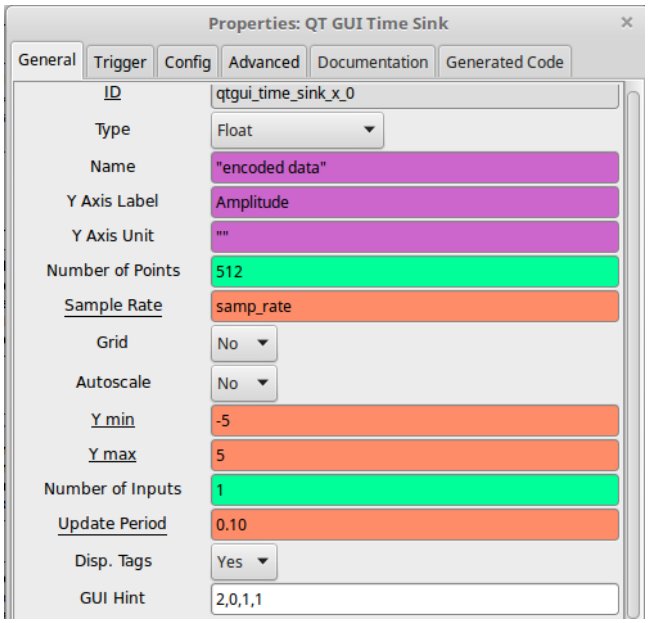
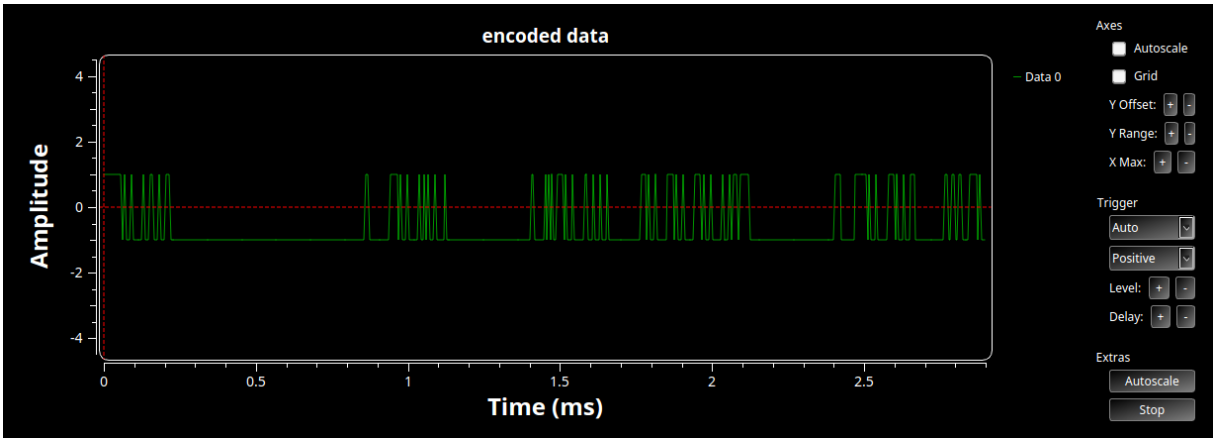


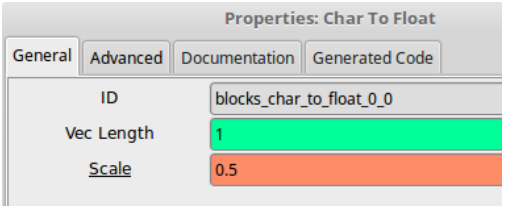
Figure 4. A bar chart of Bhattacharyya parameters for [16,11] polar code.

T5 Using **QT GUI Time Sink** block observe the encoded data:

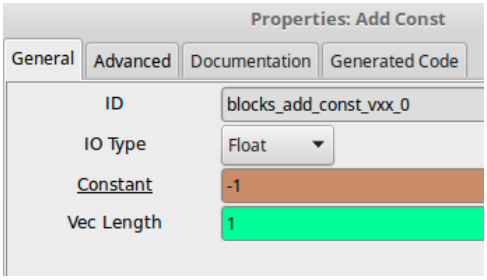




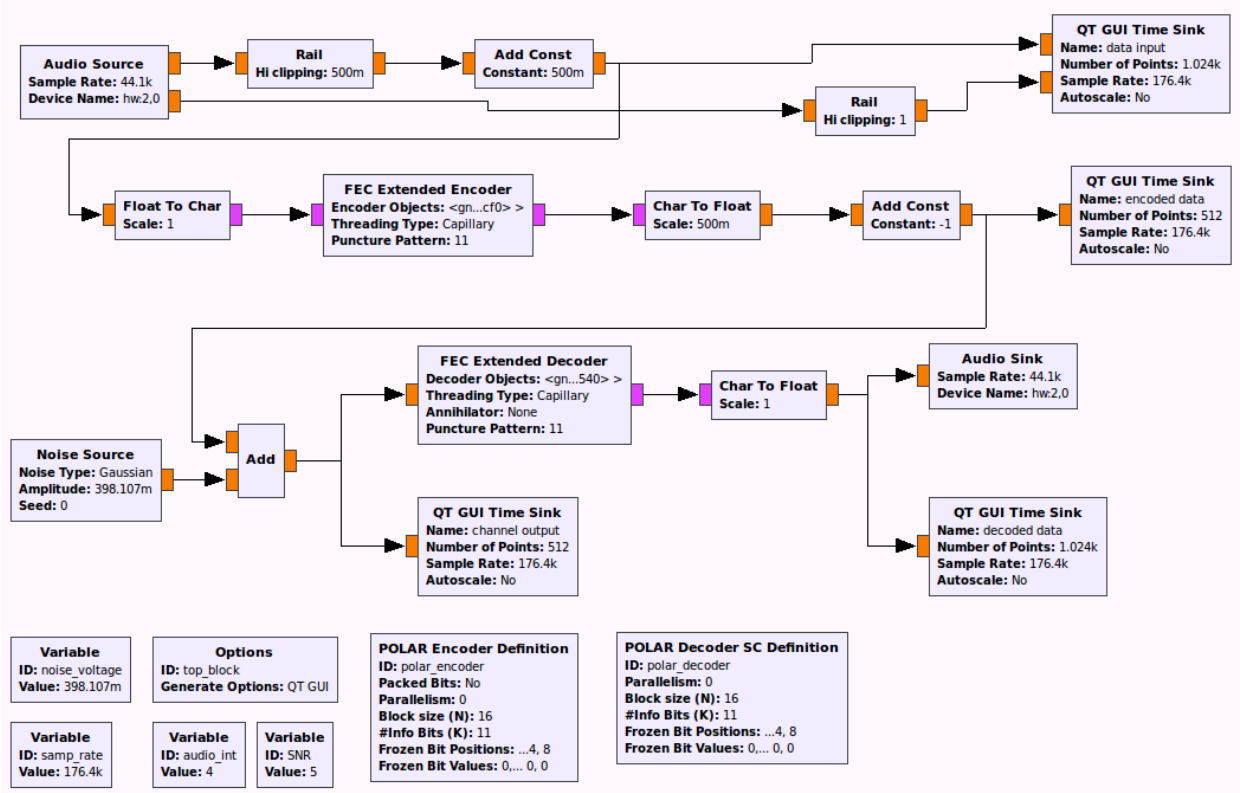
Notice that by adding redundancy the bit duration is shorter comparing it to data bits. The encoded data are converted to BPNRZ signal using **Char To Float** block:



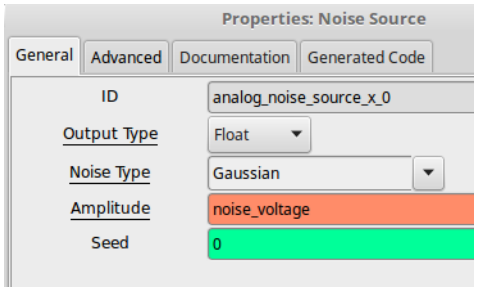
Level shifting is performed in **Add Const** block:



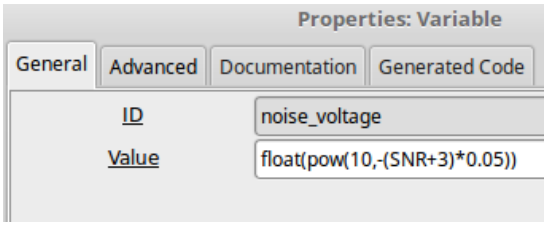
T6 Transmitting data over AWGN channel. Complete the flowchart by adding AWGN channel and decoder:



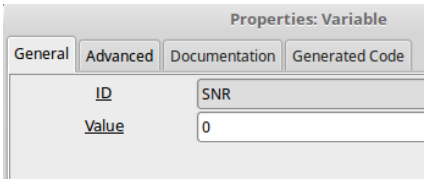
AWGN noise source is modelled by **Noise Source** block:



The noise amplitude is set by **Variable** block denoted as `noise voltage`:

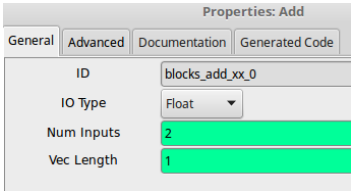


The value of a noise voltage is derived from designed SNR using **Variable** block named SNR:

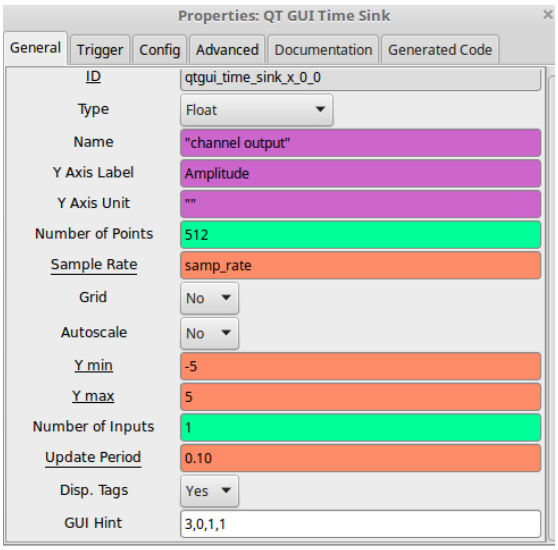


Used definition of a noise voltage enables to input SNR directly in [dB].

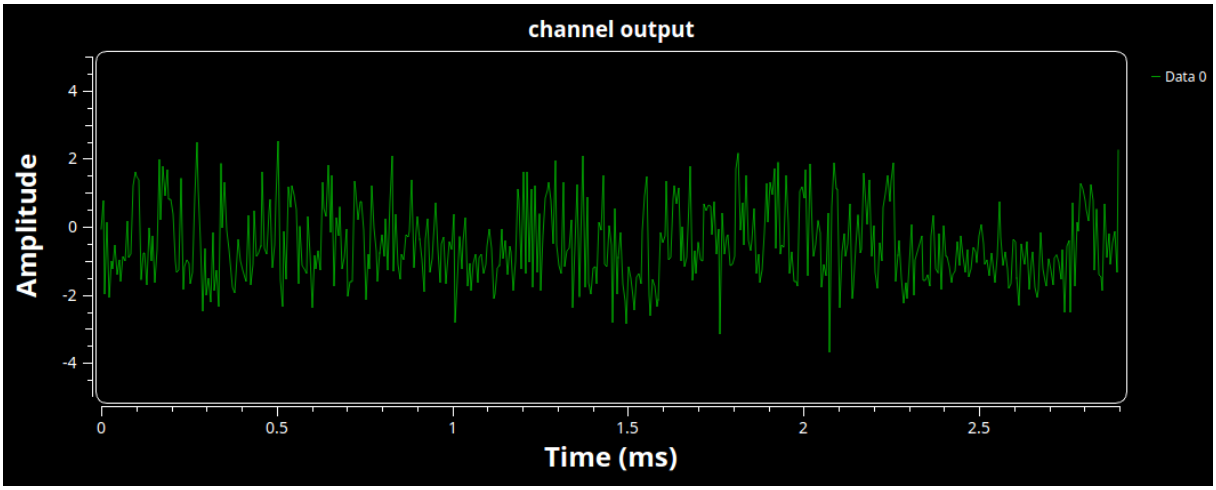
AWGN channel is modelled by simple **Add** block:



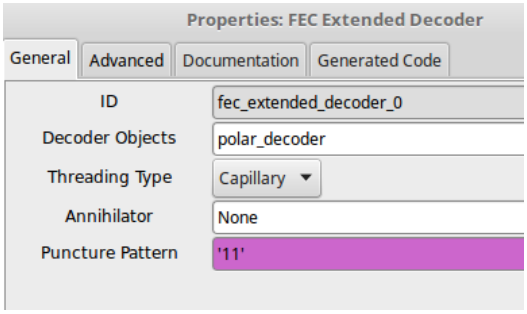
T7 Using **QT GUI Time Sink** block observe the AWGN channel output:



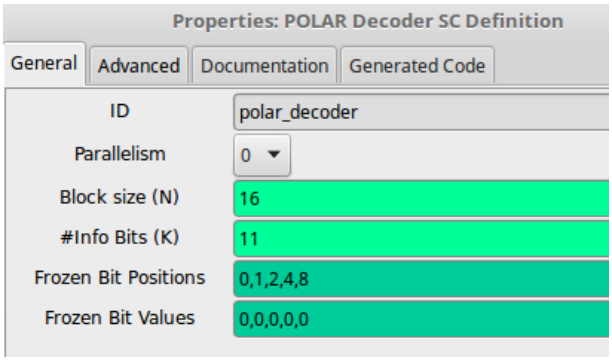
Note that set SNR = 0 dB.



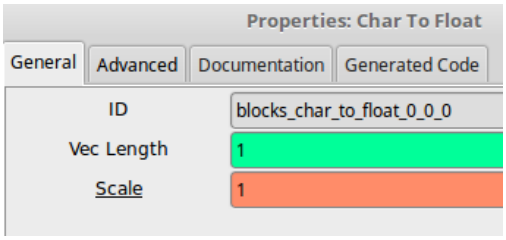
T8 Decoding of the received signal from AWGN channel. Received signal from AWGN channel is decoded in **FEC Extended Decoder** block. Its definition has to match FEC encoder definition:



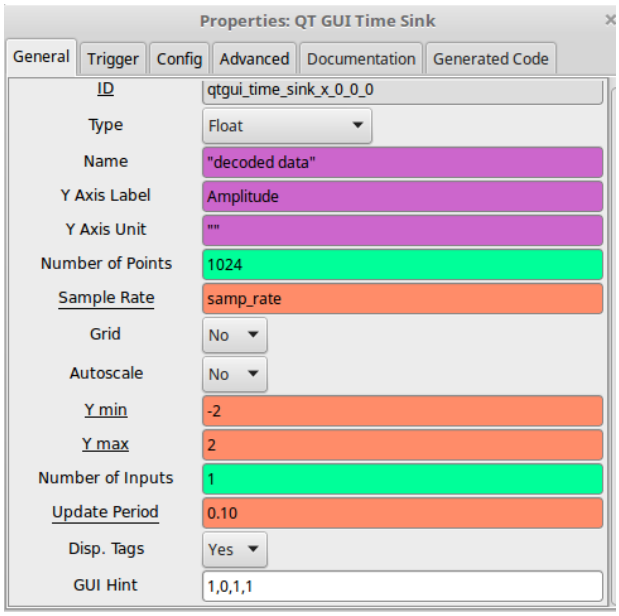
Parameters of the polar decoder are defined in **Polar Decoder SC Definition** block. The set parameters have to match the encoder definition:



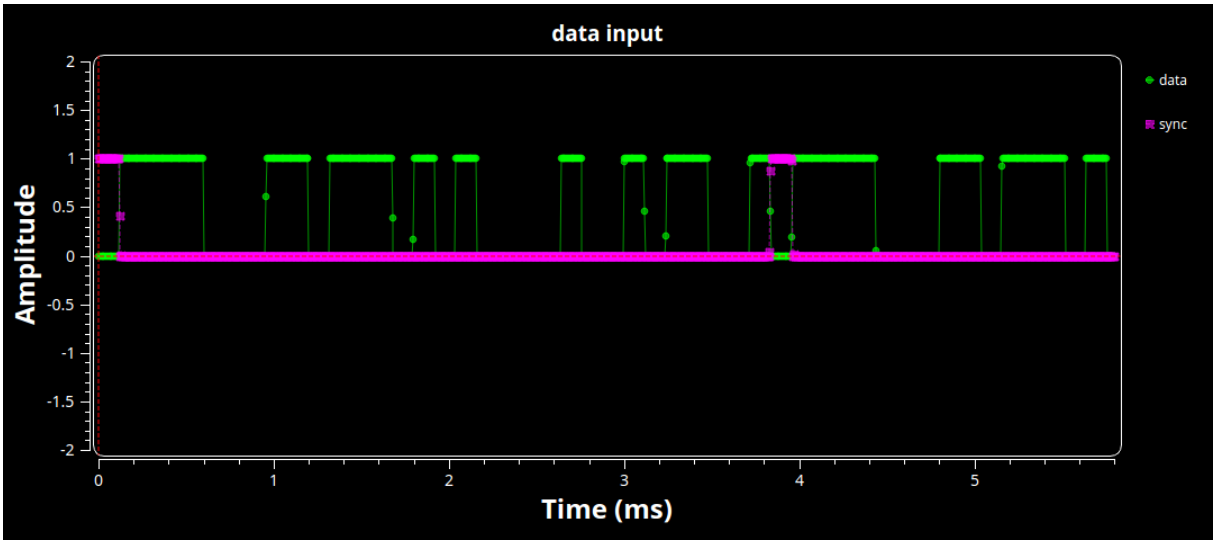
For further processing data format from decoder is changed from `char` to `float` in format conversion block: **Char To Float**:



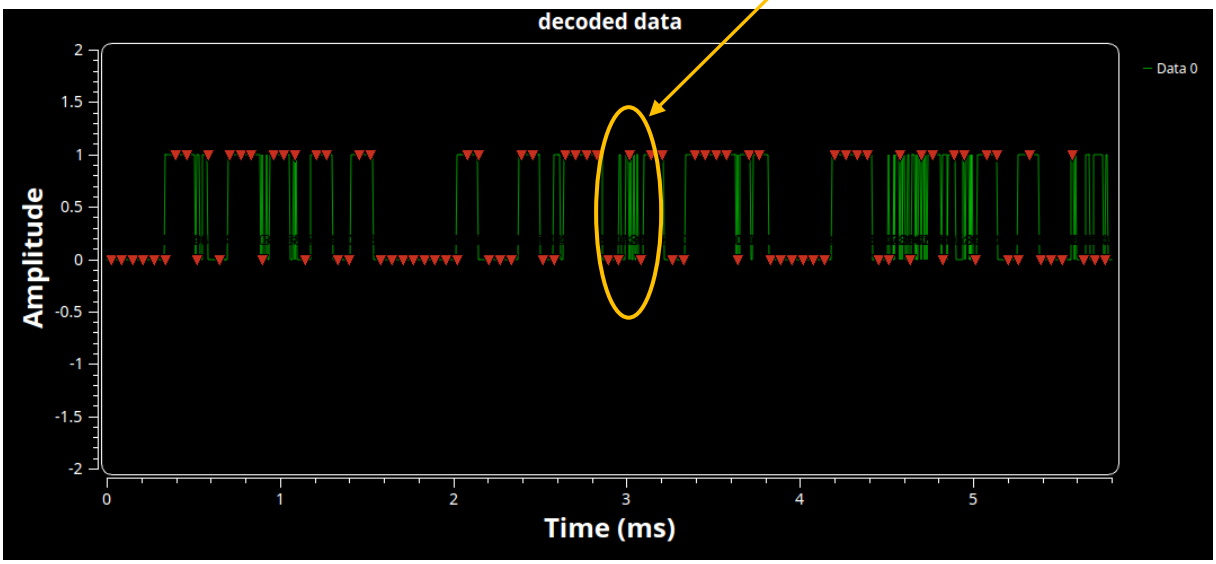
T9 Using **QT GUI Time Sink** block observe decoded data at polar decoder output:



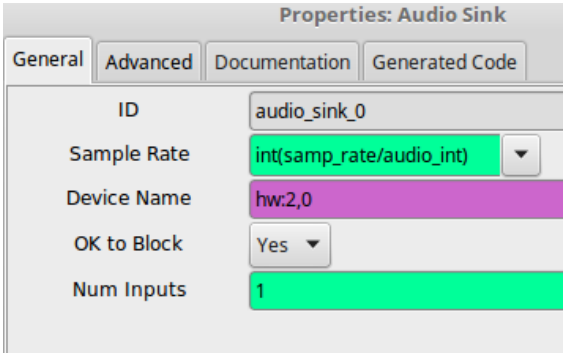
For an easier comparison of the sent and received data also sent data are displayed. Note that set SNR is still 0 dB - what is causing lot of errors. Also code parameters at this moment are not setup optimally.


data input
Amplitude
Time (ms)
data
sync

errors


decoded data
Amplitude
Time (ms)
Data 0

T10 Using an external oscilloscope observe: transmit data, decoded data and the output of **DECISION MAKER** block, see Figure 5. As an output interface serves **Audio Sink** block:



Signal from the **Audio Sink** block is reshaped using TIMS **DECISION MAKER** block.

Oscilloscope setting (PicoScope 4424):
number of samples: 500k, resolution 14bits
trig.: CHD, auto
time base: 2 ms/div, horizontal zoom: x1
vertical scale: CHA, CHC, CHD: +/- 20V DC; CHB: +/- 10V

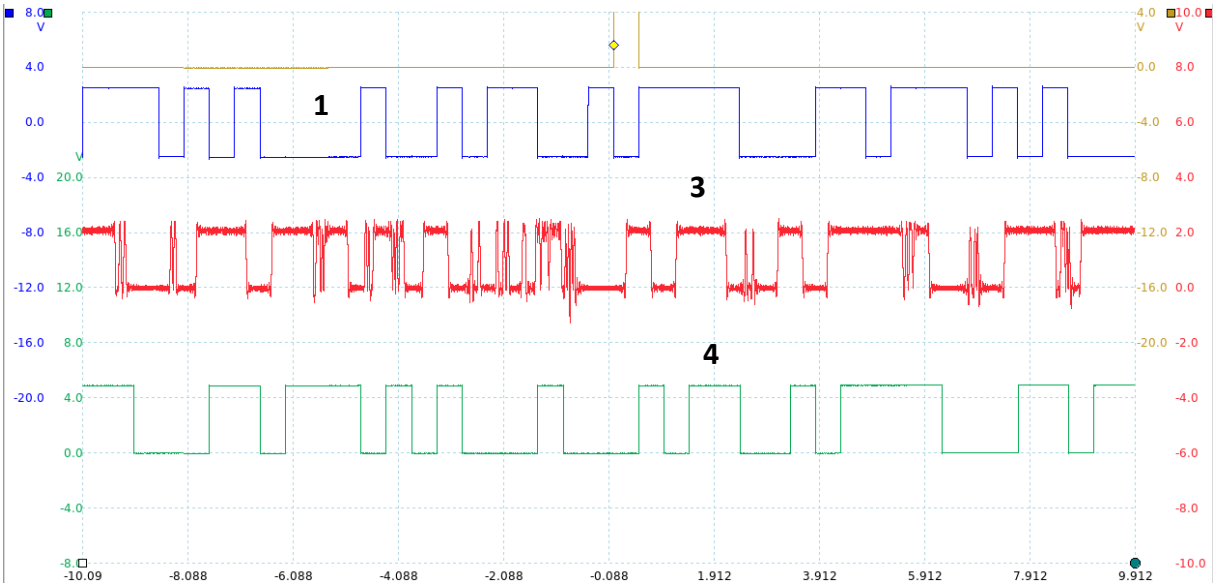


Figure 5. Transmit data, decoded data and the output of **DECISION MAKER** block - check points **1**, **3** and **4** on the block diagram.

T11 Time alignment of the received and reference data. Reference data are generated in the identical PN generator as is used for data generation in the transmitter. For a time alignment of the received data and reference data a time delay circuit from **INTEGRATE & DUMP** module is used. By using of **DELAY** button time align received and reference data, see Figure 6.

Oscilloscope setting (PicoScope 4424):
number of samples: 500k, resolution 14bits
trig.: CHD, auto
time base: 2 ms/div, horizontal zoom: x1
vertical scale: CHB , CHC, CHD: +/- 20V DC

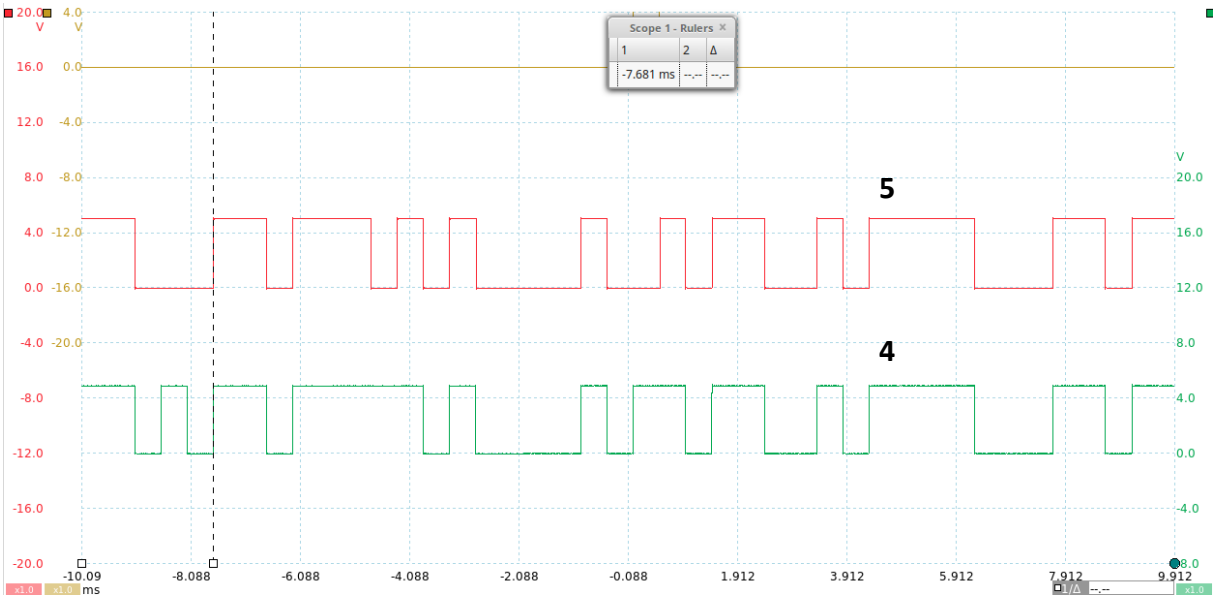


Figure 6. Decoded data and reference data
- check points **4** and **5** on the block diagram.

T12 BER measurement. BER measurement is performed outside GNU Radio environment using a standard TIMS BER measurement technique. To synchronize reference data and received data temporarily connect **A** ⊕ **B** output on **ERROR COUNTING UTILITIES** module with **RES** input on reference PN generator. Set **PULSE COUNT** to 10^3 . Set **FREQUENCY COUNTER** module to **COUNTS** and press **RESET** button. On **ERROR COUNTING UTILITIES** module press **TRIG** button. When counting is finished, display shows the number of errors occurred during the counting interval. Then:

$$BER = \frac{\#\text{of counts} - 1}{4 \times \text{pulse count}} \tag{2}$$

Repeat each measurement 5 times and input the average value of measured BER into prepared Table 2. Theoretical value of bit error probability for an uncoded system can be calculated as:

$$P_b = \frac{1}{2} erfc\left(\sqrt{10^{\frac{SNR[dB]}{10}}}\right) \tag{3}$$

The initial value of SNR set in **T6** is 0 [dB]. Repeat the same procedure as described above for SNR values: +1, and +2 dB. In order to maintain an acceptable accuracy for

SNR values: +3, and +4 dB set **PULSE COUNT** on **ERROR COUNTING UTILITIES** module to 10^4 and repeat the above described procedure. For SNR value: +5 dB set **PULSE COUNT** to 10^5 and for SNR value: +6 dB to 10^6. Before each BER measurement synchronize reference data and received data. Input measured BER values and calculated bit error probability of an uncoded system in Table 2.

E_b/N_0 [dB]	noise voltage [Vrms]	theoretical P_b uncoded system	measured BER coded system
0	0,707945784	0,078649632	7,24E-02
+1	0,630957344	0,056281969	3,11E-02
+2	0,562341325	0,037506135	1,05E-02
+3	0,501187234	0,022878411	2,55E-03
+4	0,446683592	0,01250082	3,60E-04
+5	0,398107171	0,005953868	2,71E-05
+6	0,354813389	0,002388291	2,25E-06

Table 2. P_b and BER for polar code [16,11,{0,1,2,4,8}] with $E_c/N_0 = 0\ [dB]$.

T13 Change the code parameters to: $N = 32$, $K = 21$. As in **T4** calculate the set of Bhattacharyya parameters, see Table 3 and Figure 7 and 8.

n	BP	n	BP	n	BP	n	BP
1	1,000E+00	9	9,720E-01	17	9,024E-01	25	1,375E-01
2	9,987E-01	10	6,935E-01	18	4,727E-01	26	5,080E-03
3	9,975E-01	11	5,766E-01	19	3,512E-01	27	2,633E-03
4	9,019E-01	12	1,220E-01	20	3,783E-02	28	1,736E-06
5	9,925E-01	13	4,270E-01	21	2,314E-01	29	1,341E-03
6	8,348E-01	14	5,907E-02	22	1,521E-02	30	4,500E-07
7	7,487E-01	15	3,350E-02	23	8,095E-03	31	2,251E-07
8	2,487E-01	16	2,853E-04	24	1,645E-05	32	1,266E-14

Table 3. Bhattacharyya parameters for [32,21] polar code.

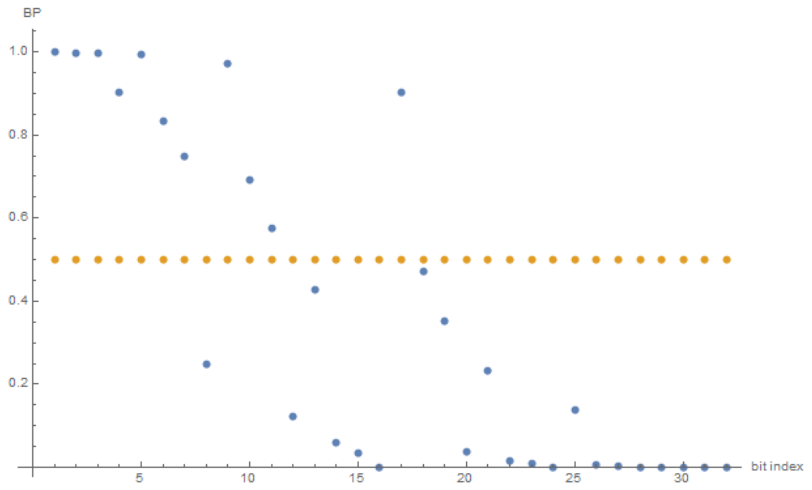


Figure. 7 Set of Bhattacharyya parameters for [32,21] polar code.

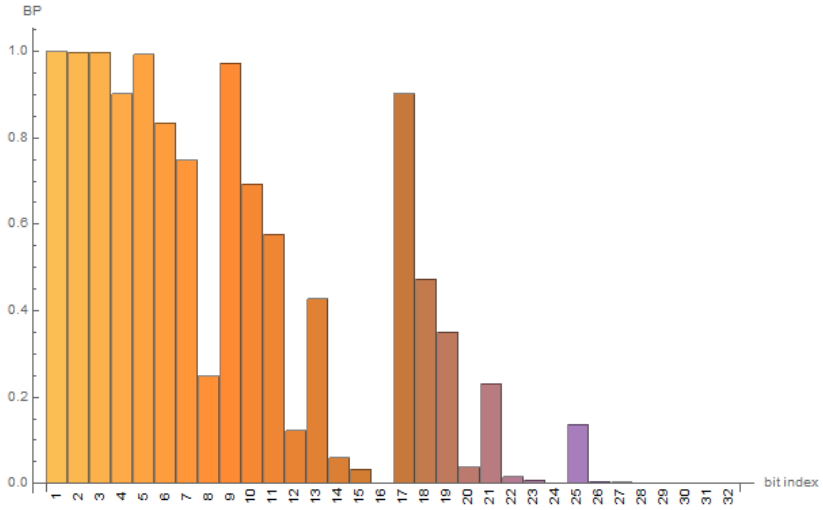
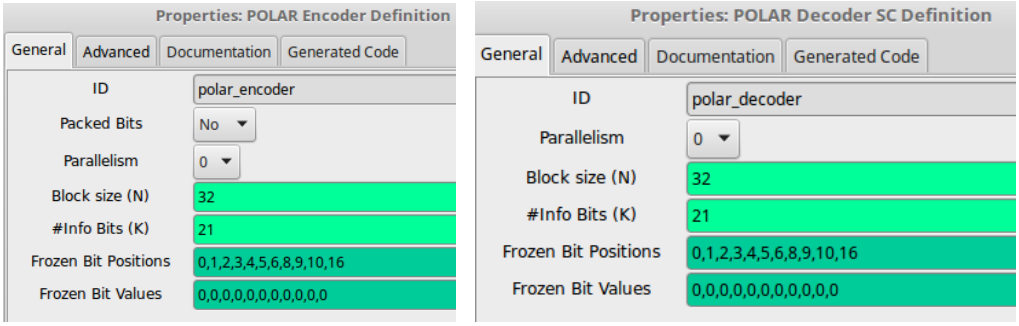


Figure 8. A bar chart of Bhattacharyya parameters for [32,21] polar code.

From Table 3, Figures 7 and 8 we can see that the frozen bit indices are: 1,2,3,4,5,6,7,9,10,11,17. Therefore encoder and decoder blocks definitions have to be changed accordingly:



T14 Measure BER for [32,21] code. As in **T12** measure BER and input measured and calculated values in the prepared Table 4.

E_b/N_0 [dB]	noise voltage [Vrms]	theoretical P_b uncoded system	measured BER coded system
0	0,707945784	0,078649632	6,27E-02
+1	0,630957344	0,056281969	2,48E-02
+2	0,562341325	0,037506135	7,75E-03
+3	0,501187234	0,022878411	1,41E-03
+4	0,446683592	0,01250082	1,66E-04
+5	0,398107171	0,005953868	1,48E-05
+6	0,354813389	0,002388291	5,00E-07

Table 4. P_b and BER for polar code [32,21,{ 0,1,2,3,4,5,6,8,9,10,16}] with $E_c/N_0 = 0\ [dB]$.

By comparing BP with [16,11] polar code we can start to see more distinct channel polarization. This will also reduce BER - what can be observed with the oscilloscope,

see Figure 9 and 10. The code rate has changed from $R = K/N = 5/16 \cong 0.3125$ to $R = K/N = 21/32 \cong 0.656$. Notice time delay τ, between transmit and received data.

Oscilloscope setting (PicoScope 4424):
number of samples: 500k, resolution 14bits
trig.: CHD, auto
time base: 2 ms/div, horizontal zoom: x1
vertical scale: CHA, CHB, CHC, CHD: +/- 20V DC

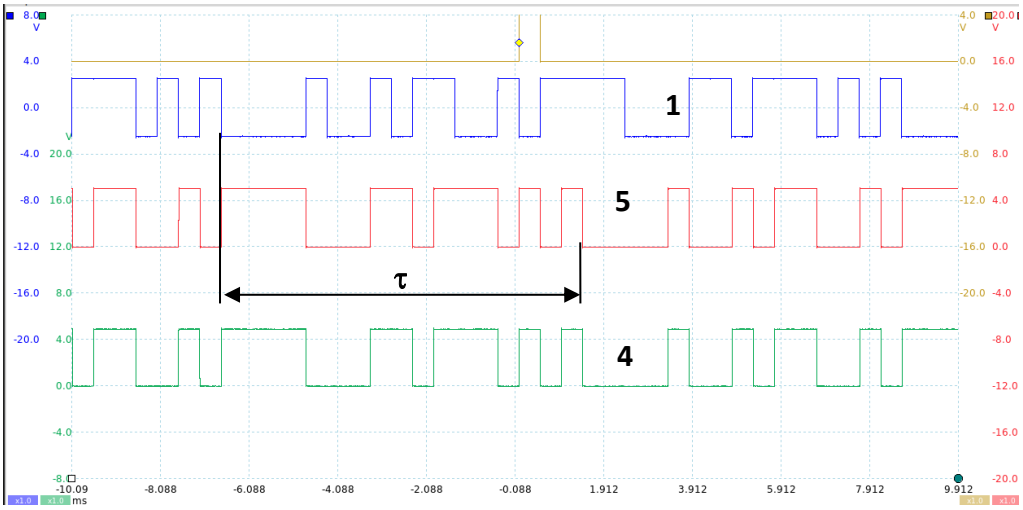


Figure 9. Transmitted data, decoded data and reference data
- check points **1**, **4** and **5** on the block diagram.

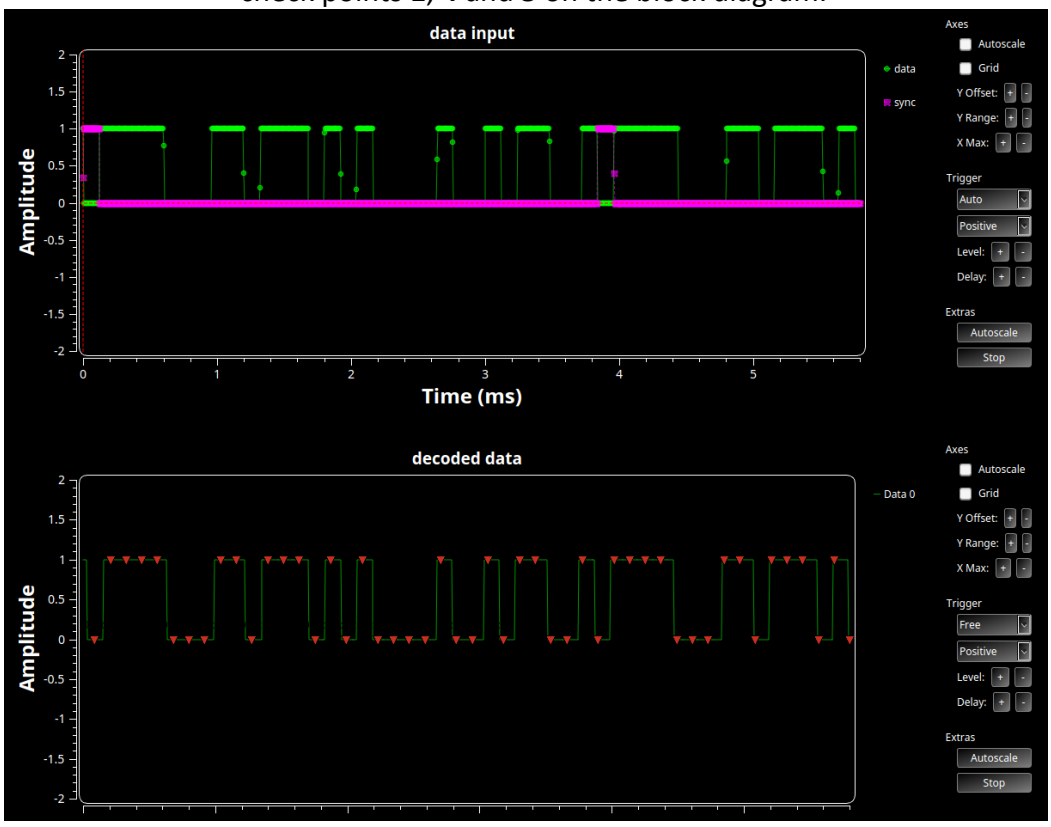


Figure 10. decoded data match input data.

T15 Change the code parameters to: $N = 64$, $K = 41$. As in **T4** calculate the set of Bhattacharyya parameters, see Table 5 and Figure 11 and 12.

n	*BP*	*n*	*BP*	*n*	*BP*	*n*	*BP*
1	1	17	0,99921773	33	0,990469238	49	0,25603936
2	0,999999155	18	0,944844105	34	0,814279523	50	0,01889758
3	0,999998312	19	0,906051553	35	0,721979645	51	0,010134229
4	0,997403219	20	0,480928228	36	0,223467644	52	2,58066E-05
5	0,999993582	21	0,820768555	37	0,579029087	53	0,005259141
6	0,994939469	22	0,332516746	38	0,123325487	54	6,93288E-06
7	0,990369759	23	0,229184515	39	0,074232874	55	3,47101E-06
8	0,813362658	24	0,014893522	40	0,001431266	56	3,01198E-12
9	0,999944438	25	0,67168684	41	0,40927788	57	0,002680552
10	0,985147552	26	0,182340805	42	0,053553138	58	1,79875E-06
11	0,972702631	27	0,11464917	43	0,030179749	59	8,99979E-07
12	0,69685906	28	0,003489167	44	0,000231207	60	2,02491E-13
13	0,936834782	29	0,065872345	45	0,016123686	61	4,50141E-07
14	0,560511388	30	0,001122063	46	6,55226E-05	62	5,06567E-14
15	0,435513144	31	0,000570548	47	3,28943E-05	63	2,53283E-14
16	0,06184005	32	8,14044E-08	48	2,70513E-10	64	1,60381E-28

Table 5. Bhattacharyya parameters for [64,41] polar code.

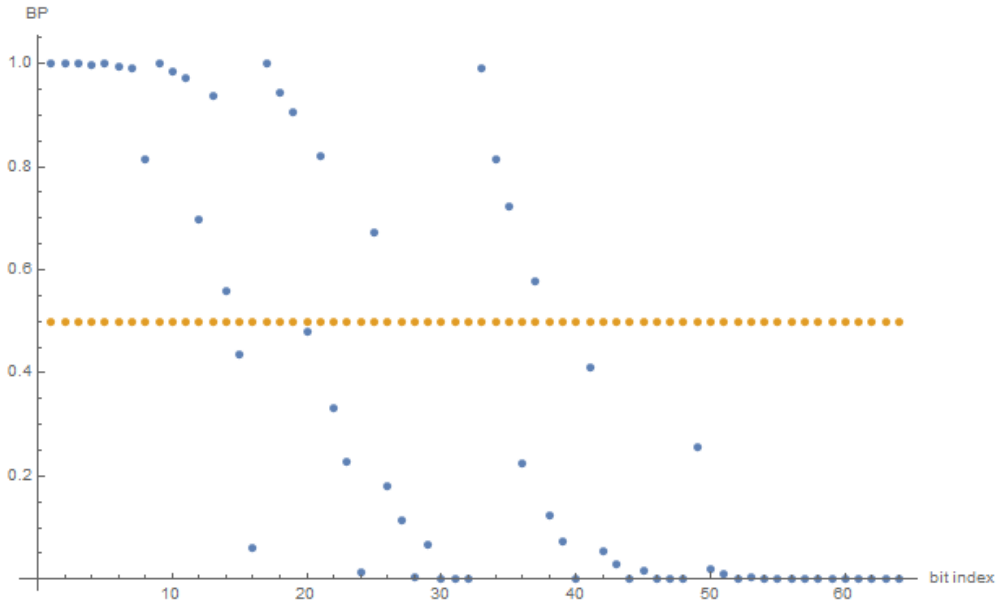


Figure. 11 Set of Bhattacharyya parameters for [64,41] polar code.

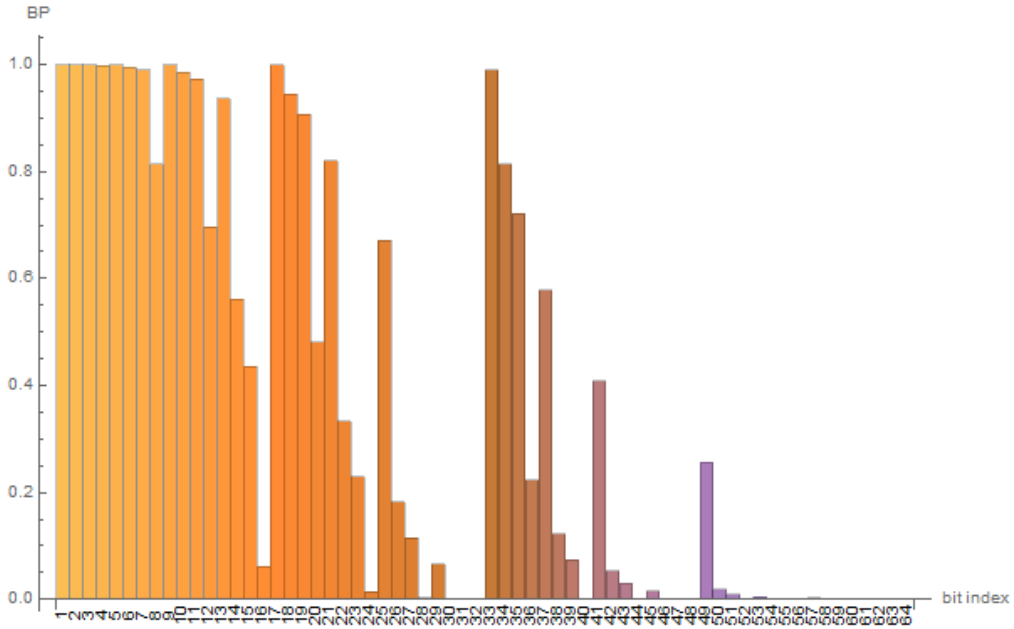
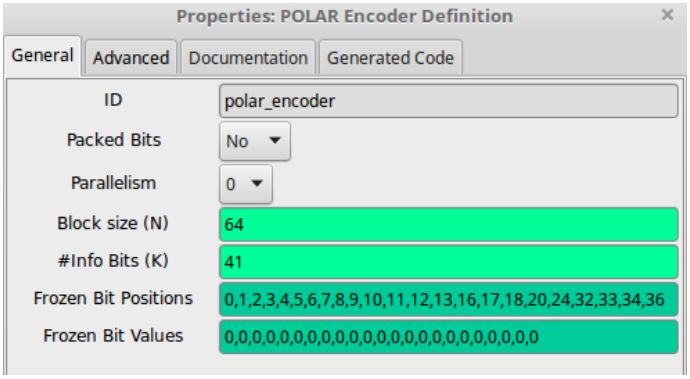
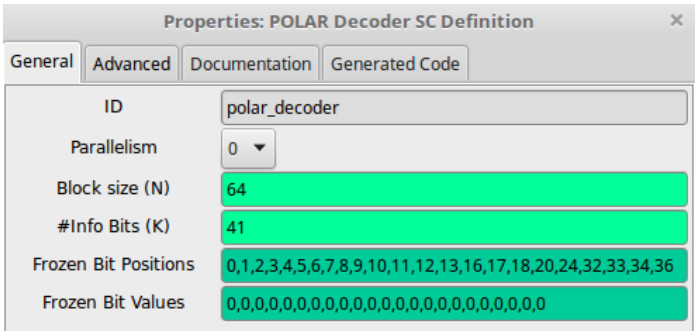


Figure 12. A bar chart of Bhattacharyya parameters for [64,41] polar code.

From Table 5, Figures 11 and 12 we can see that the frozen bit indices are: 1,2,3,4,5,6,7,8,9,10,11,12,13,14,17,18,19,21,25,33,34,35,37. Therefore the encoder and decoder blocks definitions have to be changed accordingly:





T16 Measure BER for [64,41] code. As in **T12** measure BER and input measured calculated values in prepared Table 6.

E_b/N_0 [dB]	noise voltage [Vrms]	theoretical P_b uncoded system	measured BER coded system
0	0,707945784	0,078649632	6,88E-02
+1	0,630957344	0,056281969	1,59E-02
+2	0,562341325	0,037506135	3,00E-03
+3	0,501187234	0,022878411	3,43E-04
+4	0,446683592	0,01250082	4,18E-05
+5	0,398107171	0,005953868	1,93E-06
+6	0,354813389	0,002388291	

Table 6. P_b and BER for polar code [64,41,{0,1,2,3,4,5,6,7,8,9,10,11,12,13,16,17,18,20,24,32,33,34,36}] with $E_c/N_0 = 0\ [dB]$.

Comparing BP with [16,11] polar code we can see even more distinct channel polarization. This will reduce BER further. The code rate has changed to $R = K/N = 41/64 \cong 0.641$.

Combine all 3 BER measurements together with P_b for an uncoded system and plot waterfall plot, see Figure 13.

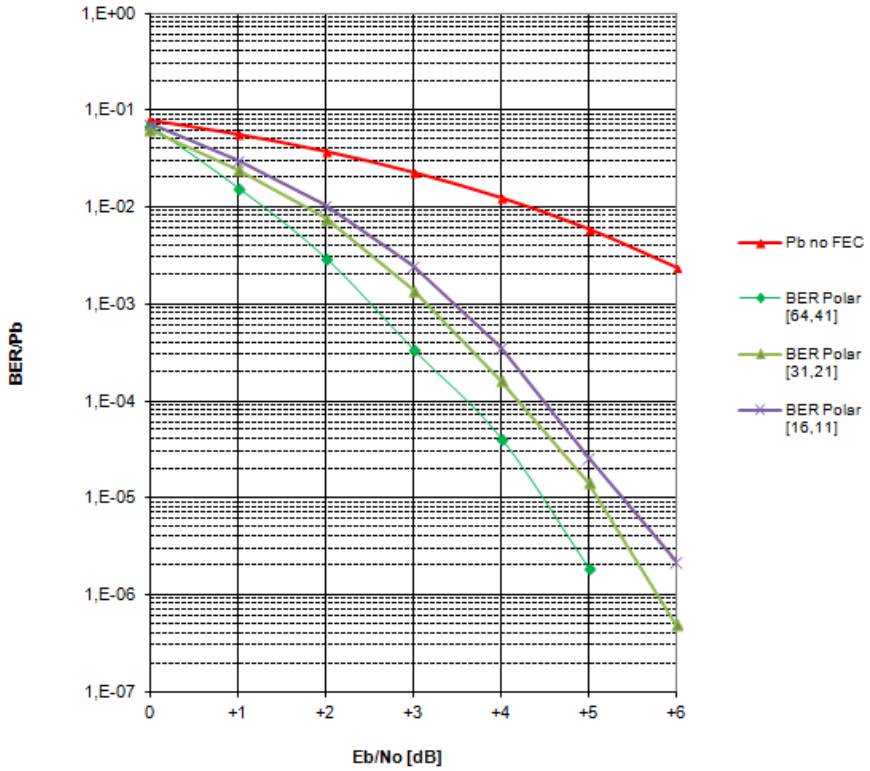


Figure 13. waterfall plot of BER and P_b.

From the displayed waterfall plot can be concluded that with increasing code length N the virtual channel polarization becomes more significant. What means that less channels are "in the middle" (of a bounded martingale - not shown). Virtual channels are becoming either "good" or "bad". This fact has a direct impact on the system BER.

ANSWERS TO DISCUSSION QUESTIONS:

Q1 Explain why with the growing code length the system BER improves.

Since more and more "good" channels (with capacity approaching to 1) are conveying the information and more and more "bad" channels (with capacity approaching to 0) convey no information (they are frozen) the decoder works with more reliable input information and it is able to decode input vector with less errors.

Q2 What are the preliminary conditions placed on the input information source with regard to polar coding.

It is assumed that it is memory less source producing independent and identically distributed binary random variable.

Q3 What are the practical application constrains of polar codes?

They can be constructed for code length : $N = 2^n$, where n is an positive integer.

They have optimum performance for designed SNR.

Q4 Explain why polar codes are used in 5G signalling system.

They are efficient and have simple and thus fast decoding.

ECE 156 - Fundamentals of Cryptography and Computer Network Security

DPA Attack on AES Labs

Dr. Hayssam El-Razouk
Electrical and Computer Engineering Department
California State University, Fresno

Acknowledgement: this lab is based on the open-source tutorial material available through the following link:

https://rtfm.newae.com/Capture/ChipWhisperer-Pro/

2. Firmware Build Setup

Supported setups:

SCOPES:

- OPENADC
- CWNANO

PLATFORMS:

- CWLITEARM
- CWLITEXMEGA
- CWNANO

This tutorial will introduce you to the software side of ChipWhisperer, including the tutorials themselves. It will also show you how to perform different operations on data based on input from the ChipWhisperer software. This can be used for building your own system which you wish to 'break'. All the `%%bash` blocks can be run either in Jupyter or in your favourite command line environment (note that Jupyter resets your path between blocks).

If you haven't run through `!!Introduction_to_Jupyter!!.ipynb` do that now.

Assuming you've done that, we can get started on the tutorial.

```
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEARM'
```

What is SimpleSerial

SimpleSerial is the communications protocol used for almost all of the ChipWhisperer demo project. It's a very basic serial protocol which can be easily implemented on most systems. This system communicates using a standard asyncronous serial protocol, 38400 baud, 8-N-1.

All messages are sent in ASCII-text, and are normally terminated with a line-feed (`'\n'`). This allows you to interact with the simpleserial system over a standard terminal emulator.

x

> Sending a 'x' resets the buffers. This does not require a line-feed termination. It is suggested to always send a stream of x's to initialize the system in case the device was already in some other mode due to noise/corruption.

`k00112233445566778899AABBCCDDEEFF\n`

> Loads the encryption key `00112233445566778899AABBCCDDEEFF` into the system. If not called the system may use some default key.

`pAABBCCDDEEFF00112233445566778899\n`

> Encrypts the data `AABBCCDDEEFF00112233445566778899` with the key loaded with the 'k' command. The system will respond with a string starting with r, as shown next.

`rCBBD4A2B34F2571758FF6A797E09859D\n`

> This is the response from the system. If data has been encrypted with a 'p' for example, the system will respond with the 'r' sequence automatically. So sending the earlier example means the result of the encryption was `cbbd4a2b34f2571758ff6a797e09859d`.

Building the Basic Example

To bulid the basic example, you'll need an appropriate compiler for your target. For the ChipWhisperer Lite/Xmega platform, you'll need `avr-gcc` and `avr-libc`, while if you're using an ARM target (like the ChipWhisperer Lite/STM32 platform), your need the GNU Toolchain for ARM devices. If you're using a target with a different architecture, you'll need to install the relevant compiler. If you're unsure, you can run the block below. If you've got the right stuff installed, you should see some version and copyright info printed for the relevant compiler:

```
%%bash
#check for avr-gcc
avr-gcc --version

#check for ARM gcc
arm-none-eabi-gcc --version
```

Now that you have the relevant toolchain installed, you should be able to build firmware for your desired platform. We'll begin by creating a new project based on simpleserial-base by making a new firmware and copying the files from the project we want to work on:

```
%%bash
cd ../../hardware/victims/firmware/
mkdir -p simpleserial-base-lab1 && cp -r simpleserial-base/* $_
cd simpleserial-base-lab1
```

Next we'll build the firmware. You'll need to specify the `PLATFORM` and `CRYPTO_TARGET` for your target. To save you from having to re-enter this info in every make block, you can edit the python below with your platform and crypto_target.

Common platforms are CWLITEXMEGA and CWLITEARM. To see a list of platforms leave `PLATFORM` as is.

This tutorial doesn't use any crypto, so we can leave `CRYPTO_TARGET` as `NONE`.

```
CRYPTO_TARGET = "NONE"
```

Provided you completed the fields above, you should be able to successfully run the block below.

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../../hardware/victims/firmware/simpleserial-base-lab1
make PLATFORM=$1 CRYPTO_TARGET=$2
```

Modifying the Basic Example

At this point we want to modify the system to perform 'something' with the data, such that we can confirm the system is working. To do so, open the file `simpleserial-base.c` in the simpleserial-base-lab1 folder with a code editor such as Programmer's Notepad (which ships with WinAVR).

Find the following code block towards the end of the file:

```
/*********************************
 * Start user-specific code here. */
trigger_high();

//16 hex bytes held in 'pt' were sent
//from the computer. Store your response
//back into 'pt', which will send 16 bytes
//back to computer. Can ignore of course if
//not needed

trigger_low();
/* End user-specific code here. *
```

Now modify it to increment the value of each data byte:

```
/*********************************
 * Start user-specific code here. */
trigger_high();

//16 hex bytes held in 'pt' were sent
//from the computer. Store your response
//back into 'pt', which will send 16 bytes
//back to computer. Can ignore of course if
//not needed

for(int i = 0; i < 16; i++){
    pt[i]++;
}

trigger_low();
/* End user-specific code here. *
 *********************************/
```

Then rebuild the file with `make` :

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../../hardware/victims/firmware/simpleserial-base-lab1
make PLATFORM=$1 CRYPTO_TARGET=$2
```

Python Script

We'll end by uploading the firmware onto the target and communicating with it via a python script. Depending on your target, uploading firmware will be different. For the XMega and STM32 targets, you can use ChipWhisperer's interface. Otherwise, you'll likely need to use and external programmer. If you have a CW1173/Xmega board, you can run the following blocks without modification. After running the final block, you should see two sets of hexadecimal numbers, with the second having values one higher than the first.

We'll begin by importing the ChipWhisperer module. This will allow us to connect to and communicate with the ChipWhisperer hardware. The ChipWhisperer module also includes analysis software, which we'll be looking at in later tutorials.

```
import chipwhisperer as cw
```

Documentation is available on ReadtheDocs or by calling `help()` on the module, submodule, function, etc.:

```
help(cw)
```

Next we'll need to connect to the scope end of the hardware. Starting with ChipWhisperer 5.1, `cw.scope` will attempt to autodetect which scope type you have (though if you have multiple ChipWhisperers connected, you'll need to specify the serial number). If you'd like, you can still specify the scope type.

```
scope = cw.scope()
```

```
help(scope)
```

We'll also need to setup the interface to the target (typically what we want to attack). Like with scopes, there's a few different interfaces we can use, which are available through `scope.targets.<target_type>`. The default, SimpleSerial, communicates over UART and is almost always the correct choice.

```
target = cw.target(scope, cw.targets.SimpleSerial)
```

Next, we'll do some basic setup. Most of these settings don't matter for now, but take note of the `scope.clock` and `scope.io`, which setup the clock and serial io lines, which needs to be done before programming the target.

Some targets require settings different than what's below. Check the relevant wiki article for your target for more information

```
# setup scope parameters
if SCOPETYPE == "OPENADC":
```

```
    scope.gain.db = 45
    scope.adc.samples = 3000
    scope.adc.offset = 1250
    scope.adc.basic_mode = "rising_edge"
    scope.clock.clkgen_freq = 7370000
    scope.clock.adc_src = "clkgen_x4"
    scope.trigger.triggers = "tio4"
    scope.io.tio1 = "serial_rx"
    scope.io.tio2 = "serial_tx"
    scope.io.hs2 = "clkgen"
elif SCOPETYPE == "CWNANO":
    scope.io.clkout = 7370000
    scope.adc.clk_freq = 7370000
    scope.io.tio1 = "serial_rx"
    scope.io.tio2 = "serial_tx"
```

Or, more simply:

```
scope.default_setup()
```

Now that the clock and IO lines are setup, we can program the target. ChipWhisperer includes a generic programming function, `cw.program_target(scope, type, fw_path)`. Here `type` is one of the programmers available in the cw.programmers submodule (`help(cw.programmers)` for more information). `fw_path` is the path to the hex file that you want to flash onto the device.

The final part of the binary path should match your platform (`<path>/simpleserial-base-CWLITEARM.hex` for CWLITEARM)

```
if "STM" in PLATFORM or PLATFORM == "CWLITEARM" or PLATFORM == "CWNANO":
    prog = cw.programmers.STM32FProgrammer
elif PLATFORM == "CW303" or PLATFORM == "CWLITEXMEGA":
    prog = cw.programmers.XMEGAProgrammer
else:
    prog = None

fw_path = '../../hardware/victims/firmware/simpleserial-base-lab1/simpleserial-base-{}.
```

And finally actually programming the device:

```
cw.program_target(scope, prog, fw_path)
```

Finally, we'll load some text, send it to the target, and read it back. We also capture a trace here, but don't do anything with it yet (that will come in later tutorials). You should see your original text with the received text below it.

```
ktp = cw.ktp.Basic() # object to generate fixed/random key and text (default fixed key,
key, text = ktp.next()  # get our fixed key and random text

target.simpleserial_write('k', key)
target.simpleserial_wait_ack()
scope.arm()

target.simpleserial_write('p', text)
```

```
ret = scope.capture()
trace = scope.get_last_trace()
output = target.simpleserial_read('r', 16)

from binascii import hexlify
print(hexlify(output))
print(hexlify(text))
```

You can also just run:

```
ret = cw.capture_trace(scope, target, text, key)
if ret:
    trace = ret
    print(hexlify(ret.textout))
    print(hexlify(text))
```

Now that we're done with this tutorial, we'll need to disconnect from the ChipWhisperer. This will prevent this session from interfering from later ones (most notably with a `USB can't claim interface` error). Don't worry if you forget, unplugging and replugging the ChipWhipserer should fix it.

```
scope.dis()
target.dis()
```

FRESNO STATE
Electrical and Computer Engineering

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

COMMUNICATION ENGINEERING LABORATORY

LABORATORY INSTRUCTION SHEET

Course Code	ECE 135L/ECE 235L
Course Name	Software Defined Radio Communions Lab
Experiment Code	4
Experiment Title	FM reception using SDR

EXPERIMENT 4: FM reception using SDR

AIM: To gain insight into principles of FM reception using SDR and getting familiar with GNU Radio concept.

MOTIVATION:

The concept of GNU Radio offers very efficient, flexible and powerful way of realization of modern communication systems. Getting familiar with basics of GNU Radio concept will be demonstrated on demodulation of FM broadcasting. The following experiment describes detection of FM radio broadcasting with the focus on the utilization of TIMS SDR module. SDR concept offers a very flexible way how to realize various parts of communication systems or communication systems as a whole.

FM radio broadcasting is still widely used today even in the presence of DAB transmission. Throughout the world, the FM broadcast band falls within the VHF part of the radio spectrum spanning usually from 87.5 to 108.0 MHz. The assigned nominal centre frequency of an FM broadcast station is usually an exact multiple of 100 kHz. Standard FM broadcasting uses a 200 kHz bandwidth for each station, see Figure 1.

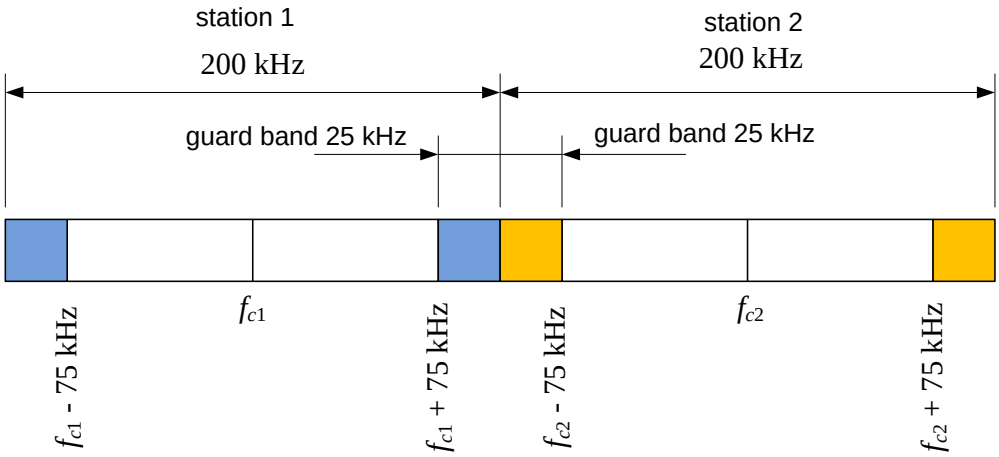


Figure 1. Commercial FM bandwidth allocation for two adjacent stations.

With FM, frequency deviation from the assigned centre frequency at any instant is directly proportional to the amplitude of the input signal, determining the instantaneous frequency of the transmitted signal. The maximum frequency deviation of the centre frequency is usually specified and regulated by the licensing authorities in each country. For FM stereo broadcasting the maximum permitted centre frequency deviation is set to ±75 kHz. Adjacent stations are separated with 25 kHz guard bands at the upper and lower end of 200 kHz bandwidth. The instantaneous frequency deviation of the transmitter centre frequency due to the stereo audio signal and pilot tone (at 10% modulation) is:

$$f(t)=\left[0.9\times\left(\frac{A+B}{2}+\frac{A-B}{2}\times\sin\left(4\pi f_p t\right)\right)+0.1\times\sin\left(2\pi f_p t\right)\right]\times 75kHz \quad (1)$$

where A and B are the pre-emphasized left and right audio signals and f_p = 19 kHz is the frequency of the pilot tone. Typical spectrum of composite baseband FM signal is shown in Figure 2.

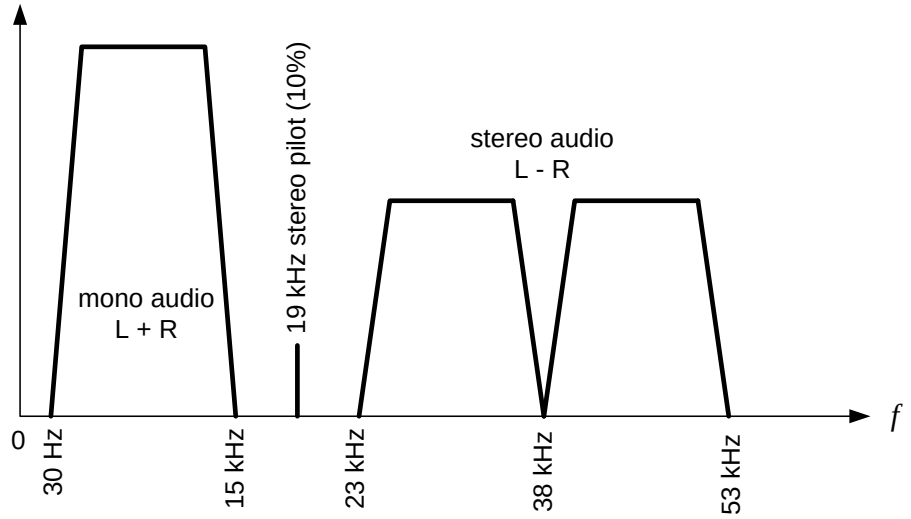


Figure 2. Typical spectrum of composite baseband FM signal.

In the next we will assume the reception of monophonic FM transmission of L+R signals with block diagram shown in Figure 3.

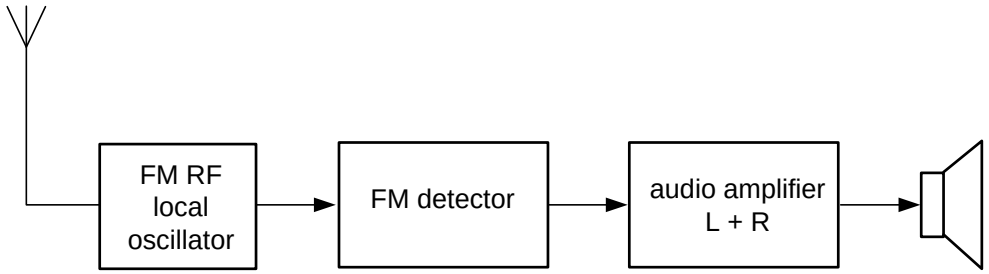


Figure 3. Block diagram of monophonic FM receiver.

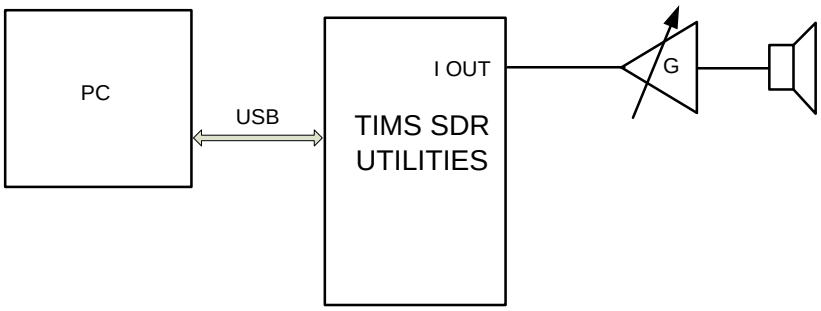
OBJECTIVES:

1. To understand the principles of GNU Radio concept.
2. To obtain hands-on experience with GNU Radio.
3. To obtain hands-on experience with FM reception using GNU Radio and SDR.

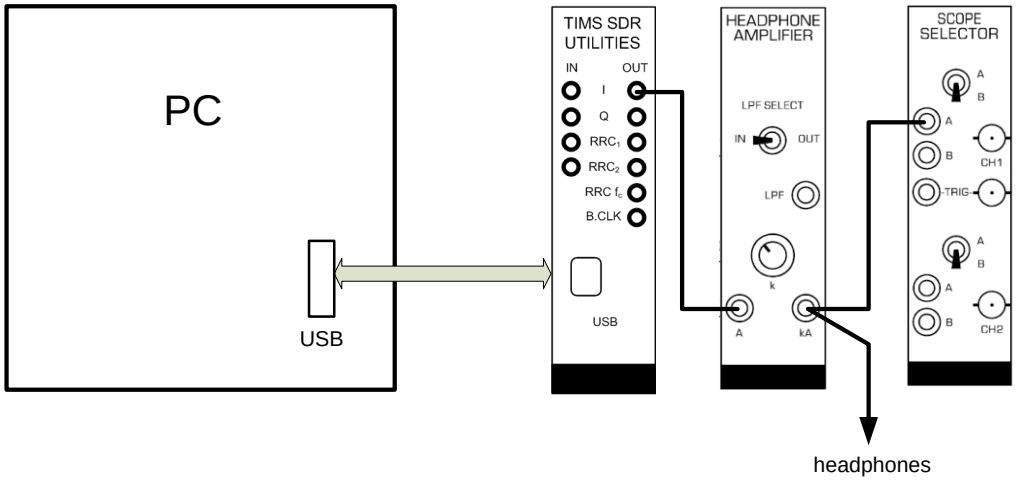
EQUIPMENT LIST:

1. TIMS SDR UTILITIES
2. HEADPHONE AMPLIFIER

BLOCK DIAGRAM:



WIRING DIAGRAM:



MODULE SETTINGS:

1. HEADPHONE AMPLIFIER:
   LPF SELECT: OUT

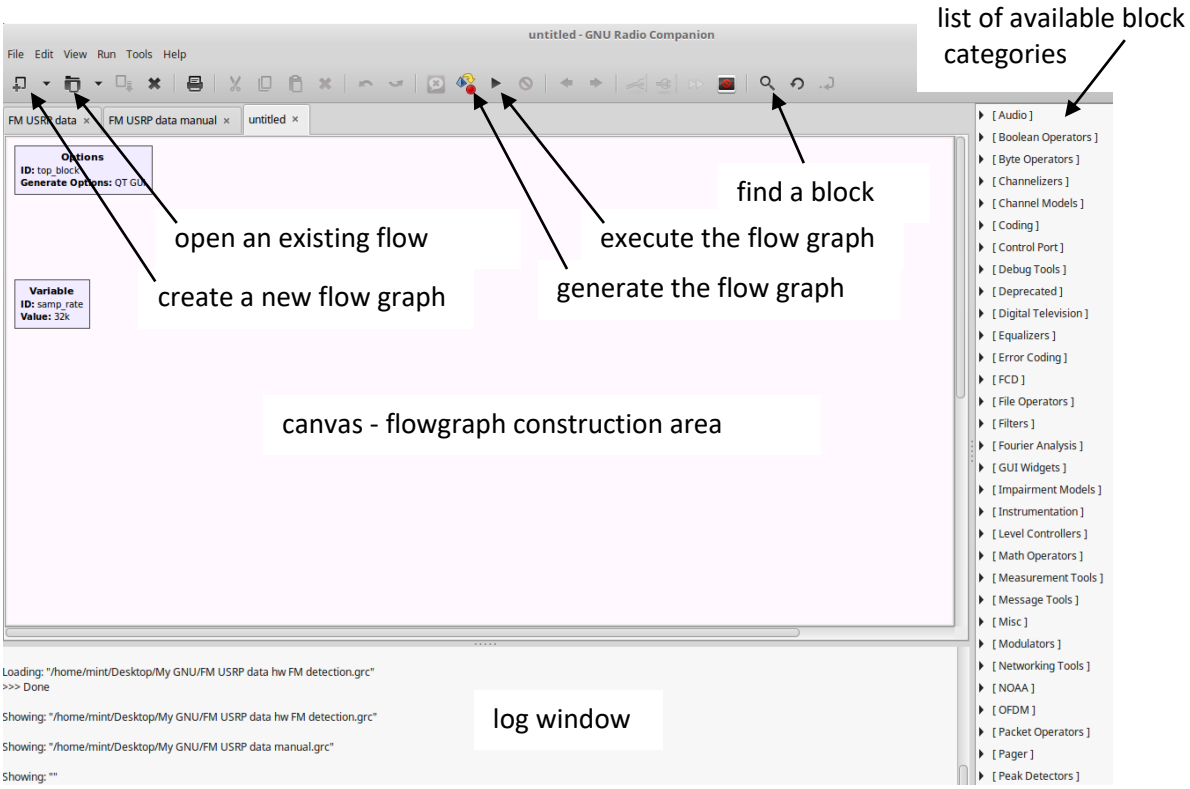
A BRIEF DESCRIPTION OF GNU RADIO COMPANION:

GNU Radio is a free, open source software development toolkit that provides signal processing blocks to implement SDR. GNU Radio applications are primarily written using Python programming language, while the supplied, performance-critical signal processing path is implemented in C++ . To simplify interaction with GNU Radio a graphical user interface - GNU Radio Companion (GRC) was created. All necessary information regarding GNU Radio can be found on: https://www.gnuradio.org/ .

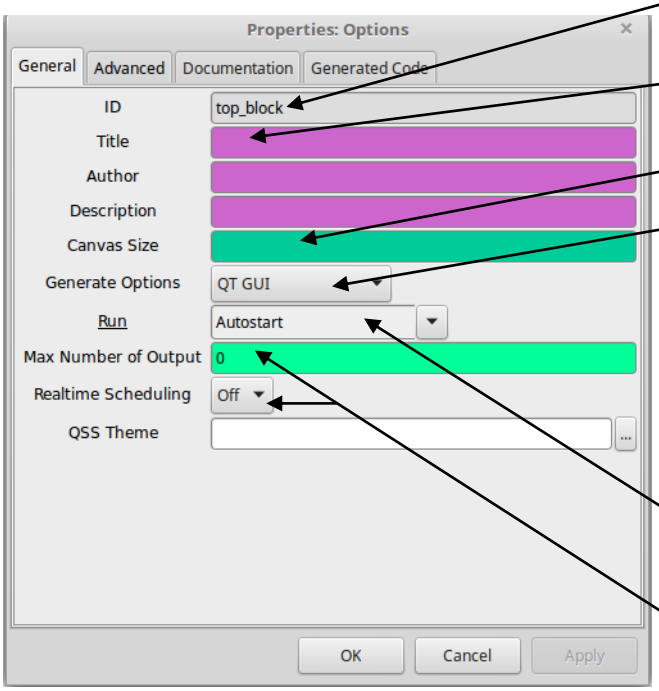
A brief description of GNU Radio concept:

- All signal processing in GNU Radio is done through flow graphs.
- A flow graph consists of blocks. A block does one signal processing operation, such as filtering, adding signals, transforming, decoding, hardware access, etc.
- Data passes between block in various formats (complex, real, integers,...).
- Every flow graph requires at least one sink and one source.

After starting GRC a new untitled GRC window will appear:



On new empty canvas two default blocks will appear: **Options** and **Variable**. Double click on block **Options**. This block sets the global parameters:



name of generated Python file

title of the main GUI window, or name of **Hier**archical block

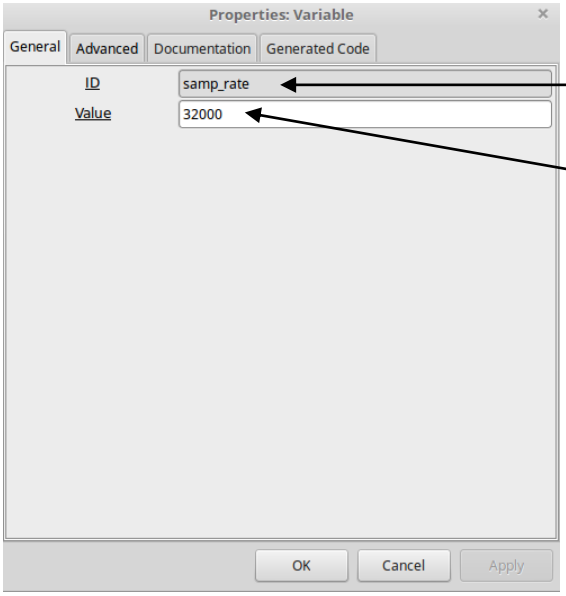
canvas size can be input here, e.g. 1280, 1024

type of code to be generated:
WX GUI - GUI app using WX toolkit (use WX GUI blocks)
QT GUI - GUI app using Qt toolkit (use Qt GUI blocks) - recommended to use
No GUI - command line app, text based (run in a - console) without GUI
Hier - creates a hierarchical block that will appear in the block list

sets how to start and stop the flow graph:
Autostart - automatically start flow graph
Off - do not automatically start flow graph

default values - do not have to be altered (advanced)

Each change of block parameter has to be confirme . . . ns! Double click on block **Variable**. This block sets the global sampling rate:



ID: variable name (Python). It is possible to refer to it in another blocks.

Value: an arbitrary Python expression, e.g.
integer: 32000 (default)
floating point number: 32e6
integer cast of floating point number: int(32e6)

TIMS system does not enable to directly process signals from broadcast FM spectrum, therefore a selected portion of FM spectrum was scanned using USRP and saved to file: fmdata.wav . Download this file from EMONA TIMS webpage and save it to suitable directory, e.g. where created GRC files will be saved. This file contains 500kHz wide scanned FM spectrum with centre frequency: $f_{centre} = 105.55\ MHz$ which is down converted to baseband.

FM reception will be demonstrated on two examples. The first receiver will perform individual demodulation steps and the second one will used already made block which performs the complete FM demodulation.

FM receiver with individual demodulation steps:

Block diagram of the FM receiver with individual demodulation steps is shown in Figure 4.

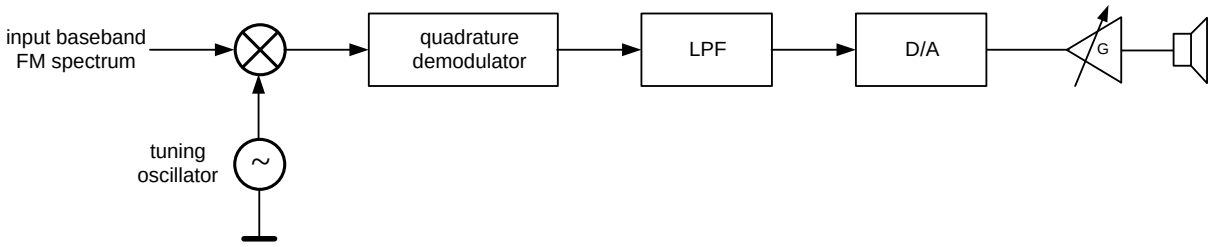
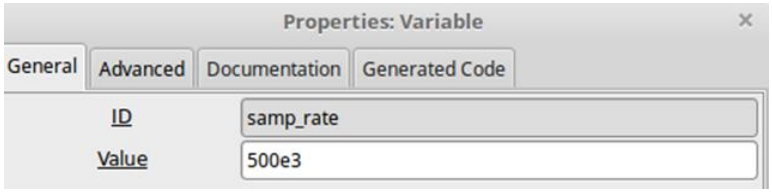


Figure 4. Block diagram of the FM receiver with individual demodulation steps.

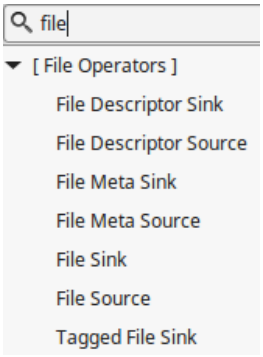
Tuning oscillator selects the desired FM station. Quadrature demodulator together with low pass filter demodulate selected FM station. Digital signal is converted to analog one using TIMS SDR UTILITIES module. Resulting audio signal is amplified using HEADPHONE AMPLIFIER module.

T1 Setting the global sampling rate. In the opened new untitled GRC window double click on the block **Variable** and set the value of the sampling rate to 500k to match the sampled source signal:

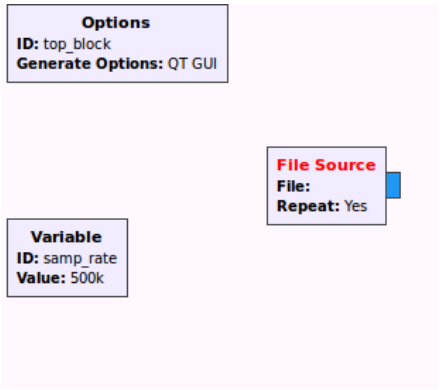


Save the file to the appropriate directory. Note that colour of the file name changes from red to black.

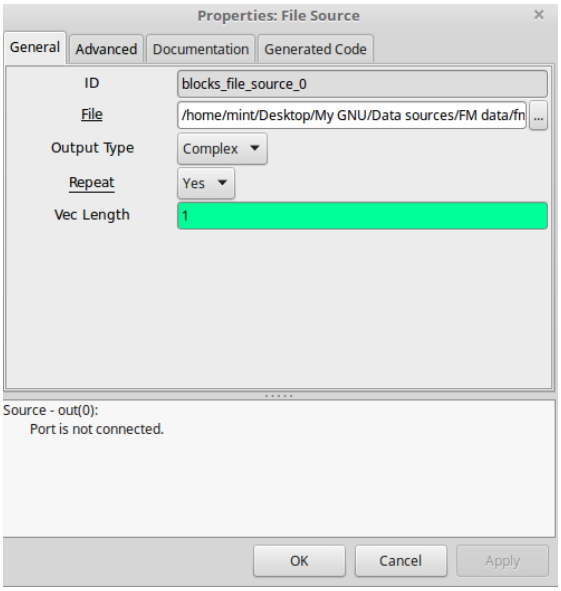
T2 Placing the file source with the recorded FM spectrum in the flow graph. Double click on the icon: placed in the GRC menu toolbar. This opens a dialog enabling to search for a block by its name. In the search command line type: `file`:



Double click on **File Source**. This action will place **File Source** block on the canvas:



File Source block will supply the receiver with sampled data of the received signal. It replaces antenna and the RF input circuits. Double click on the **File Source** block and set the following parameters:



ID: you can leave default setting

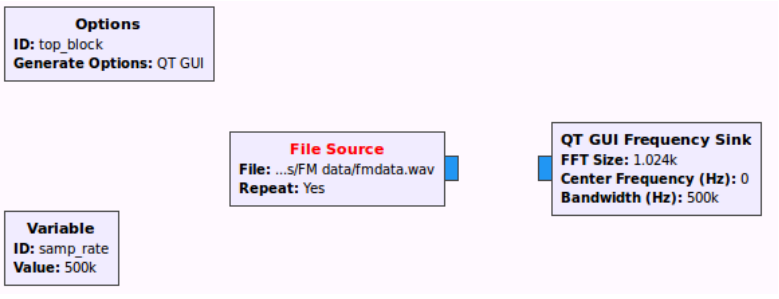
File: click on ellipsis (next to the File parameter) and locate the path where the fmdata.wav file is stored

Output Type: selects the type of the output data, leave `Complex`

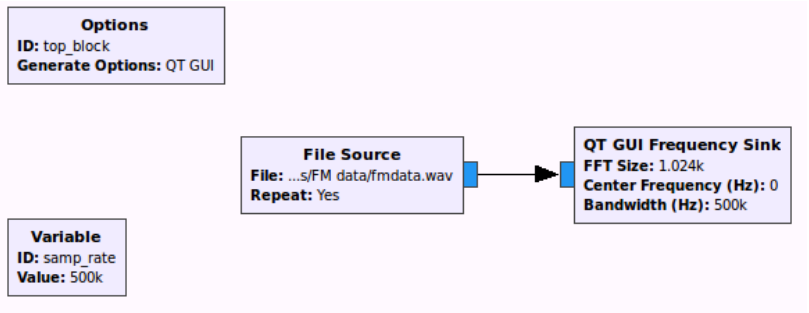
Repeat: `Yes` - this will cause the data to repeat so that you have a continuously playing signal

Note that at this point block's output port is not connected and the block name is red.

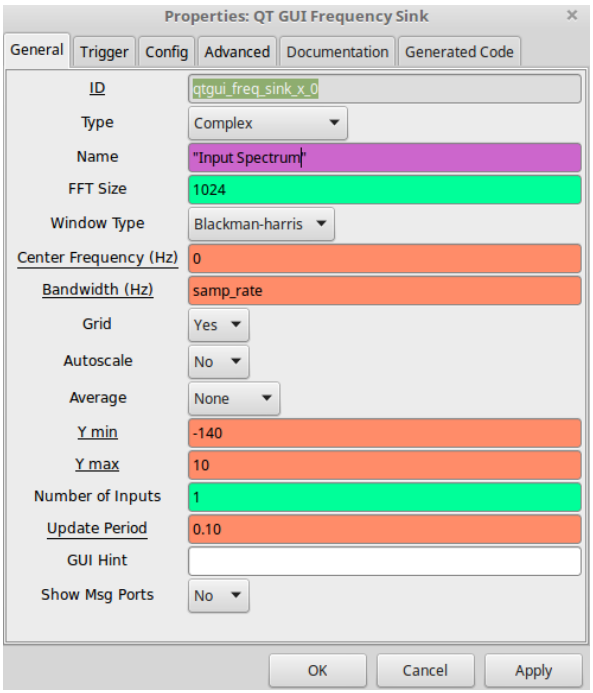
T3 Observing frequency spectrum of the input signal. In the block search dialog type: `qt` and place block **QT GUI Frequency Sink** in the canvas:



QT GUI Frequency Sink block acts as a spectrum analyzer. Connect **File Source** block with **QT GUI Frequency Sink** by clicking on the **File Source** block output port and then on **QT GUI Frequency Sink** block input port:



Valid connections are shown with black arrow. Note that the colour of the **File Source** block name has changed from red to black - meaning the correct connection and the correct setup of the block's parameters. In case that you want to delete connection, click on that connection - it changes colour to cyan and then press Delete. Double click on **QT GUI Frequency Sink** block:



ID: you can leave default setting

Type: leave `Complex` . It should match the type of the output port where it is connected to.

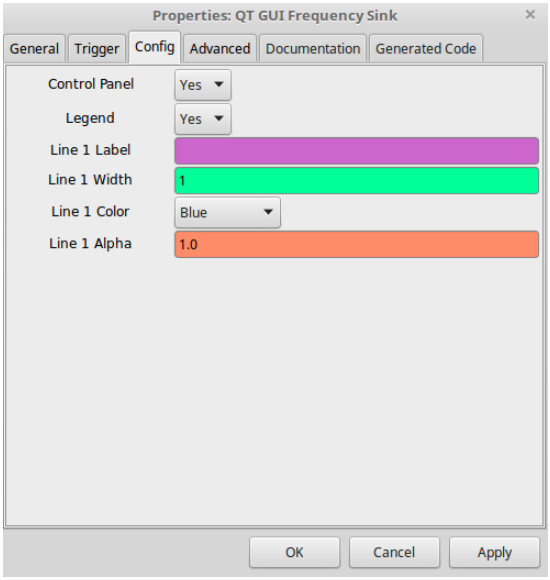
Name: you can input the name of the graph which will be shown in the **Top Block**

Bandwidth (Hz): selects the displayed bandwidth. Default is `samp_rate`. Usually other bandwidth values are derived from this value.

Grid: `Yes` - this places grid on the screen.

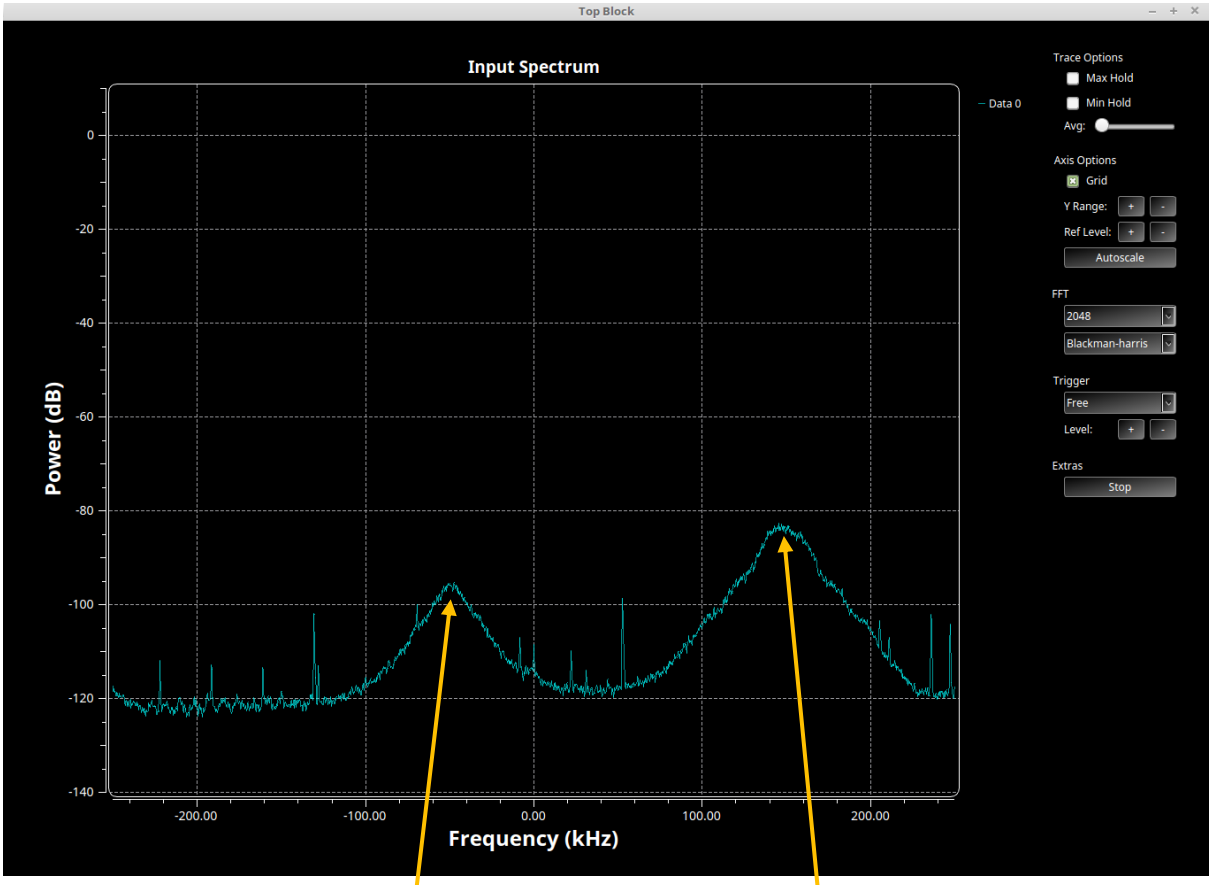
Y min, **Ymax**: are set limits in relative dB. To display signal levels in the actual dB, a calibration has to be performed.

In the **Properties** of **QT GUI Frequency Sink** window click on **Config**:



Control Panel: `Yes` - this activates various display options in the **Top Block**.

In the GRC menu toolbar click on the icon: generate the flow graph and then on the icon execute the flow graph. If all processing went well **Top Block** window should appear:

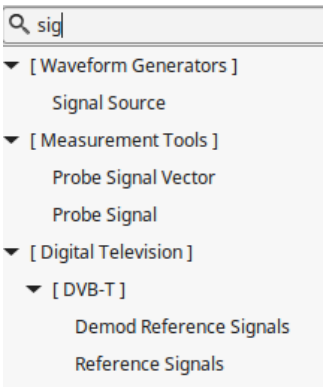


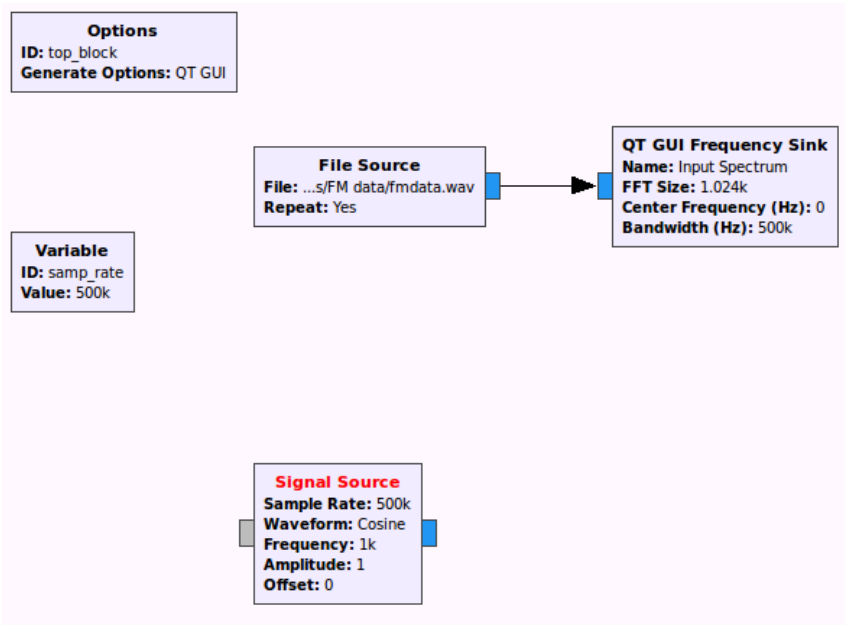
T4 Tuning the receiver [illegible] iency shifting of the scar [illegible] pectrum. In the frequency spectrum shown in T[illegible] FM stations 200kHz apart can be located. Place the cursor at some point of the displayed frequency spectrum and an exact frequency and signal level of the cursor position is shown. For an easier readout of the centre frequencies of the received FM stations set averaging to `max` (with the slider on the control

panel) and FFT size to: `2048`. As was mentioned before displayed frequency: 0.00 now corresponds to the down converted centre frequency of the USRP which was used to scan the live FM broadcasting. We know that centre frequencies of the FM stations are placed at the exact integer multiples of 100 kHz. With the cursor it is possible to identify that FM station denoted as **FM1** is located at -50kHz from the centre frequency and station **FM2** is located at +150kHz from the centre frequency. It is also possible to check that those two stations are exactly 200kHz apart.

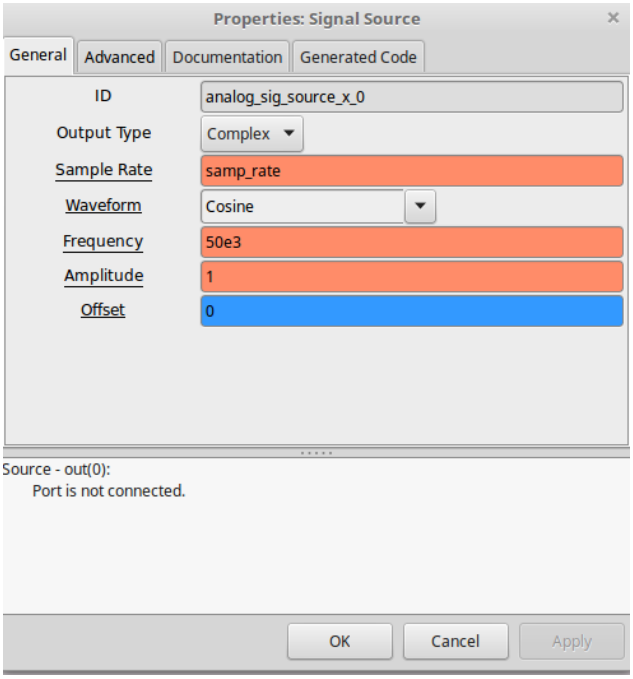
In the next we will demodulate signal of station **FM1**. To tune the receiver to this station it is necessary to shift its centre frequency by 50kHz to the right (using frequency shifting property of FT). This can be achieved by multiplying the received spectrum by 50kHz harmonic signal in time domain.

In the block search dialog type: `sig` and place block **Signal Source** in the canvas:

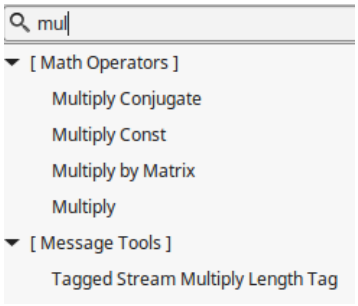


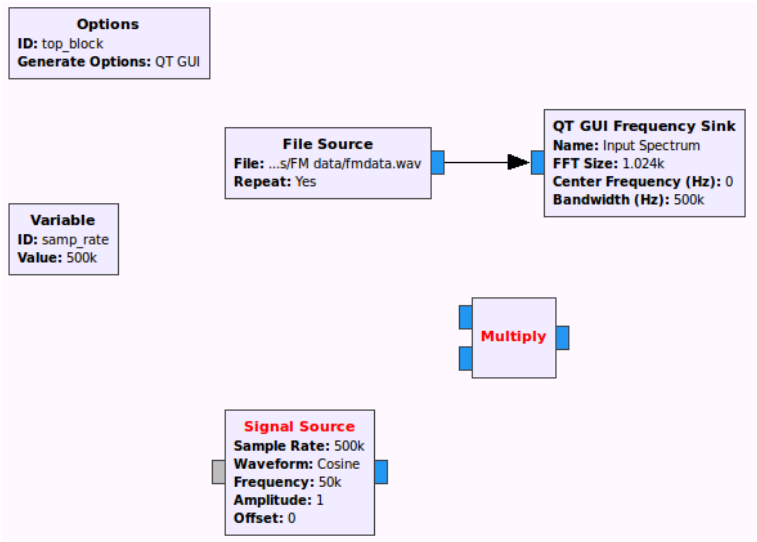


Set the frequency of **Signal Source** block to 50kHz:

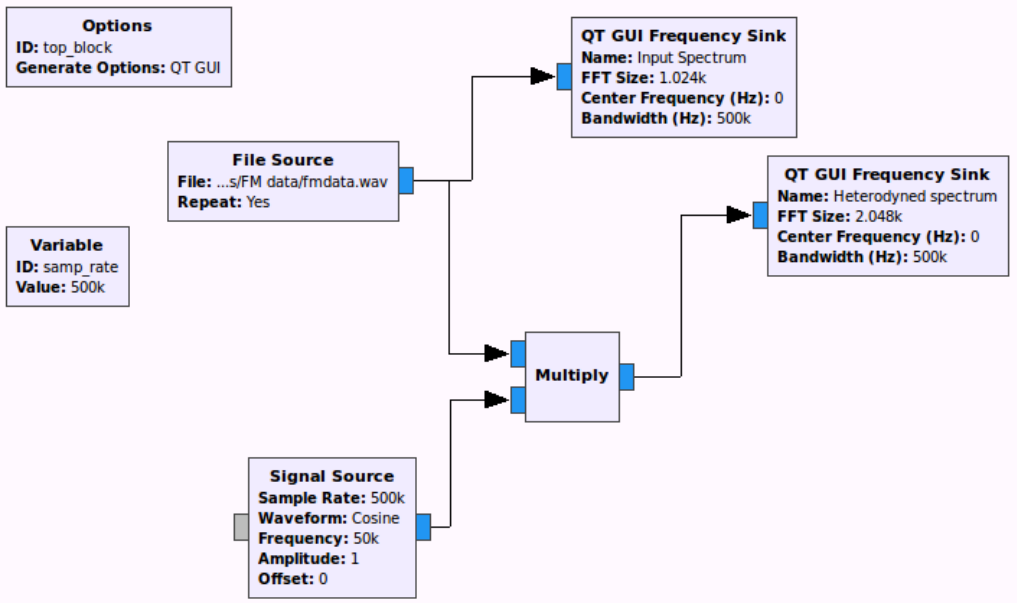

Properties: Signal Source
General
Advanced
Documentation
Generated Code
ID
analog_sig_source_x_0
Output Type
Complex
Sample Rate
samp_rate
Waveform
Cosine
Frequency
50e3
Amplitude
1
Offset
0
Source - out(0):
Port is not connected.
OK
Cancel
Apply

Multiplication of the received signal with tuning oscillator signal (50kHz) is performed by multiplier. In the block search dialog type: `mul` and place block **Multiply** in the canvas:

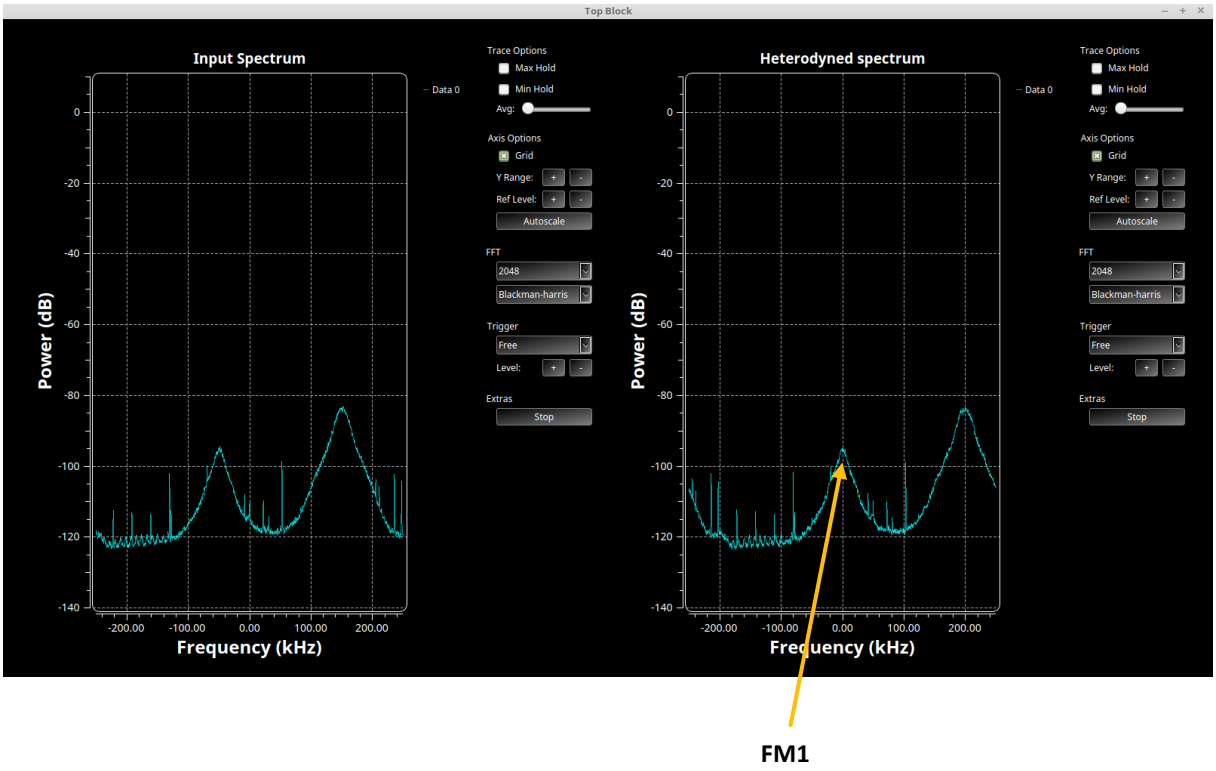




Connect one input of the **Multiply** block to the output of the **File Source** block and the other one to the output of the **Signal Source** block. To observe the frequency shifted spectrum place another **QT GUI Frequency Sink** block in the canvas and connect it to the output of the **Multiply** block. This can be performed easily by clicking on **QT GUI Frequency Sink** block named: "Input Spectrum" copying it (CTRL C) and pasting it (CTRL V). The second **QT GUI Frequency Sink** block can be named e.g. "Heterodyned Spectrum".



Save the modified GRC file and execute generated flow graph:



We can see that centre frequency of **FM1** station now is shifted to 0.00 Hz.

In case of placing more than one graphical element (slider, chooser, time or frequency sink etc.) in the flow graph when the **GUI Hint** entry (in the block properties window) is left blank, then they are stacked vertically on top of each other. To place them in the landscape arrangement, individual positions of graphical elements can be specified by setting their grid positioning arguments in the **GUI Hint**

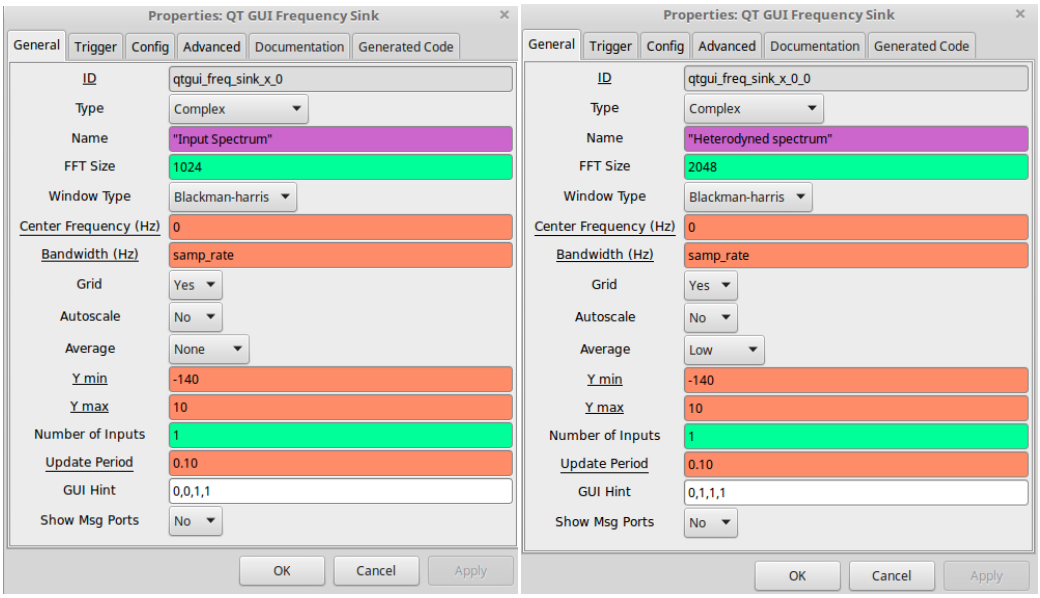
entry of the individual blocks. A grid positioning argument is a list of four integers in the form:

`row, column, row span, column span`

If this argument is filled in, graphical elements are placed in the specified row `row` and the specified column `column`, spanning `row span` rows and `column span` columns. Note that `row span` and `column span` have to be ≥ 1. Example of the graphical elements placement together with **GUI Hint** entries:

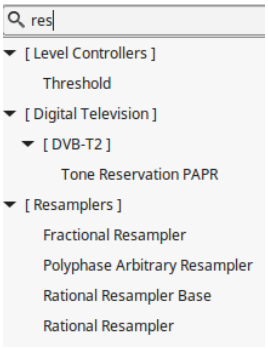
signal selector (0,0,2,1)	slider # 1 (0,1,1,1)
	slider # 2 (1,1,1,1)
frequency display (2,0,1,1)	time display (2,1,1,1)

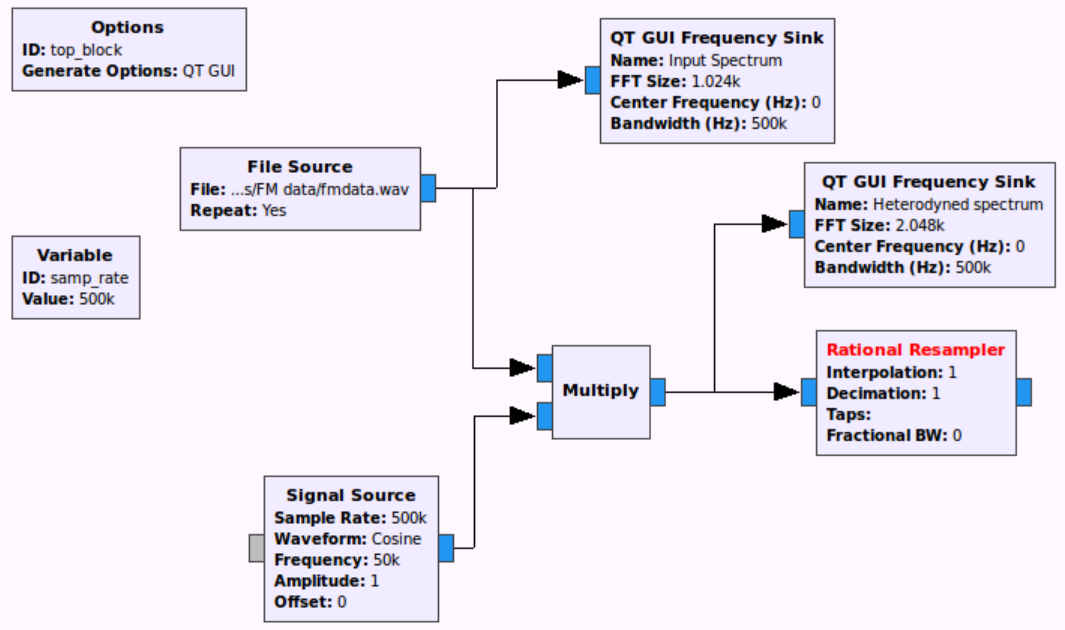
Displayed **Top Block** with two graphical elements was obtained with the following setup:



T5 Demodulation of baseband FM spectrum using quadrature demodulator. The output of quadrature demodulator is fed to low pass filter to remove high frequency components. Low pass filtered signal is then D/A converted using TIMS SDR UTILITIES module. This module has 48ks/s input sampling rate. Denote this rate as: $audio_rate = 48 \times 10^3$. **Low Pass Filter** block has to have sampling rate higher than used D/A convertor. Let set low pass filter sampling rate to be: $4 \times audio_rate = 4 \times 48 \times 10^3 = 192 \times 10^3$. Denote this new variable as: $audio_interp = 4$. The global sampling rate was set to 500ks/s. For further processing (signal demodulation) this sampling rate has to be lowered to 192ks/s. Block which performs this rate matching is called resampler.

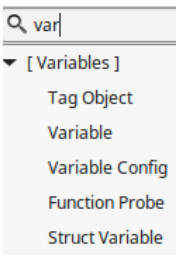
In the block search dialog type: `res` and place block **Rational Resampler** in the canvas and connect its input with **Multiply** block output:



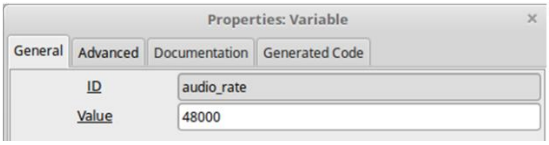


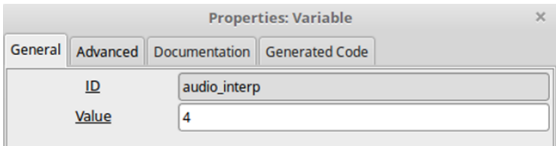
The output sampling rate of **Rational Resampler** block is: $input_rate \times \frac{interpolation}{decimation}$.

As was mentioned before we want the low pass filter sampling rate to be 192ks/s. To set the *interpolation* and *decimation* values we will place two variables in the flow graph: *audio_rate* and *audio_interp*. In the block search dialog type: `var` and place two blocks **Variable** in the canvas:

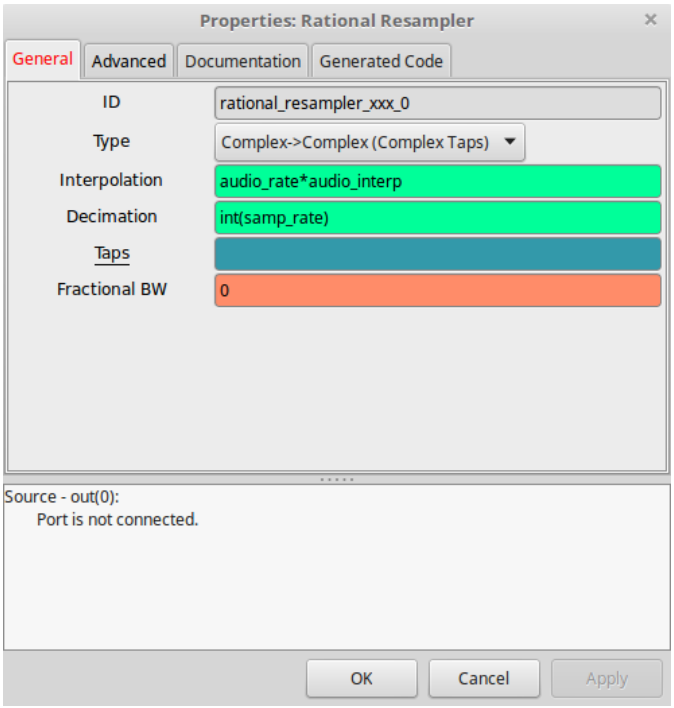


Name one variable as *audio_rate* and the other one as *audio_interp* and set its parameters as was previously discussed:

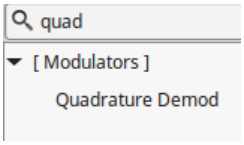




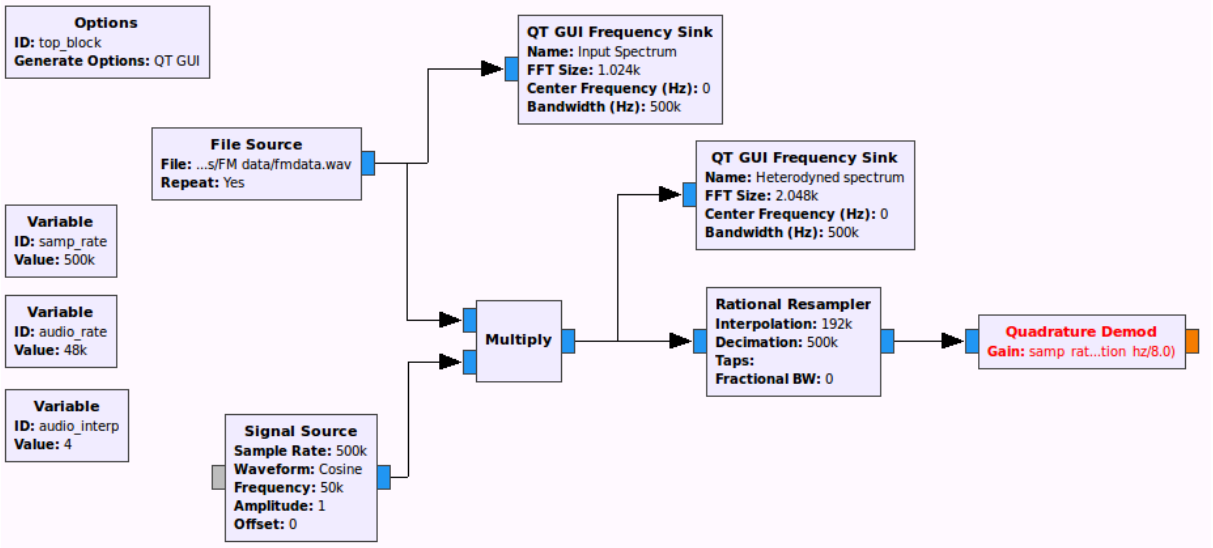
Using those two variables set the properties of **Rational Resampler** block:



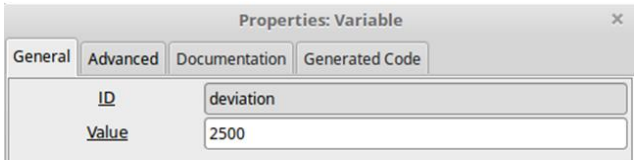
Demodulation of FM signal will be performed by quadrature demodulator. In the block search dialog type: `quad` and place block **Quadrature Demod** in the canvas:



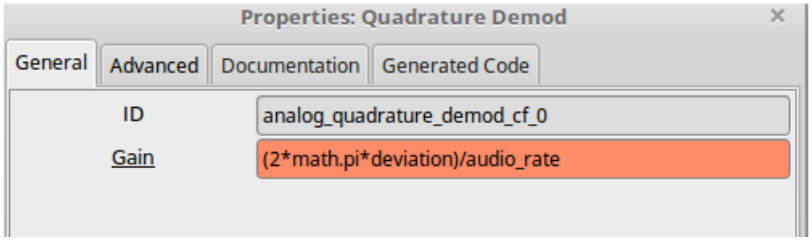
Connect the output of **Rational Resampler** block to the input of **Quadrature Demod** block:



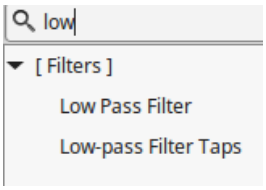
Create a new variable for setting the maximal frequency deviation and denote it: *deviation*. With respect to the other hardware set its value to:



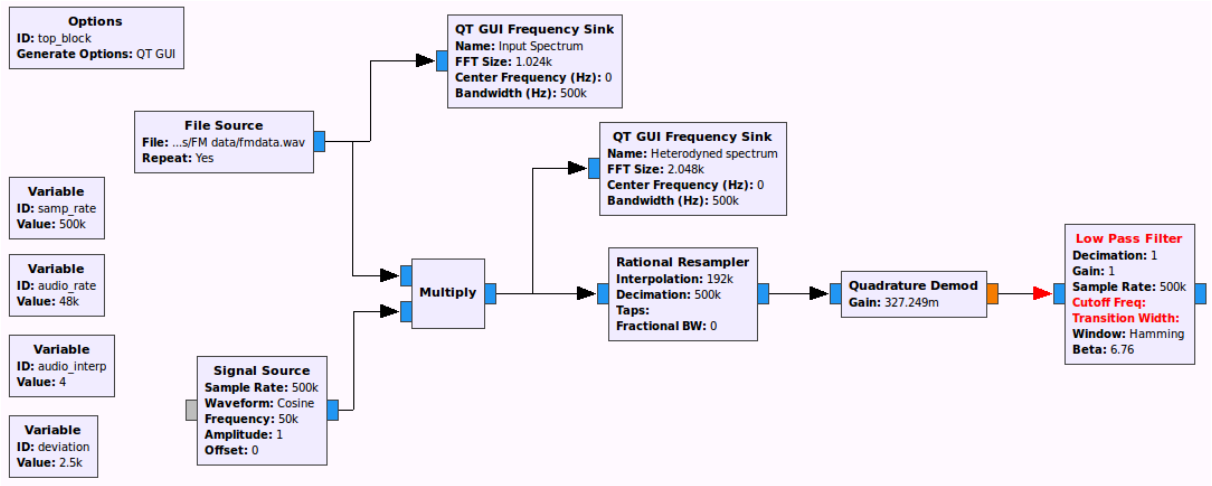
Set the gain of **Quadrature Demod** block to:



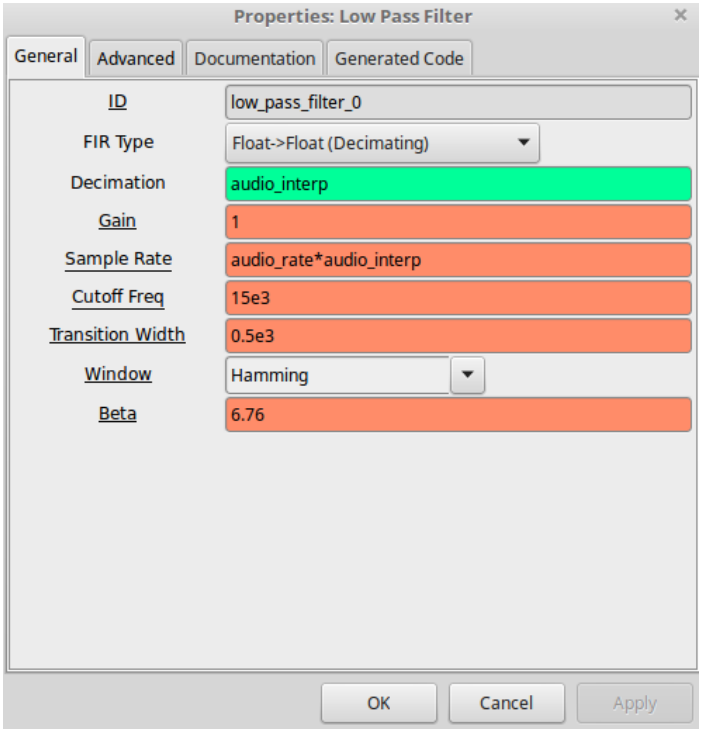
In the block search dialog type: `low` and place block **Low Pass Filter** in the canvas:



Connect the output of **Quadrature Demod** block to the input of **Low Pass Filter** block:

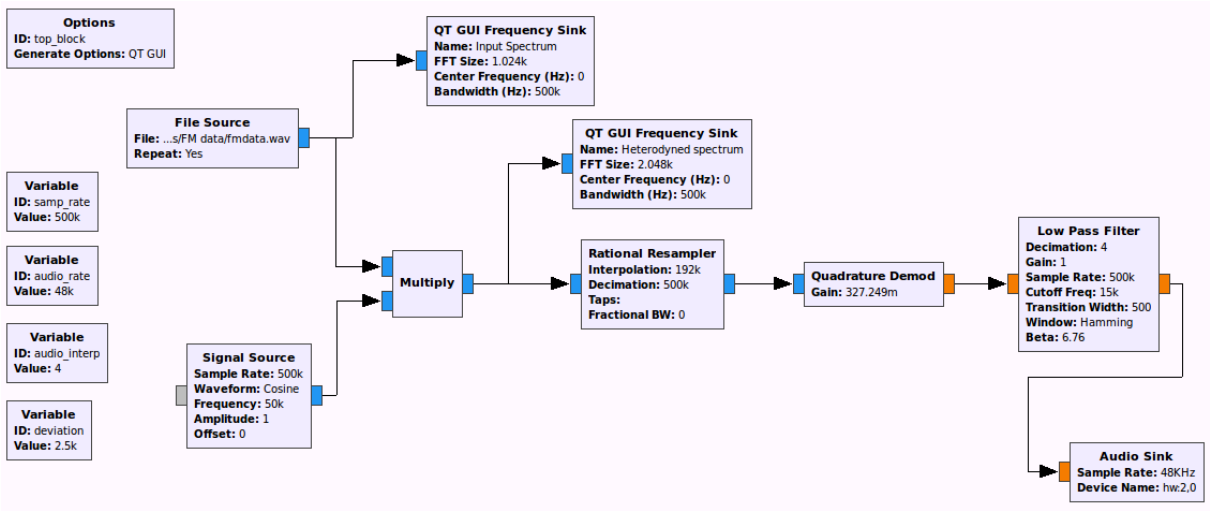


Set the filter parameters:



FM demodulated and low pass filtered signal in digital form is converted to analog signal using D/A convertor from TIMS SDR UTILITIES module. The signal output from the GRC software is realized with **Audio Sink** block. In the block search dialog type: `aud` and place block **Audio Sink** block in the canvas:

Connect the output of **Low Pass Filter** block to the input of **Audio Sink** block:



Set **Audio Sink** block sample rate to 48 kHz. To identify TIMS SDR UTILITIES module in PC hw, in command prompt type: `aplay -l` :

The TIMS SDR UTILITIES audio device is detected as card # 2, and the subdevice # 0. These values creates a unique **Device Name** of **Audio Sink** block. This device name is input in the format: `hw:card#,device#` . In this case it would be: `hw:2,0` . Save the flow graph.

T6 Plug headphones to the headphone plug in the HEADPHONE AMPLIFIER module. After generating and executing flow graph you should hear boradcasting of **FM1** station.

FM receiver with integrated FM demodulation block:

GRC software enables to integrate individual FM demodulation steps into one block and thus to simplify receiver realization. In contrast to FM receiver from Figure 4, quadrature demodulator and LPF are part of wideband FM receive block, as it is shown in Figure 5.

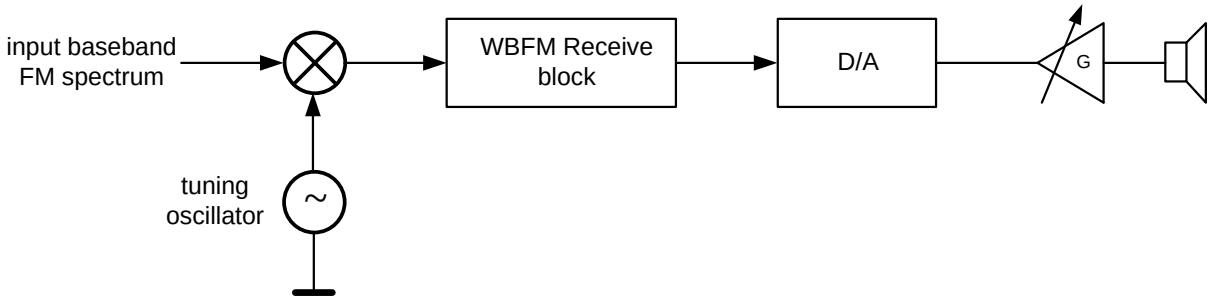
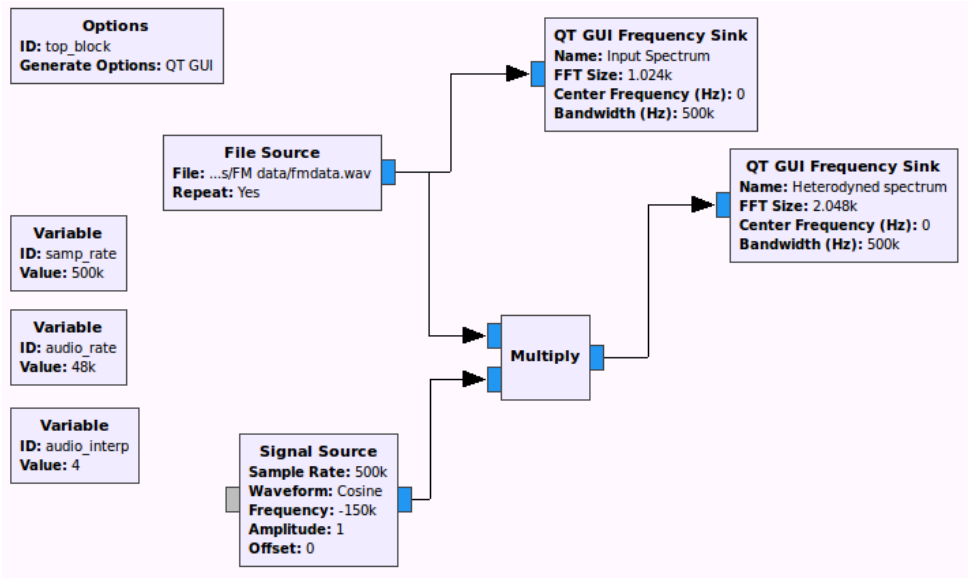
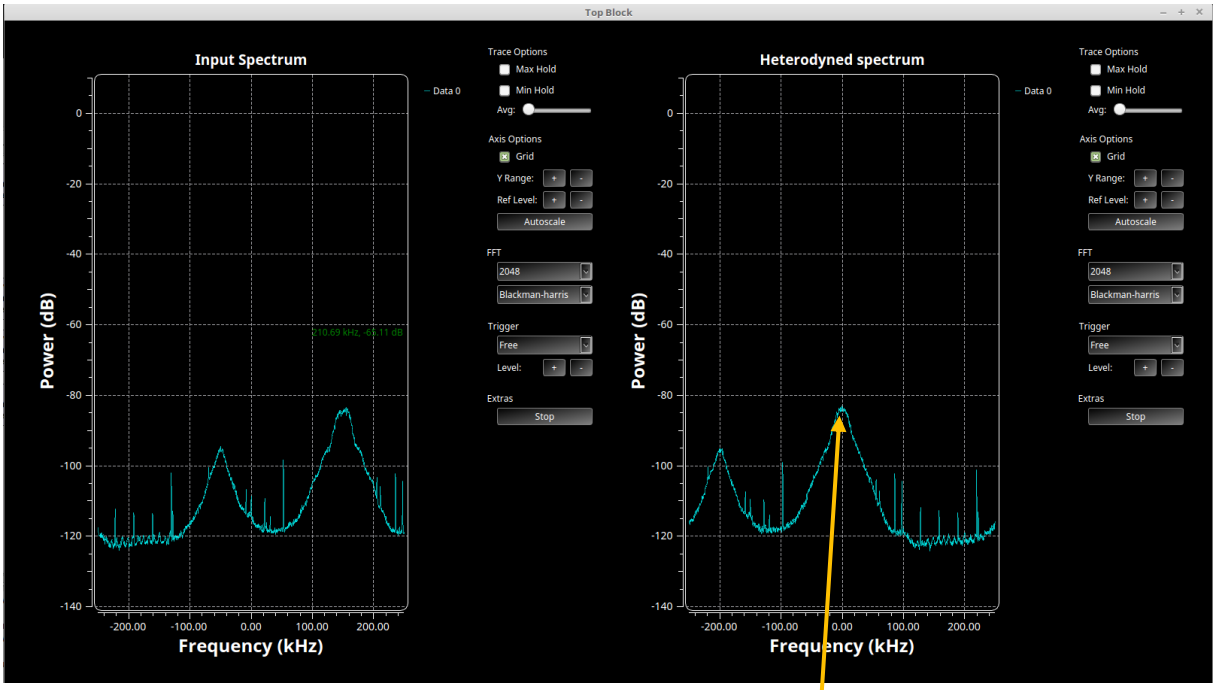


Figure 5. Block diagram of the FM receiver with integrated FM demodulation block.

T7 Tuning the receiver to **FM2** station. Now we will listen FM station denoted as **FM2**. As was mentioned in **T4**, station **FM2** is located +150 kHz from the centre frequency. Therefore in order to tune the receiver to this station the input signal has to be frequency shifted by -150 kHz. This is performed by changing the frequency of the tuning oscillator. The easiest way how to create a new flow graph for this receiver is to make a copy of the previous one and save it under different name. Then delete from it all blocks except: **Options**, **Variable** (samp_rate), **Variable** (audio_rate), **Variable** (audio_interp), **Signal Source**, **Multiply** and two **QT GUI Frequency Sink** blocks. Set the frequency of the tuning oscillator to -150 kHz:

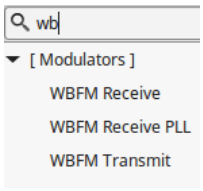


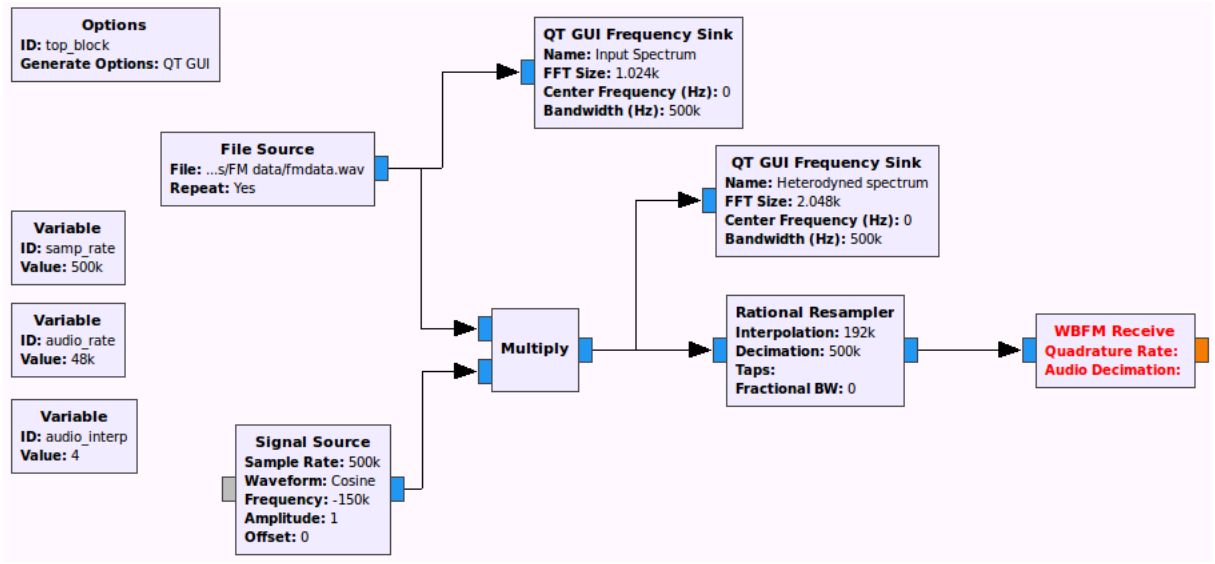
Save the flow graph and execute it:



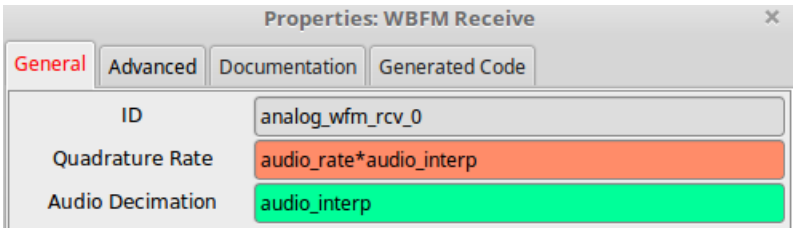
We can see that the centre frequency of the FM station denoted as **FM2** is now shifted to 0.00 Hz. Sampling rate has to be slowed down the same way as in the previous case so place the **Rational Resampler** block in the flow graph (copy it from the previous receiver with the same parameters). Connect the **Rational Resampler** block to the output of **Multiply** block.

T8 Demodulate baseband FM signal with **WBFM Receive** block. This block performs all necessary FM demodulation operations. In the block search dialog type: `wb` and place block **WBFM Receive** in the canvas and connect it to the output of the **Rational Resampler** block:

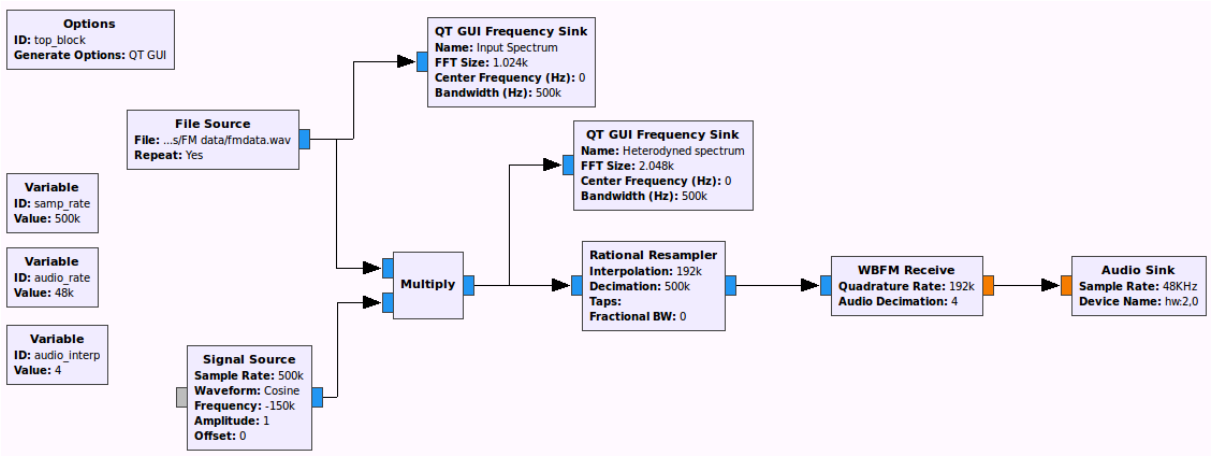




Set its parameters: **Quadrature Rate** and **Audio Decimation** to match **Rational Resampler** output and **Audio Sink** input:



Copy **Audio Sink** block from the previous receiver (with the same parameters) and connect it to the output of the **WBFM Receive** block:



T9 Plug headphones to the headphone plug in the HEADPHONE AMPLIFIER module. After generating and executing flow graph you should hear boradcasting of **FM2** station.

ANSWERS TO DISCUSSION QUESTIONS:

Q1 Explain the advantage of SDR concept.

As was seen in the presented examples by quick software change it is possible to create different hw systems processing real signals.

Q2 Which property of Fourier transform is used when we tune the receiver to the desired station?

Frequency shifting property.

Q3 Explain why low pass filter has to be placed after quadrature demodulator?

Because high frequency products resulting from mixing the input signal with its phase shifted copy have to be removed.

Q4 Which other ways of FM demodulators exists (besides quadrature demodulator)?

slope detector
ratio detector
Foster Seeley FM discriminator
phase locked loop demodulator

California State University, Fresno
Lyles College of Engineering
Electrical and Computer Engineering Department

TECHNICAL REPORT

Work Title: Artix-7 FPGA Bitstream Encryption Attack

Course Title: ECE 156 (Fundamentals of Crypto and Comp Network Security)

Date Submitted: May 16, 2022

Prepared By:	Sections Written:
Brenden Simpkins **Julius Andrade** **Trevor Ridley**	**Brenden Simpkins** **Julius Andrade** **Trevor Ridley**

INSTRUCTOR SECTION

Comments: Technical advisor: Dr. Hayssam El-Razouk

Final Grade: Team Member 1: _____

Team Member 2: _____

Team Member 3: _____

TABLE OF CONTENTS

Section	Page
Title Page ……	1
Table of Contents ……	2
1. Abstract ……..……	3
2. Literature Review ……...……	3
3. Theoretical Background ……...……	4
3.1 Required Hardware and Software……	4
3.2 Project Cost Estimate ……..……	4
3.3 Technical Specifications ……..……	4
3.4 Background ……..	6
3.4.1 Xilinx Artix 7 CW305 FPGA Target Board …….	6
3.4.2 Vivado 2021 … ……..…….	7
3.4.3 JTAG-HS3 Programming Cable.……...	7
3.4.4 USB-A to USB-B Cable ……	7
3.4.5 Cipher Block Chaining ……	7
4. Experimental Procedure …...….	7
4.1 Downloading an Encrypted Bitstream to the CW305 in Vivado……..	7
4.2 Attack to Break Confidentiality ……...	10
4.2.1 Attack Overview ……..……..	10
4.2.2 Normal Operation … ……..……...	14
4.2.3 Cipher Block Chaining.…….	14
4.2.4 HMAC ……	15
4.2.5 Configuration Header ……	16
4.2.6 Readout Bitstream ……...	16
4.3 Breaking Authenticity ……..……	16
4.4 Major Obstacles and Delays ……	17
4.5 Time Log …….	18
5. Analysis ……	18
5.1 Experimental Results …….	18
5.2 Data Analysis …….	22
6. Conclusion ……	22
6.1 Prevention ……..	22
6.2 Summary ……	22
7. References ……	23

1. ABSTRACT

As the demand for FPGAs increases in the technology industry, ensuring the security of FPGA devices has become a high priority. Demonstrating effective attacks on FPGAs can help develop adequate countermeasures to maintain security in the event of similar attacks. One such attack was developed recently in order to target the bitstream encryption of Xilinx 7 FPGAs. By making use of CBC malleability, specifically the tendency of errors to propagate through CBC data blocks, to alter unencrypted instructions to store the desired information in the WBSTAR register. The WBSTAR register does not clear after reset, allowing the data to be read out after power cycling the board. By repeating this process the encrypted bitstream can be retrieved word by word, breaking confidentiality. The last step is then to break authenticity by using the extracted data to allow for the generating of a fake HMAC tag that is accepted. The goal of this project is to successfully implement this attack on a Xilinx Artix 7 CW305 target board. The expected result is a successful attack that increases familiarity with the Xilinx design flow, experience with Xilinx FPGAs, and competency in launching embedded security attacks.

2. LITERATURE REVIEW

The primary source for information on this Xilinx 7 FPGA bitstream encryption attack is the work completed in [1], which is the original publication documenting the entirety of the attack. The two primary stages of the attack are described as breaking confidentiality and breaking authenticity. In addition to the detailed process of the attack the work includes an overview of the pitfalls that allowed the attack to be implemented. Section 5 introduces countermeasures against the attack, one of which is validating the configuration header itself prior to any other actions taking place. Shortcomings of the proposed solutions are also explored, such as the inability to modify the actual silicon based hardware design of the already-manufactured Xilinx 7 boards.

Another very important source of information is the Xilinx 7 Series FPGAs Configuration Guide [2]. This document details the steps needed to properly configure and utilize the Xilinx 7 FPGA devices. The guide consists of key information applicable to all Xilinx 7 FPGAs. Constraints and design flow limitations are also documented, such as the need to use a JTAG cable to implement bitstream AES encryption. AES key storage options are also documented, with the two primary options mentioned in the document being BBRAM and eFUSE. Between this user guide and the aforementioned attack publication, most of the information required to implement the attack is present.

Some of the sources that were used were some of the support pages from the xilinx forums that help with most of the obstacles that we encountered. One of the forums that was used helped with the implementation for generating an encrypted bitstream as it broke down the process on understating how to set up the proper settings to be able to achieve this. It also goes over in detail how to go into the settings and program the BBR key.

3. THEORETICAL BACKGROUND

3.1 Required Hardware and Software

Hardware:

- Xilinx Artix 7 CW305 FPGA Target Board
- Digilent JTAG-HS3 Programming Cable
- USB-A to USB-B Cable
- Attacking PC

Software:

- Vivado/ISE Design Suite Webpack Edition
- Hex Editor

Justification:

- The Artix 7 C305 FPGA is target device of the attack
- The JTAG-HS3 Programming Cable was a strict requirement to encrypt the bitstream to the target FPGA from the attacking PC
- The USB-A to USB-B Cable was required to provide power to the CW305 board

3.2 Project Cost Estimate

Xilinx Artix 7 CW305 FPGA Target Board ---------- $1,050
Digilent JTAG-HS3 Programming Cable ---------- $59
USB-A to USB-B Cable ---------- INCLUDED WITH CW305

Total Cost ---------- $1,109

3.3 Technical Specifications

Xilinx Artix 7 CW305 FPGA:

- The Target Device is the Xilinx Artix-7
- Can be programmed via USB and JTAG(bitstream encryption is only possible with JTAG)
- Hosts a 40-pin header mounted to the board labeled as JP3
- Is a standalone target, hardware other than the PC is not necessary for loading to the board
- An image of the Xilinx Artix 7 CW305 FPGA is shown below in **Figure 3.1**.

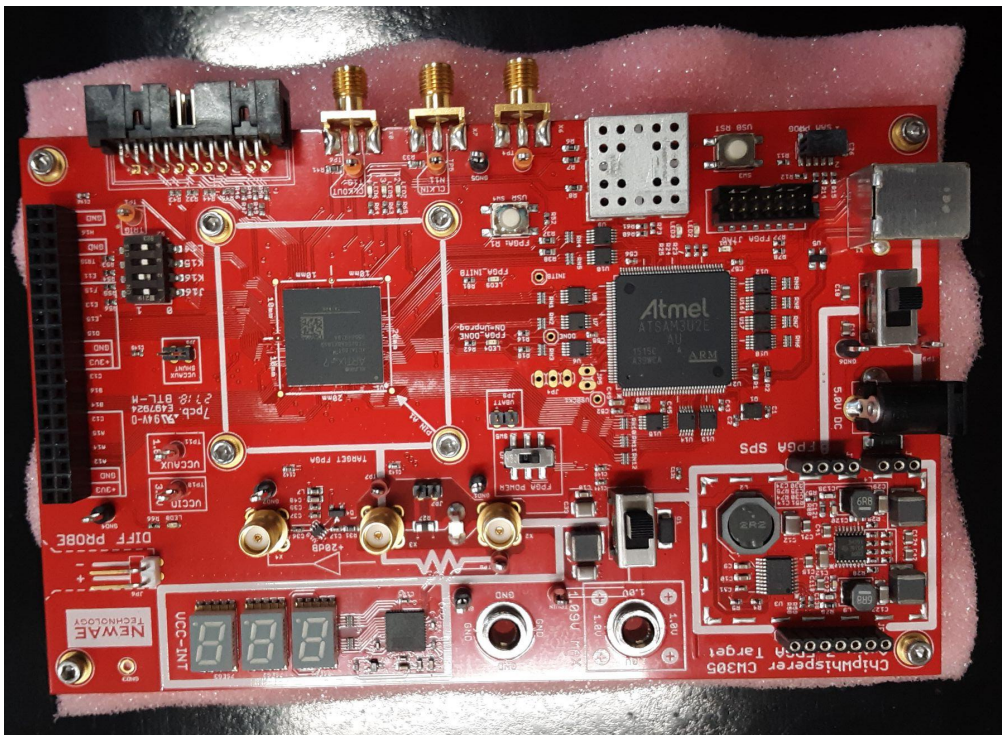


Figure 3.1: CW305 FPGA Target Board

Digilent JTAG-HS3 Programming Cable:

- Compatible with Xilinx iMPACT, ChipScope, and EDK tools
- Supports JTAG signal voltages between 1.8V and 5V
- Compatible with Xilinx ISE 14.1 and any subsequent versions
- Max speed of 30 megabits per second
- An image of the Digilent JTAG-HS3 Programming Cable is shown below in **Figure 3.2**.

Figure 3.2: JTAG-HS3 Programming Cable

USB-A to USB-B Cable:

- Typical USB cable commonly used for interfacing between USB-A and USB-B
- An image of the Digilent JTAG-HS3 Programming Cable is shown below in **Figure 3.3**.

Figure 3.3: USB-A to USB-B Cable

3.4 Background

3.4.1 Xilinx Artix 7 CW305 FPGA Target Board

The CW305 is a target board for use in projects and attacks with a focus on embedded security created by NewAE Technology Inc. to be used with the Chipwisphere software which uses an Artix 7 FPGA developed by Xilinx. Typical use scenarios include side-channel attack implementations, bitstream encryption/decryption implementations, and physically unclonable function implementations. The cables required for an implementation depend on exactly what software or external devices that are used to interact with the board. If the Chipwhisperer software is used along with other chipwhisperer products only the USB cable must be connected and interaction is through the python coding language. For this project direct access to the FPGA is needed so the JTAG connection is used with the JTAG-HS3 cable to connect Vivado for bitstream encryption and the USB cable to provide power to the CW205 board.

3.4.2 Vivado 2021

This software is used for synthesis and analysis for hardware designs to communicate with the CW305 FPGA that was used to produce an encrypted bitstream that is delivered with the JTAG cable. The software tool is used in creating many designs that can be implemented into the FPGA that can consist of different languages which include hardware description language files, or block design, which could include a variety of IP blocks.

3.4.3 JTAG-HS3 Programming Cable

The JTAG-HS3 cable is actually a term representing two separate components. The first component is a USB to micro USB cable, and the second component is a micro USB to 7x2 pin JTAG adapter. The cable is used by plugging the adapter into the JTAG slot on the Xilinx 7 board and connecting the micro USB side of the adapter to the USB slot of a computer using the USB to micro USB cable. One alternative is the xilinx platform cable, but this solution is much more expensive. It is important to understand that it is not possible to complete this project without a JTAG cable, as it is required for bitstream encryption to the FPGA.

3.4.4 USB-A to USB-B Cable

The USB-A to USB-B cable is a common cable used for many devices. When it comes to connecting to the CW305 target board, the cable serves two purposes. The first is to download a bitstream to a board when the board is set to be configured by chipwhisperer using python code. The second is to provide the board power no matter the communication method. Power to the board can be provided through the 5 Volt DC input or the USB cable, but the USB cable is what was used for this project.

3.4.5 Cipher Block Chaining

Cipher Block Chaining, or CBC, is one of the operation modes for a block cipher, and is used to perform block encryption and decryption. The most important trait of CBC for the purposes of this project is CBC malleability. This means that because CBC encryption relies on the XORing of the decryption of the current ciphertext block with the previous ciphertext block to generate the current plaintext. This XOR chaining leads to any errors or modifications made to the initial block propagate to future blocks. An attacker can use this to their advantage by making modifications to ciphertext blocks in an attempt to set data of certain registers to the desired values, the target in this case being the WBSTAR register.

4. EXPERIMENTAL PROCEDURE

4.1 Downloading an Encrypted Bitstream to the CW305 in Vivado

The first step to implementing the attack on the CW305 is to successfully download an encrypted bitstream to the CW305 board. There are a number of steps needed to do this. First, the basic Vivado design flow was followed creating a simple one bit adder for testing. Next, the S1 switches on the back of the CW305 were set to 101 enabling JTAG programming mode. Once this was done the encryption settings were changed in order to enable AES bitstream encryption. These settings were accessed by selecting “implemented design”, then clicking the “tools” dropdown menu and selecting “edit device

properties". Lastly, "encryption" was clicked in the vertical list on the left side of the window. Once inside the encryption settings, several changes were made to the default settings in order to configure Vivado to enable bitstream encryption. The encryption settings were set to simple values to allow for reproduction for testing purposes, HMAC authentication KEY to 256 bits of 'A' repeated 64 times, AES encryption key to 256 bits of 'B' repeated 64 times, and starting CBC value to 128 bits of 'C' repeated 32 times. These changes are listed below.

1. "Enable Bitstream Encryption" was set to YES
2. "Select location of encryption key" was set to BBRAM
3. "HMAC Authentication key" was set to a 256 bit value
4. "AES encryption key (key0)" was set to a 256 bit value
5. "input encryption file" was left blank
6. "Starting cipher block chaining (CBC) value" was set to a 128 bit value

The resulting device properties configuration is shown in **Figure 4.1** below.

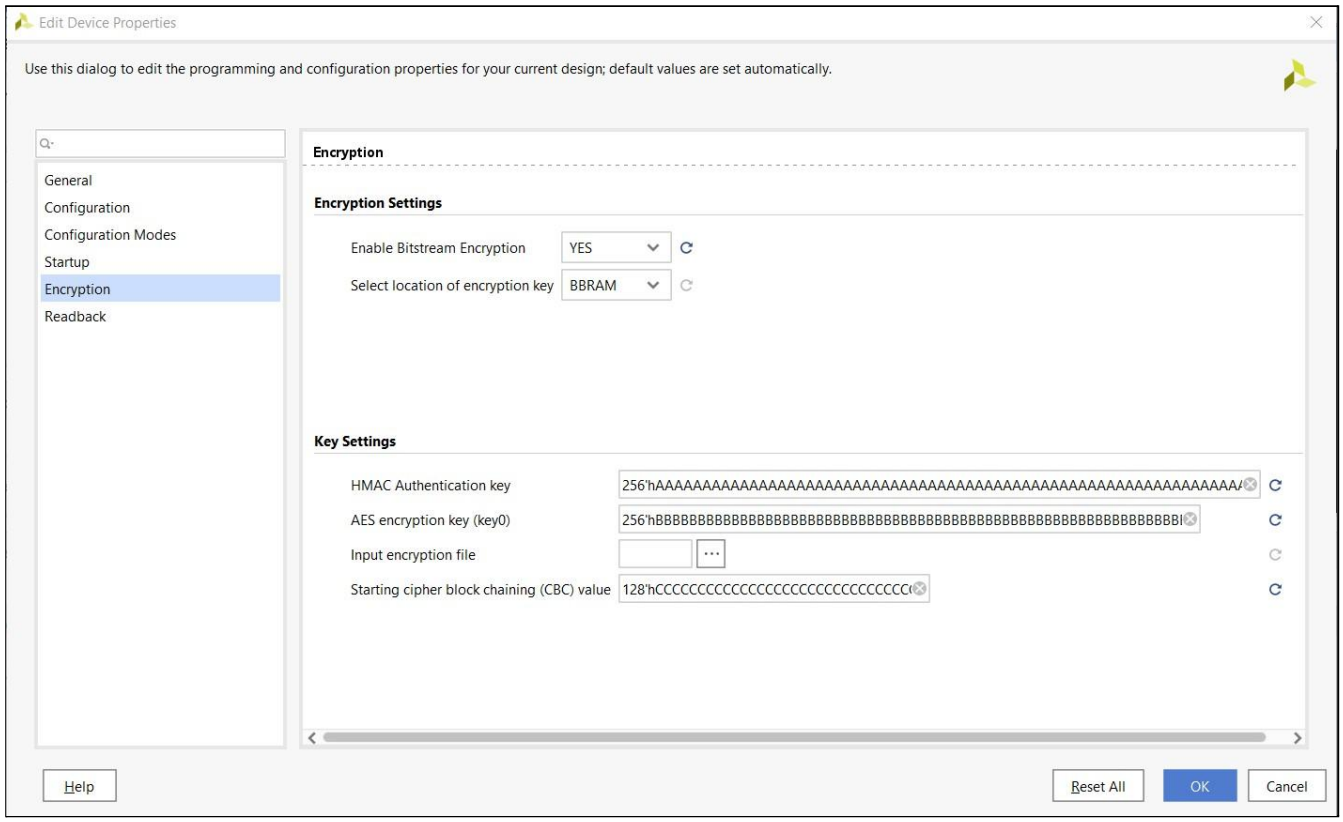


Figure 4.1: Encryption Settings

After changing the settings as described above, the next major step was to load the AES key to the Battery-Backed Random Access Memory (BBRAM). This was done by right clicking on the name of the FPGA device in the hardware manager window and selecting "program BBR key". A NKY keyfile must then be specified for loading. This keyfile should be prepared using a format similar to what is shown below in **Figure 4.2**.

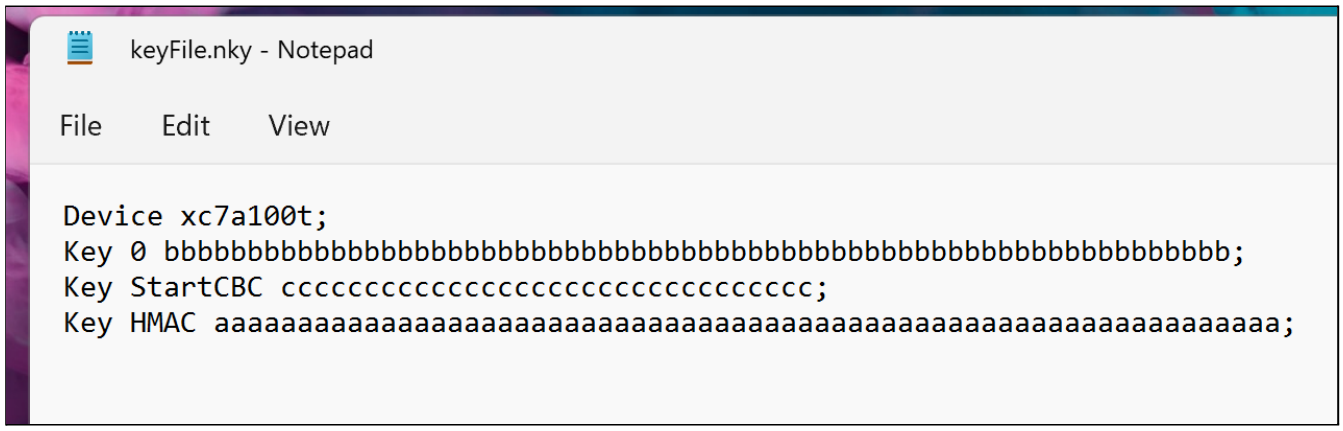


Figure 4.2: NKY File Format

After correctly storing the AES key on the BBRAM the encrypted bitstream was ready to be downloaded to the CW305 board. This was done by clicking Having successfully loaded an encrypted bitstream to the board, the next phase was to begin the process of launching an attack with the goal of compromising the bitstream security, specifically the confidentiality and authenticity. For the purposes of testing, a one bit adder was loaded to the board and tested by controlling the input pins via the GPIO header where the two inputs were tied to one input pin each and the output to another pin. The output of the and gate was monitored using a digital multimeter. The results are shown below in **Figure 4.3** and **Figure 4.4**.

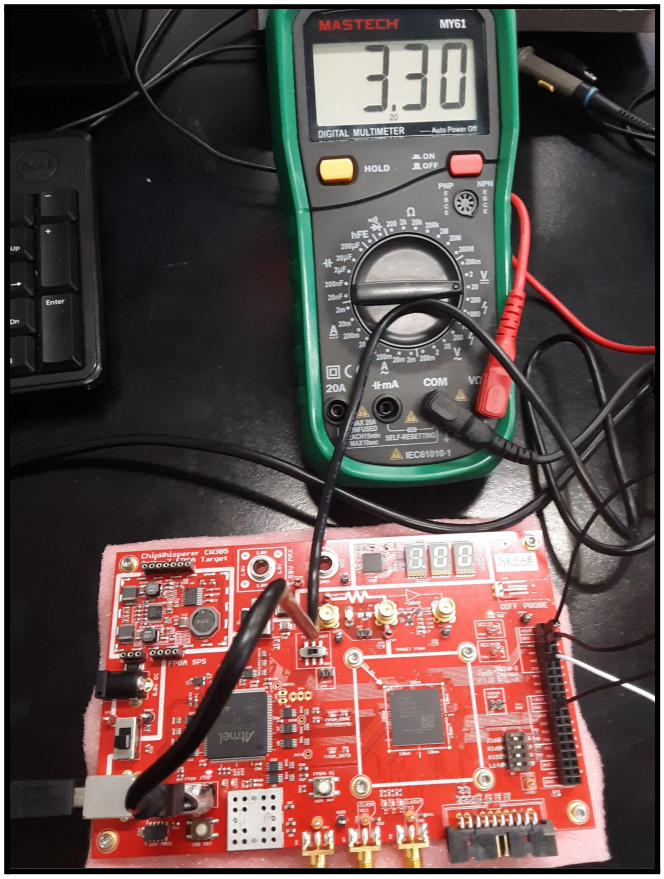


Figure 4.3: Adder gate with one high input

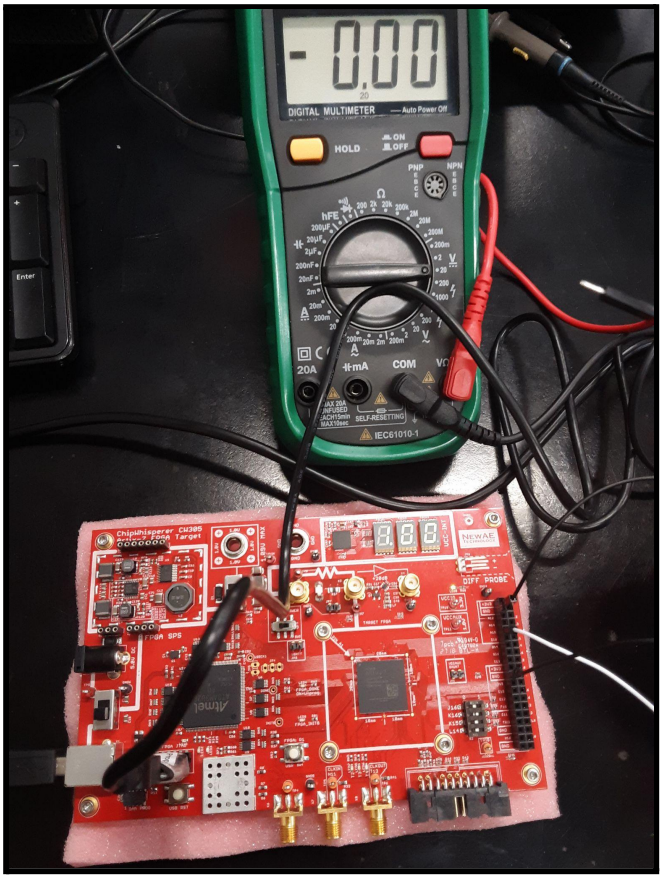


Figure 4.4: Adder gate with two high inputs

In the figures above it is clear that the one bit adder design was successfully programmed to the board with bitstream encryption enabled. As the GPIO pins have pull up resistors when neither of the input pins are connected to ground the input is 1 + 1 = 0 as shown in **Figure 4.4**, when one of the pins is forced to ground the result is 1 + 0 = 1 as shown in **Figure 4.3** with the high output of 3.3v.

This shows that a valid bitream was created to program the FPGA with a valid decryption and valid HMAC header and tail.

4.2 Attack to Break Confidentiality

4.2.1 Attack Overview

In total the attack on the bitstream confidentiality is to implement an attack which allows the unencrypted plaintext of a confidential bitstream which is encrypted to be known to an unauthorized attacker. It is assumed that the attacker has access to an FPGA with the correct keys to use an encrypted bitstream that describes a desired design of a rival corporation (or other motive).

This is done by manipulating the CBC portion of the AES decryption and an overlooked reset behavior of the FPGA to store a word (32 bits) into the WBSTAR register. This stored word would successfully pass through the AES decryption before being placed into the WBSTAR register so this word will be

plaintext unencrypted. The method used to manipulate the CBC will cause the FPGA to do a reset where the FPGA resets almost all registers with the exception of the WBSTAR register (this is the main weakness that is exploited).

A second unencrypted bitstream (named the ‘readout’ bitstream) is used that simply requests from the FPGA to read out the value in the WBSTAR register to the JTAG. As the bitstream that is used is unencrypted the value in WBSTAR is read out of the FPGA to the JTAG where the attacker now has successfully extracted one word (32 bits) of plaintext data of the encrypted bitstream (and they know exactly where that data is located in the encrypted bitstream). By repeating the attack over and over again each word of the design can be extracted at a time.

The unaltered bitstream generated that was tested in **Figure 4.3** and **Figure 4.4** is shown below in **Figure 4.5** opened in a hex editor.

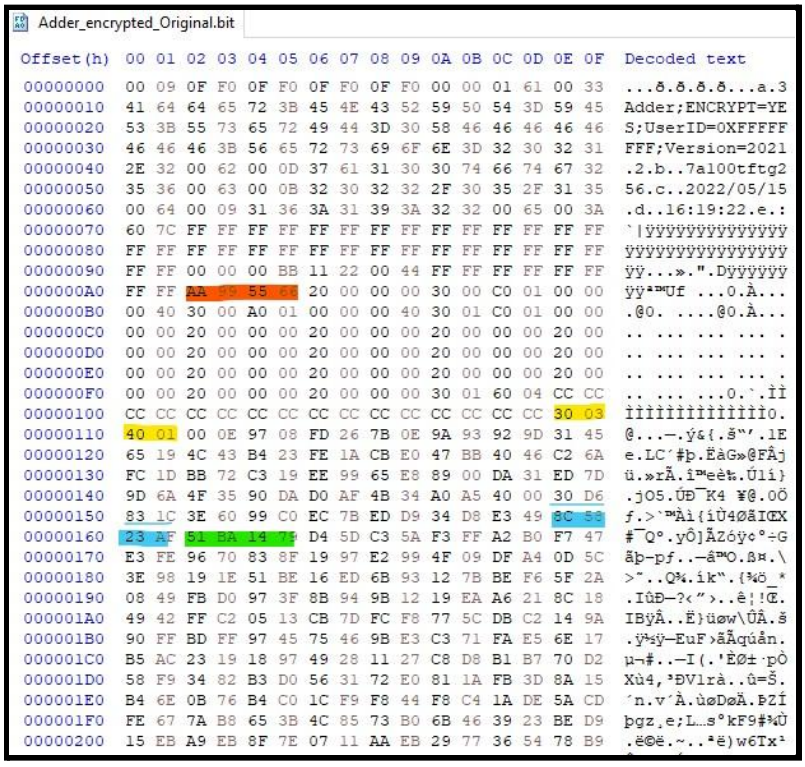


Figure 4.5: Unaltered_Encrypted_Bitstream

We also created the same program again but made the version that was unencrypted to show the unencrypted bits as shown in **Figure 4.6**.

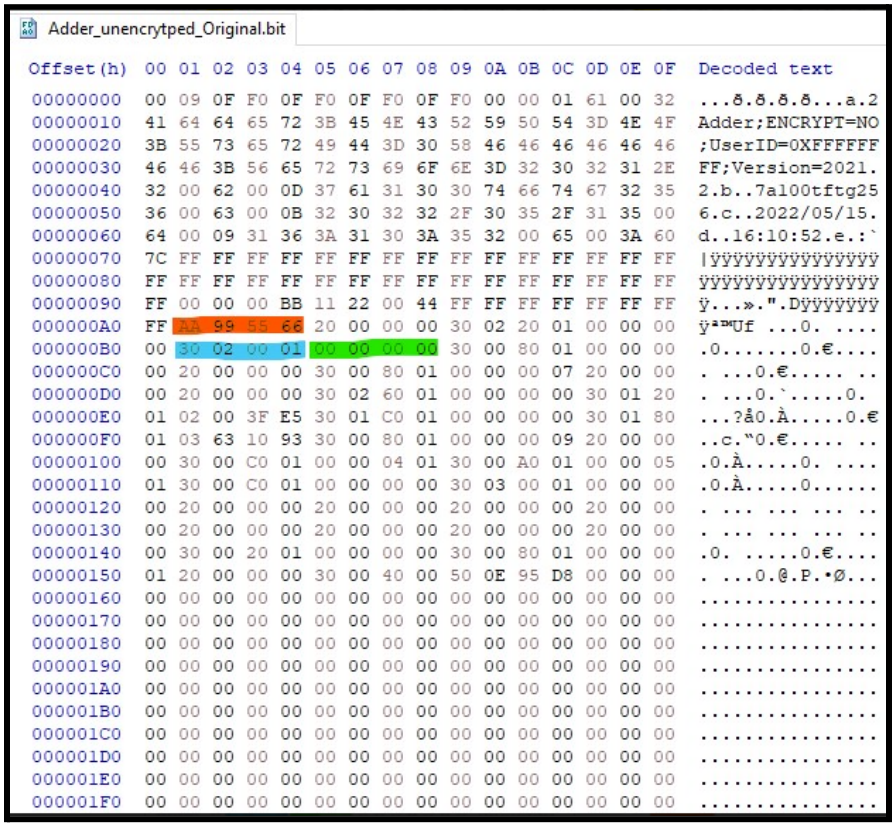


Figure 4.6: Unaltered_Unencrypted_Bitstream

It is shown that there are many bytes of information at the beginning of the file and this information includes identifying information such as if encryption is used, ID number, vivado version used, and other information that is used by the system to identify the contents of the file (this information can vary in size). A large array of 0xFF bytes are used to pass the information followed by the sequence ‘00 00 00 BB 11 22 00 44” which describes some of the bus width information. More 0xFF bytes follow and then the important sequence ‘AA 99 55 66’ is shown which is the ‘sync’ instruction. A ‘NOP’ instruction follows with ‘20 00 00 00’.

There are three ‘types’ of instructions used in the bitstream named ‘Type 0’, ‘Type 1’, and ‘Type 2’. The instructions are 32 bits (4 bytes). Relevant information is shown below in **Figure 4.7**, **Figure 4.8**, **Figure 4.9**, **Figure 4.10** which are from the ‘7 Series FPGA Configuration User Guide’[2].

Header Type	Opcode	Register Address	Reserved	Word Count
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Figure 4.7: Type 1 Layout

OPCODE	Function
00	NOP
01	Read
10	Write
11	Reserved

Figure 4.8: Opcode Options

WBSTAR	Read/Write	10000	Warm Boot Start Address Register

Figure 4.9: Type 2 Layout

The important instruction is the ‘Type 1’ as these instructions instruct how the information is moved into the control registers and the FPGA fabric. The first 3 bits identify the type of instruction which has the value of 0b001 to select ‘Type 1’. The 2 bit opcode determined if this instruction was a NOP, Read, Write, or simply Reserved for special operation. A register address field is 14 bits long where only the lowest 5 bits are used to identify control registers. The last field is the word count which is a count for the number of words that will be pushed to this register (some instructions which push data to the FPGA fabric can move lots of data into the FPGA with a single instruction followed by many bytes of payload). Some registers like the WBSTAR register can only hold a single 32 bit word value so only the last word remains after the entire payload is pushed to the register (if 3 words are pushed to the WBSTAR register each word is written to the register one at a time resulting in only the last word remaining in the register as all previous ones are overwritten by the payload word after it).

The most important instructions are the writing to the WBTAR register such as ‘30 02 00 01’ which is written to WBSTAR register with a word count of 1 (001 10 00000000010000 00 00000000001), this instruction would write the next 32 bits to the WBSTAR register. To read from the WBSTAR Register the instruction ‘28 02 00 01’ which reads out the word in the WBSTAR register to the JTAG connection (001 01 00000000010000 00 00000000001).

The ‘Type 2’ is a payload extension which is meant to be used after a ‘Type 1’ instruction with a ‘word count’ of 0 where the ‘Type 2’ instructions ‘word count’ acts as the word count for the instruction before it.

Header Type	Opcode	Word Count
[31:29]	[28:27]	[26:0]
010	XX	xxxxxxxxxxxxxxxxxxxxxxxxxx

Figure 4.10: WBSTAR Register Address

If encryption is enabled any word coming through JTAG into the FPGA goes through AES to be decrypted into plaintext and be operated on, and any word read out from the FPGA goes through AES to be encrypted into cipher text before exiting through the JTAG port. Anyone can connect to a FPGA running a certain design and pull out the encrypted bitstream but this attack is a method to bypass the AES encryption by abusing the WBSTAR register.

4.2.2 Normal Operation

Under normal Operation the FPGA is reading the bitstream from the JTAG port. The first part of the bitstream is always unencrypted to provide basic information about the file. After the 'sync' instruction and some other instructions a '30 03 40 01' instruction is 'encrypt start' where all bytes after this instruction are encrypted and are diverted through the AES decryption before being processed one word at a time. These decrypted plaintext words are processed normally. When any data is requested to be read out of a register the value is processed normally and then passed through the AES encryption. The encrypted ciphertext is then output from the FPGA to the JTAG port.

4.2.3 Cipher Block Chaining

This attack utilizes the fact that the AES module is functioning in CBC mode. This means when a dataword is decrypted (in 128 bit blocks, aka 4 words at a time) the resulting plaintext is also XORed with the cipher text that was part of the decryption the round before ($P_n = C_{n-1}$ XOR AES_Decrypt(C_n)).

This means that if an attacker has a good idea of what the value of AES_Decrypt(C_n) will be they can change the value of C_{n-1} to force the decrypted value to be whatever the attacker desires. This will make AES_Decrypt(C_{n-1}) a random value as C_{n-1} was altered in a hex editor to be a certain value which may or may not lead to issues depending on what the resulting value is. This may cause problems as this instruction comes before the altered instruction P_n by 4 words.

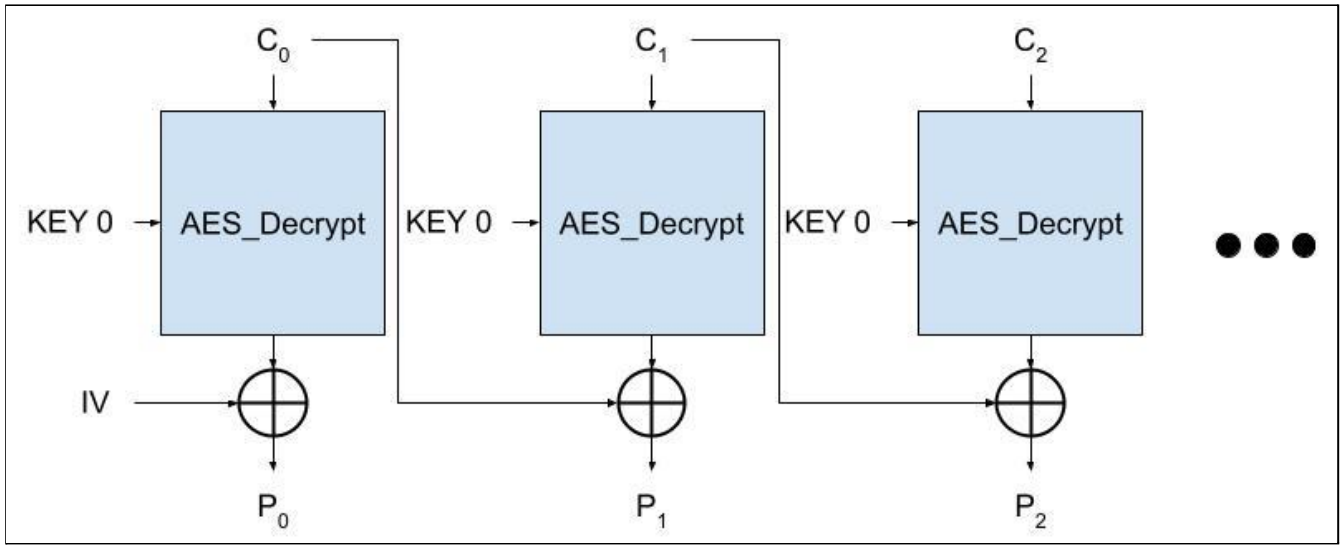


Figure 4.11: CBC Malleability Pre-Modification

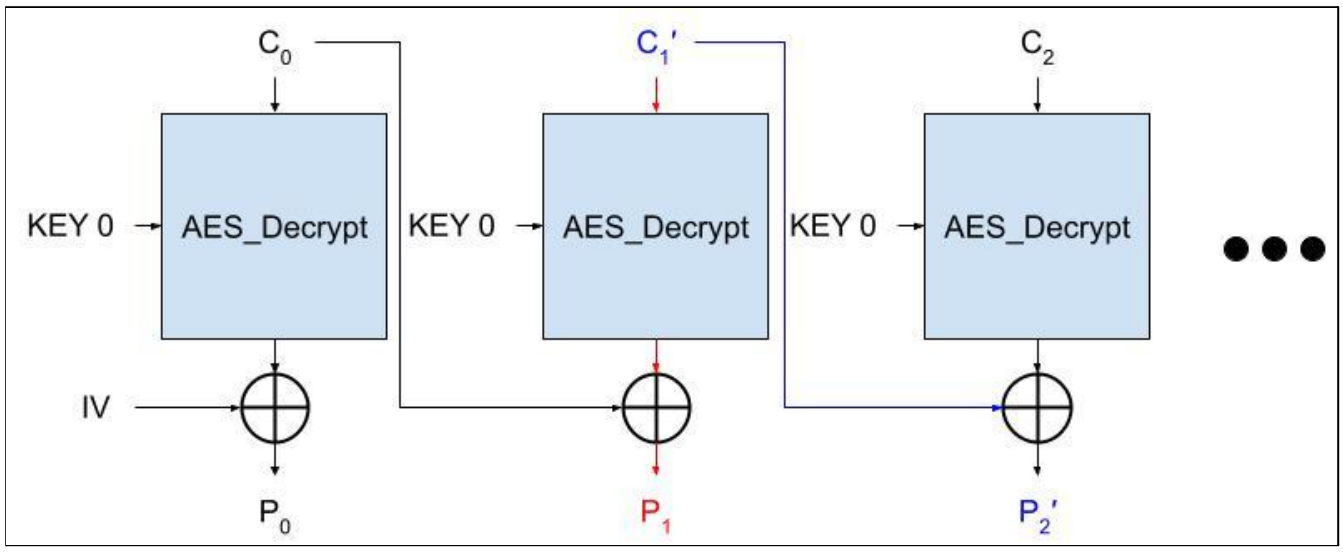


Figure 4.12: CBC Malleability Post-Modification

To prevent errors when executing the CBC two main tasks need to be completed.

1. The first is that the AES_Decrypt(C_{n-1}) needs not to throw errors or some other weird behavior that ends the program before the malicious P_n is executed.
2. The unencrypted value of the chosen C_n needs to be known to allow useful XOR of the altered C_{n-1} to make P_n be an instruction which achieves a desired value (this is not a hard requirement but can speed up the repeated process as it removes unexpected errors).

Both of these goals are achieved due to the known positioning of the 'HMAC header' and the 'Configuration Header'.

4.2.4 HMAC

The authentication of the bitstream that is being loaded into the FPGA comes with a created HMAC header and HMAC footer that sandwiches the fabric data which programs the FPGA. The FPGA will load the entire bitstream into the FPGA normally and once the HMAC footer is reached a check is done to determine if any of the values within the fabric data have been altered resulting in a failed HMAC authentication. If an error is detected the FPGA will shutdown, clear most of the control registers, and clear the fabric before it is activated to prevent possible damage.

The attack relies on the subtle detachment that not 'all' of the control registers are reset only most. One of the registers is the WBSTAR register as this register points to where some startup boot drive can be located so losing this variable information may accidentally brick the FPGA so it is not at reset. This means that any value placed into the WBSTAR register will be retained in that register even after a reset of the FPGA where it can be extracted.

As any manipulation of the HMAC header or footer or the the contents between them will result in a reset of the FPGA it does not matter which C_{n-1} we choose as long as the resulting P_{n-1} does not cause premature termination of the bitstream read if it is an value that causes an error. The solution to this is to choose C_{n-1} to be a value in the HMAC header as these values are intended to be random and are

only checked when the HMAC footer is reached. This prevents any error caused by a random P_{n-1} to result in premature termination of the bitstream load before the malicious P_n moves a value into the WBSTAR register.

4.2.5 Configuration Header

Another important feature about the HMAC header is that it is located right before the 'configuration header' for the fabric data. This 'configuration header' is well known in its layout including that the first 4 words are an instruction to write to the TIMER register, the second word is the value written to the TIMER register of zero, the third word is a write instruction to the WBSTAR register, and the fourth word is the value written to the WBSTAR register is zero. The first word writing to the TIMER register could be used as it is known but the simplest option is to choose the write to WBSTAR register instruction to be the target of the malicious attack.

By changing the encrypted value 4 words before the third word in the 'configuration header' (this changed word is the 15th word in the HMAC header) the unencrypted value of the write to WBSTAR register can be changed. The change that is desired is the 'word count' of the write to WBSTAR instruction from 1 (which would load the next word of 0x0 into the WBSTAR register) to a larger value such as 7 which will load that word 7 words away into the WBSTAR register that has been decrypted (as the last word to be 'loaded' into the register stays in the register).

After the malicious write to WBSTAR register instruction is executed and the desired plaintext word is loaded into the WBSTAR register any actions in the loading of the bitstream are irrelevant. The HMAC fails and the FPGA resets leaving the plaintext word in the WBSTAR register.

4.2.6 Readout Bitstream

A created bitstream that is unencrypted is loaded into the FPGA after the reset. As this bitstream is unencrypted and does not include the '30 03 40 01' instruction of 'encrypt start' the AES module is not used (the FPGA at this state is suppose to be 'empty' so there would be no reason to require encryption unless an encrypted design was being loaded to the FPGA). Other than the required sync words the instruction of '28 02 00 01' is used to read out 1 word from the WBSTAR register. The FPGA reads out the value currently in the WBSTAR register which is the unencrypted plaintext word to the JTAG to be recorded by the attacker.

The attack would then repeat to extract another word from the encrypted bitstream.

Breaking the Confidentiality allows for design cloning and reverse engineering of industrial design secrets or possibly military secrets depending on how the FPGA is used.

4.3 BREAKING AUTHENTICITY

The work completed in [1] describes how to use the CBC malleability and manipulating values in the bitstream using a previously discovered method cited by Rizzo and Duong to 'move up the CBC chain'

until the IV can be set to any value that the attacker desires using a ‘encryption oracle’ and ‘decryption oracle’. This process is not elaborated in detail by the original paper but does describe a general process.

For a given block of encrypted ciphertext C_i there is a decrypted plaintext named P_i and there is a encrypted ciphertext block that is decrypted before it named C_{i-1}. The value of plaintext is P_i = AES_decrypt(C_i) XOR C_{i-1}. If we want to generate our own plaintext P_i' to replace a generic P_i' a value C_{i-1}' will need to be found which generates the P_i' desired. To calculate C_{i-1}', $C_{i-1}' = P_i$ XOR C_{i-1} XOR P_i'. The first equation can be used to calculate a resulting plaintext value P_{i-1} when C_{i-1}' is enforced and then the second equation to calculate C_{i-2}'. This process can keep moving up the CBC chain until the very first P_1' is found which is caused by changing C_0'. The value of C_0' is the IV of the CBC mode, as this is in the header this can also be changed.

This change along with using the extraction of plaintext words from anywhere in the encrypted bitstream results in the HMAC encryption key being part of the encrypted bitstream and with full control of the CBC can change the HMAC header and footer at will. The attacker only has to write their desired design, generate the corresponding HMAC tags to validate it and then load the completely malicious bitstream into the FPGA. The attacker successfully broke the authenticity of this bitstream which now allows the attacker to have the fake authority to run any design they want on the FPGA without directly breaking the AES encryption (simply going around it with CBC malleability).

This has major implications where designs can be sabotaged such as in the form of hardware trojans which can have major implications if the FPGA is used for industrial or military usage.

4.4 MAJOR OBSTACLES AND DELAYS

A number of obstacles appeared during the attempt to perform this attack on the Xilinx-7 FPGA bitstream encryption, and these played a large part in delaying the project. The first was that the project began without the proper JTAG cable. Due to inexperience with Xilinx FPGAs and the unique behavior of using specific Chipwhispere hardware the team members initially assumed that JTAG was being implemented over the USB connection, similar to how it is done with Altera FPGAs. After several meetings and attempts over many weeks to debug failed connectivity with the CW305 board it was determined that an entirely different cable was required for the JTAG connection. Some time later the required cable was obtained, allowing the team to work their way through the Xilinx documentation to program the CW305 FPGA with an encrypted bitstream. This process took most of the time spent on the project with many weeks of meeting and attempting to go through the documentation to solve the problem.

The next major problem was reading out the WBSTAR configuration register. The team attempted all known methods of reading out this register. The first attempted method was using ChipWhisperer and python code. They were able to read out data registers using this method along with the CW1200 scope, but the values written out did not correspond to the WBSTAR register.

The second method attempted involved starting a Vivado project with the CW305 and moving it into Vitis in order to code a C language program in eclipse to read out the configuration registers. This also failed because the CW305 device was not supported for use in Vivado leading to issues with Vitis. While it could be used for basic encrypted bitstream programming, due to the lack of hardware support it could not be used for programming C programs to the board using Eclipse in Vitis.

The third method attempted was the one used in [1], which states in the section 3.1 under "Configuration Interfaces" that "debug readout" using the JTAG interface is a necessary part of this attack. The team attempted to use the XSDB(Xilinx System Debugger) to read out the register values of the CW305 FPGA. The team got as far as selecting the actual device and attempting to read from the WBSTAR register address, but the read failed. The team later learned that when trying to access the JTAG UART the XSDB informed us that the connected device (CW305 board) did not appear to support JTAG UART that XSDB could use, meaning that the configuration register values could not be read out using the debugger.

It seems likely that there is some methodology to properly read out the configuration registers of the CW305, but all attempts by the team to do so did not succeed. Reading out the memory was a vital part of the attack and required to verify the breaking of confidentiality and by extension breaking the authenticity. These problems made it so the team was unable to make progress after encountering this problem. Were it not for other delays there may have been sufficient time to determine a method to read the output from the WBSTAR register within the allowed time frame.

4.5 TIME LOG

The team began meeting to work on the project on the 25th of March. All three team members met for 3 hours on each Friday until their last meeting on May 6th. The team met again for an average of 3 hours a day for 4 days meeting on May 11, 12, and 16 in person. In total the team spent at least 30 hours in person working on the project. Additional time was also spent outside of in person meetings researching solutions, contacting the NEWAE forums, vivado forums, and creating materials for the project.

5. ANALYSIS

5.1 Experimental Results

The experimentation was only able to proceed to the stage of enacting the malicious bitream attack but not extracting the word written to the WBSTAR register. This is due to a problem of actually being able to view any messages from the FPGA through the JTAG. All methods that were tried to enable this viewing of the JTAG were not fruitful making it impossible to verify that the changes made in the malicious bitstream were successful.

The unaltered unencrypted bitstream is shown below in **Figure 5.1** with the ‘sync’ instruction highlighted. The ‘write to WBSTAR register for 1 word count’ instruction is shown a few lines below in blue with the value of ‘30 02 00 01’. This bitstream was written to the board and the implemented one bit adder (XOR gate) to the FPGA and tested using a multimeter to verify that the design was loaded correctly.

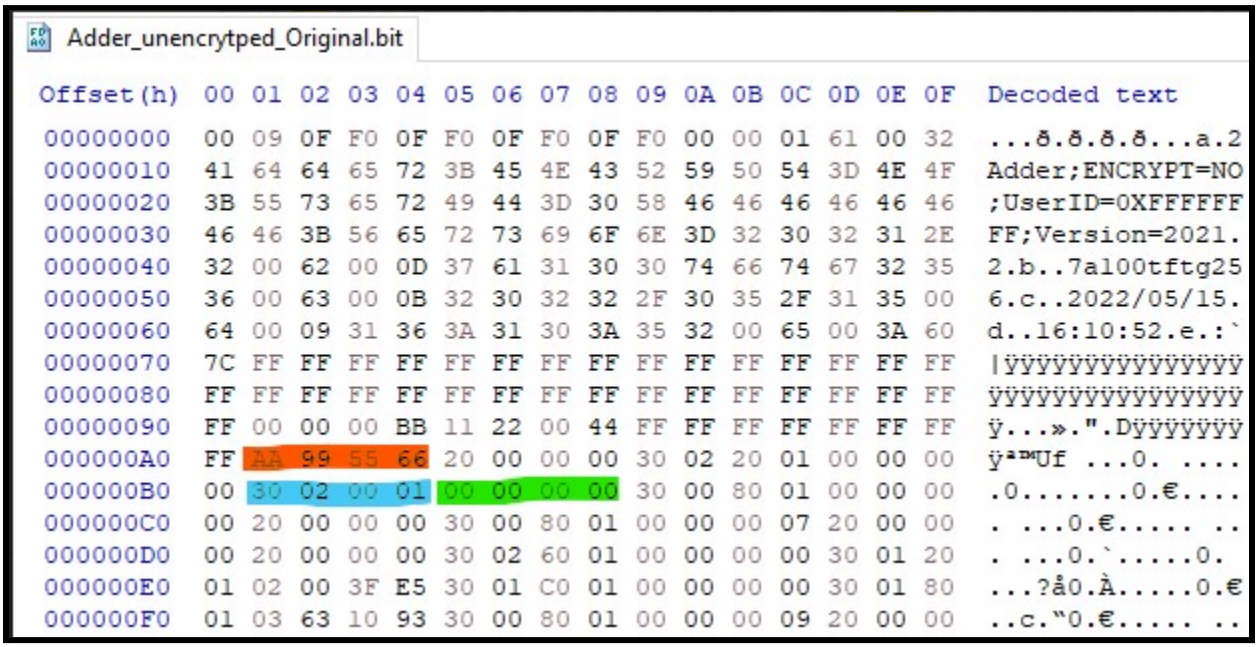
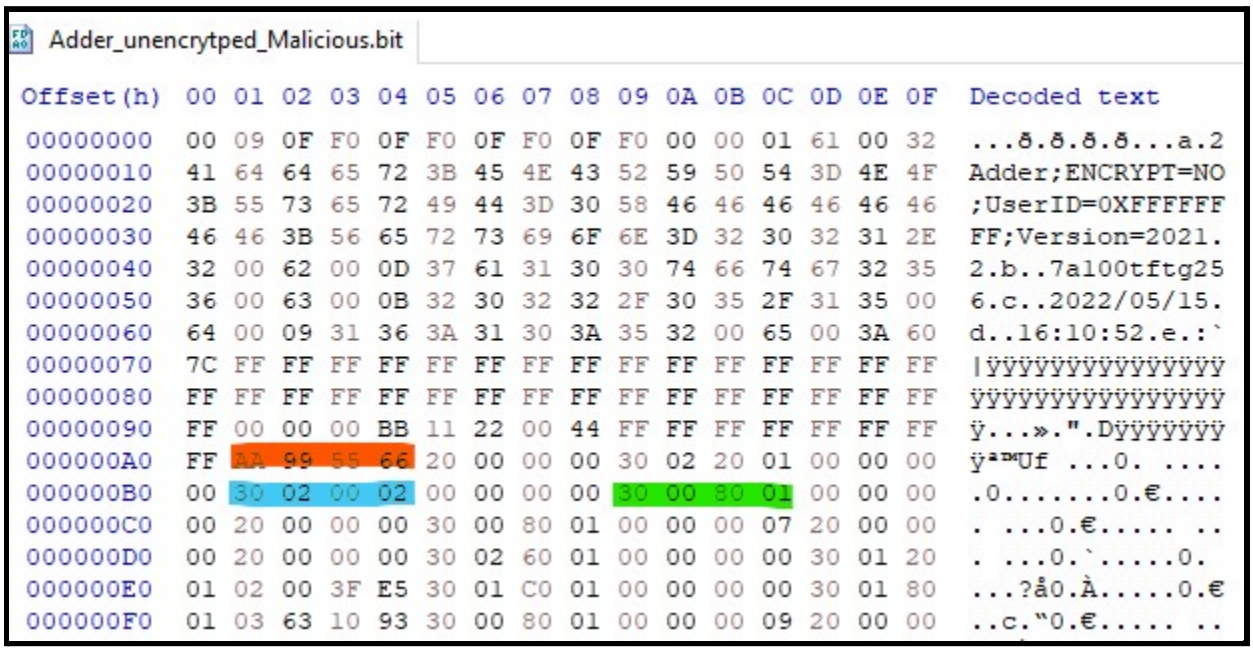


Figure 5.1: Unaltered Unencrypted Bitstream

The malicious attack changes the ‘word count’ from 1 to 2 to attempt to write the value of ‘30 00 80 01’ into the WBSTAR register to save it. The altered malicious unencrypted bitstream is shown in **Figure 5.2.** When written to the FPGA and the FPGA reset itself as expected.

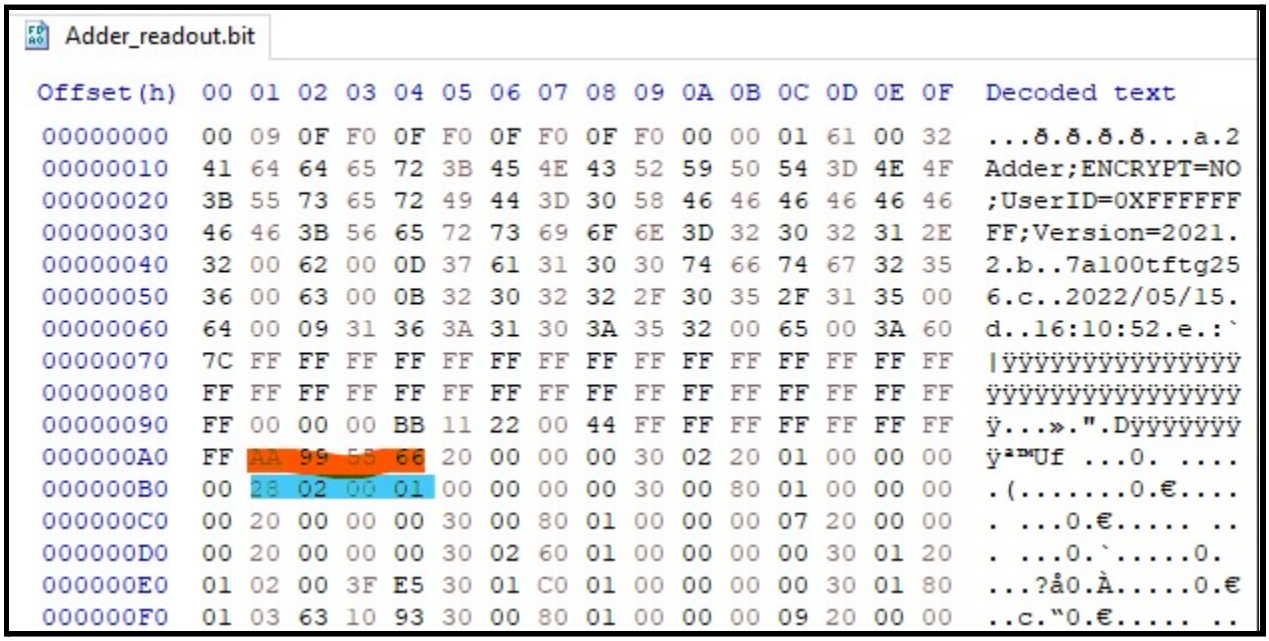


Adder_unencrytped_Malicious.bit

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	00	09	0F	F0	0F	F0	0F	F0	0F	F0	00	00	01	61	00	32	...ð.ð.ð.ð...a.2
00000010	41	64	64	65	72	3B	45	4E	43	52	59	50	54	3D	4E	4F	Adder;ENCRYPT=NO
00000020	3B	55	73	65	72	49	44	3D	30	58	46	46	46	46	46	46	;UserID=0XFFFFFF
00000030	46	46	3B	56	65	72	73	69	6F	6E	3D	32	30	32	31	2E	FF;Version=2021.
00000040	32	00	62	00	0D	37	61	31	30	30	74	66	74	67	32	35	2.b..7a100tftg25
00000050	36	00	63	00	0B	32	30	32	32	2F	30	35	2F	31	35	00	6.c..2022/05/15.
00000060	64	00	09	31	36	3A	31	30	3A	35	32	00	65	00	3A	60	d..16:10:52.e.:`
00000070	7C	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	|ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000090	FF	00	00	00	BB	11	22	00	44	FF	FF	FF	FF	FF	FF	FF	ÿ...».".Dÿÿÿÿÿÿÿÿ
000000A0	FF	AA	99	55	66	20	00	00	00	30	02	20	01	00	00	00	ÿª™Uf ...0. ....
000000B0	00	30	02	00	02	00	00	00	00	30	00	80	01	00	00	00	.0.......0.€....
000000C0	00	20	00	00	00	30	00	80	01	00	00	00	07	20	00	00	. ...0.€..... ..
000000D0	00	20	00	00	00	30	02	60	01	00	00	00	00	30	01	20	. ...0.`.....0.
000000E0	01	02	00	3F	E5	30	01	C0	01	00	00	00	00	30	01	80	...?å0.À.....0.€
000000F0	01	03	63	10	93	30	00	80	01	00	00	00	09	20	00	00	..c."0.€..... ..

Figure 5.2: Malicious Unencrypted Bitstream

The readout bitstream was applied to the FPGA. However, we could not read the fpga terminal so we could not verify that it works(as mentioned in section 4.4). Assuming that the unencrypted attack worked as expected the encrypted method would be tested next.

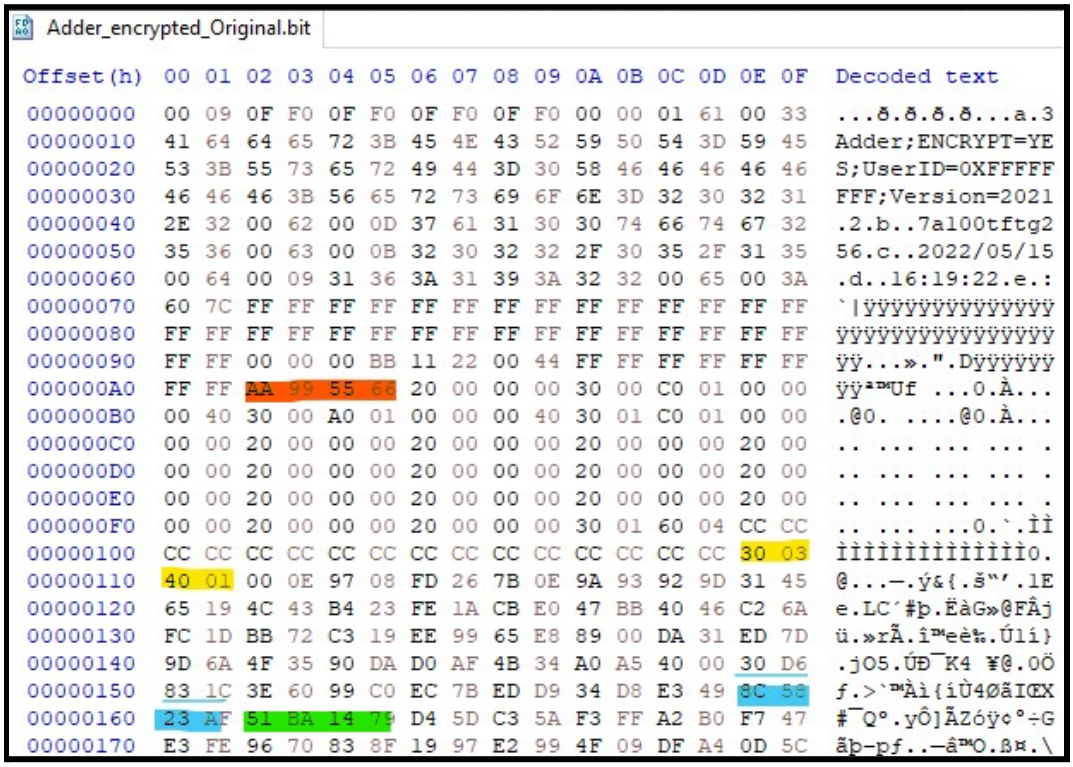


Adder_readout.bit

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	00	09	0F	F0	0F	F0	0F	F0	0F	F0	00	00	01	61	00	32	...ð.ð.ð.ð...a.2
00000010	41	64	64	65	72	3B	45	4E	43	52	59	50	54	3D	4E	4F	Adder;ENCRYPT=NO
00000020	3B	55	73	65	72	49	44	3D	30	58	46	46	46	46	46	46	;UserID=0XFFFFFF
00000030	46	46	3B	56	65	72	73	69	6F	6E	3D	32	30	32	31	2E	FF;Version=2021.
00000040	32	00	62	00	0D	37	61	31	30	30	74	66	74	67	32	35	2.b..7a100tftg25
00000050	36	00	63	00	0B	32	30	32	32	2F	30	35	2F	31	35	00	6.c..2022/05/15.
00000060	64	00	09	31	36	3A	31	30	3A	35	32	00	65	00	3A	60	d..16:10:52.e.:`
00000070	7C	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	|ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000090	FF	00	00	00	BB	11	22	00	44	FF	FF	FF	FF	FF	FF	FF	ÿ...».".Dÿÿÿÿÿÿÿÿ
000000A0	FF	AA	99	55	66	20	00	00	00	30	02	20	01	00	00	00	ÿª™Uf ...0. ....
000000B0	00	28	02	00	01	00	00	00	00	30	00	80	01	00	00	00	.(.......0.€....
000000C0	00	20	00	00	00	30	00	80	01	00	00	00	07	20	00	00	. ...0.€..... ..
000000D0	00	20	00	00	00	30	02	60	01	00	00	00	00	30	01	20	. ...0.`.....0.
000000E0	01	02	00	3F	E5	30	01	C0	01	00	00	00	00	30	01	80	...?å0.À.....0.€
000000F0	01	03	63	10	93	30	00	80	01	00	00	00	09	20	00	00	..c."0.€..... ..

Figure 5.3: Readout Bitstream

The unaltered encryption bitstream is shown in **Figure 5.4** where the IV of 128 bits of C are shown and then followed by the 'encryption start' instruction '30 03 40 01' where all words after this are encrypted.



Adder_encrypted_Original.bit

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	00	09	0F	F0	0F	F0	0F	F0	0F	F0	00	00	01	61	00	33	...ð.ð.ð.ð...a.3
00000010	41	64	64	65	72	3B	45	4E	43	52	59	50	54	3D	59	45	Adder;ENCRYPT=YE
00000020	53	3B	55	73	65	72	49	44	3D	30	58	46	46	46	46	46	S;UserID=0XFFFFF
00000030	46	46	46	3B	56	65	72	73	69	6F	6E	3D	32	30	32	31	FFF;Version=2021
00000040	2E	32	00	62	00	0D	37	61	31	30	30	74	66	74	67	32	.2.b..7a100tftg2
00000050	35	36	00	63	00	0B	32	30	32	32	2F	30	35	2F	31	35	56.c..2022/05/15
00000060	00	64	00	09	31	36	3A	31	39	3A	32	32	00	65	00	3A	.d..16:19:22.e.:
00000070	60	7C	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	`|ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000080	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000090	FF	FF	00	00	00	BB	11	22	00	44	FF	FF	FF	FF	FF	FF	ÿÿ...».".Dÿÿÿÿÿÿ
000000A0	FF	FF	AA	99	55	66	20	00	00	00	30	00	C0	01	00	00	ÿÿª™Uf ...0.À...
000000B0	00	40	30	00	A0	01	00	00	00	40	30	01	C0	01	00	00	.@0. ....@0.À...
000000C0	00	00	20	00	00	00	20	00	00	00	20	00	00	00	20	00	.. ..
000000D0	00	00	20	00	00	00	20	00	00	00	20	00	00	00	20	00	.. ..
000000E0	00	00	20	00	00	00	20	00	00	00	20	00	00	00	20	00	.. ..
000000F0	00	00	20	00	00	00	20	00	00	00	30	01	60	04	CC	CC	.. ..0.`.ÌÌ
00000100	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	CC	30	03	ÌÌÌÌÌÌÌÌÌÌÌÌÌÌ0.
00000110	40	01	00	0E	97	08	FD	26	7B	0E	9A	93	92	9D	31	45	@...—.ý&{.š"'.1E
00000120	65	19	4C	43	B4	23	FE	1A	CB	E0	47	BB	40	46	C2	6A	e.LC´#þ.ËàG»@FÂj
00000130	FC	1D	BB	72	C3	19	EE	99	65	E8	89	00	DA	31	ED	7D	ü.»rÃ.î™eè‰.Ú1í}
00000140	9D	6A	4F	35	90	DA	D0	AF	4B	34	A0	A5	40	00	30	D6	.jO5.ÚÐ¯K4 ¥@.0Ö
00000150	83	1C	3E	60	99	C0	EC	7B	ED	D9	34	D8	E3	49	8C	58	ƒ.>`™Àì{íÙ4ØãIŒX
00000160	23	AF	51	BA	14	79	D4	5D	C3	5A	F3	FF	A2	B0	F7	47	#¯Q°.yÔ]ÃZóÿ¢°÷G
00000170	E3	FE	96	70	83	8F	19	97	E2	99	4F	09	DF	A4	0D	5C	ãþ–pƒ..–â™O.ß¤.\

Figure 5.4: Unaltered Encrypted Bitstream

The 'write to WBSTAR' word is located at the hex address 0x15E (C_n) and the word that needs to be altered to leverage the CBC malleability at 0x14E (C_{n-1}). The formula to calculate the change in the preceding cipher text for the desired plaintext is shown is $C_{i-1}' = P_i$ XOR P_i' XOR C_{i-1}. Below shows the calculated alteration to make the 'write WBSTAR' register have a 'word count' of 2 instead of 1.

$$C_{i-1}' = P_i \oplus P_i' \oplus C_{i-1}$$

'30 D6 83 1F' = '30 02 00 01' ⊕ '30 02 00 02' ⊕ '30 D6 83 1C'

The altered malicious encrypted bitstream is shown in **Figure 5.5.**

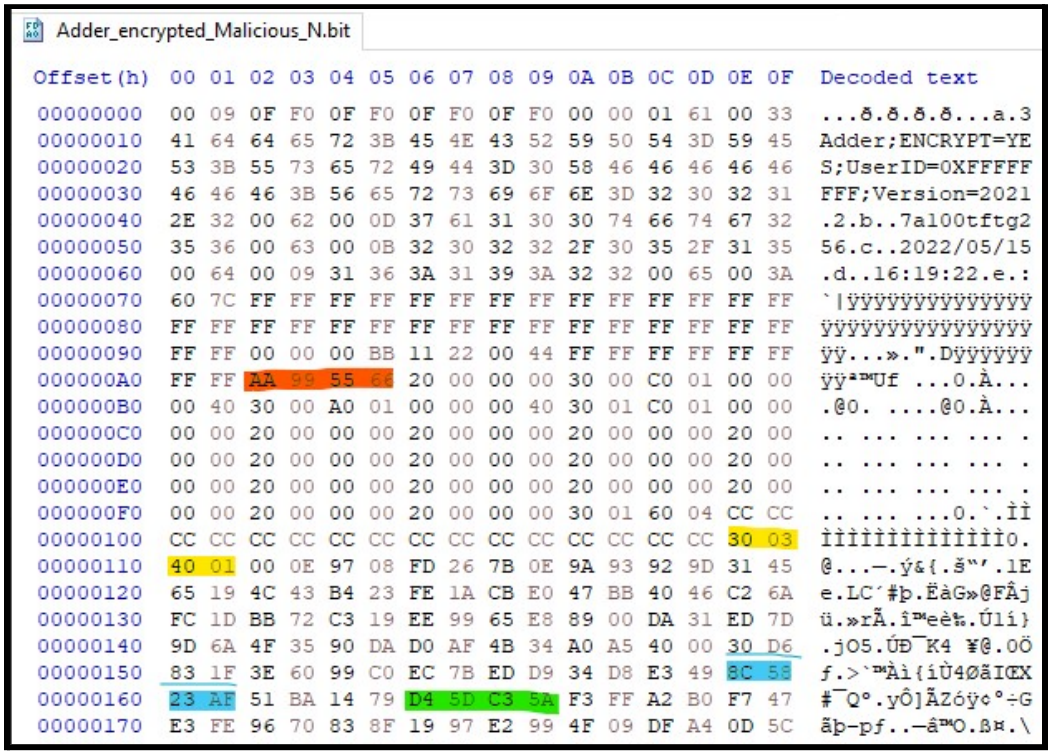


Figure 5.5: Malicious Encrypted Bitstream

The readout bitstream was applied to the FPGA.

The attack to break the authenticity was never attempted as the first attack to break confidentiality must be functioning first.

5.2 Data Analysis

Overall the data showed that we were on the right track to successfully implementing the attack. Unfortunately a roadblock was hit when it came to reading the data back from the WBSTAR register and the team could not progress to the second stage of the attack. As a result there is limited data to analyze but the methodology was followed closely as detailed previously in section 5.1.

6. CONCLUSION

6.1 Prevention

As mentioned in [1] this entire attack could be prevented if the HMAC is checked before any of the information is written to the FPGA (this has issues of time efficiently) or change the way some of the registers that are not reset can be accessed to prevent this from occurring. There is also mention of increasing the efficiency of protocols that test the security systems of FPGA at the design stage such as making the development more rigorous or making the development open source to test the system better.

6.2 Summary

In conclusion the attack was successfully implemented up until the point that reading out the WBSTAR register was required. The only problem was the inability to use debug readout to check the values of the registers, and this prevented the attack from being fully realized. Even so the team still had several takeaways from their work on the project. These include increased competency using Xilinx software and hardware, knowledge of how to program an encrypted bitstream to a Xilinx FPGA, and use of the Xilinx System Debugger. In addition the team successfully programmed bistreams to a Xilinx FPGA using both possible methods, namely ChipWhisperer and Vivado. Although many missteps occurred owing to unfamiliarity with the hardware and software and convoluted documentation, the team was able to make some measurable progress in regard to the attack and understand the steps that would be enacted after the readout of the WBSTAR register.

7. REFERENCES

[1] M. Ender, A. Moradi, and C. Paar, "The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs," in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 1803–1819. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ender

[2] *7 Series FPGAs Configuration User Guide*, Version 1.13.1. Xilinx, San Jose, CA, United States, 2018 [Online]. Available: https://docs.xilinx.com/v/u/en-US/ug470_7Series_Config

California State University, Fresno
Lyles College of Engineering
Electrical and Computer Engineering Department

TECHNICAL REPORT

Experiment Title:	Fault Injection on AES Using the Chip Whisperer Development Toolkit
Course Title:	ECE 156 - Fundamentals of Cryptography and Computer Network Security
Date Submitted:	May 18, 2022

Prepared By:	**Sections Written:**
Brandon Poorman	All
Zachary Curtis	All
Javier Gutierrez	All

INSTRUCTOR SECTION

Comments: Technical advisor: Dr. Hayssam El-Razouk

Final Grade: Team Member 1: _____

Team Member 2: _____

Team Member 3: _____

Table of Contents

Page

Title Page **1**

Table of Contents **2**

1 ABSTRACT **3**

2 LITERATURE REVIEW **3**

3 TECHNICAL BACKGROUND **3**

3.1 Fault Injection 3

3.2 AES 4

4 EXPERIMENTAL PROCEDURE **6**

4.1 Equipment Used 6

4.2 Procedure 9

4.2.1 Chip Whisperer Setup 9

4.2.2 Python Code 10

5 ANALYSIS **13**

5.1 Experimental Results 13

5.2 Data Analysis 18

5.2.1 Attack 18

5.2.2 Defense 19

6 CONCLUSION **20**

7 REFERENCES **21**

8 APPENDIX **22**

1 ABSTRACT

Cryptographic algorithms were developed in order to preserve and protect the confidentiality, integrity, and availability of information systems. However, if these systems are bypassed then the strength of the cryptographic algorithm used will not matter. One such attack utilized is the fault injection attack, which primarily has been used as an invasive attack on the hardware of system. The specific fault injection attack used will be a clock glitch attack on an AES module that will bypass the security parameters on the Chip Whisperer Development Toolkit.

2 LITERATURE REVIEW

Shylendra et al. utilize more advanced techniques than implemented below, such as a "non-parametric kernel density estimation (KDE)-based technique to develop statistical model of power side-channel leakage" [1]. Their model focuses more on using this technique to detect clock based glitch attacks while the implementation below focuses more on understanding how to implement a fault attack on AES-128 for atatck. Shylendra et al. use power side channel analysis for defense against such fault injection attacks. [1] was helpful in understanding Differential Fault Analysis (DFA) in crypto systems on the Chip Whisperer.

Dehbaoui et al. have executed a similar fault injection attack to the one executed in this project, except rather than exploiting the physical connections on a processor, they were able to perform an attack using electromagnetic pulses (EMP) [2]. This project was more so geared to find susceptibilities in existing systems using EMP as opposed to performing a very specific kind of attack. This source was useful for understanding that fault injection is really a broad spectrum of attacks, and the specific route of using the Chip Whisperer to perform an attack on a physical level is just one such attack.

3 TECHNICAL BACKGROUND

3.1 Fault Injection

Fault injection attack is a physical attack on the device to inject the system deliberately to change its intended behavior. This type of attack can bypass system security features, change system behavior to accomplish malicious intents, or extract secret information. Various ways to inject fault include , voltage glitching, clock glitching, laser injection, Electromagnetic injection. A glitch is a short-lived fault in the system. This is something that creates a sudden change that is not part of the system's normal operation. This can lead to a bricked device or if implemented properly can disrupt operations, skip instructions and/or corrupt variables. Differential Fault Analysis injects faults during the encryption process with faulty ciphertexts to extract the secret key. In order to utilize DFA, previous knowledge of the

crypto mechanism in place needs to be known. A combination of testing faulty ciphertexts and possibly correct ciphertexts provides valuable fault statistics that provides valuable information to recover the secret key and break the encryption algorithm [3]. Figure 3.1 shows a very simple fault injection attack known as clock glitching that demonstrates what happens when a fault is injected into the clock of the system. Instructions are loaded from memory into the main processing unit on the device. In the instance that the instruction that requests a user password is loaded, a glitch can be injected that makes the clock skip that instruction and the next instruction is loaded from memory and then executed instead of executing the previous instruction that was bypassed. This attack may be simple but is highly effective.

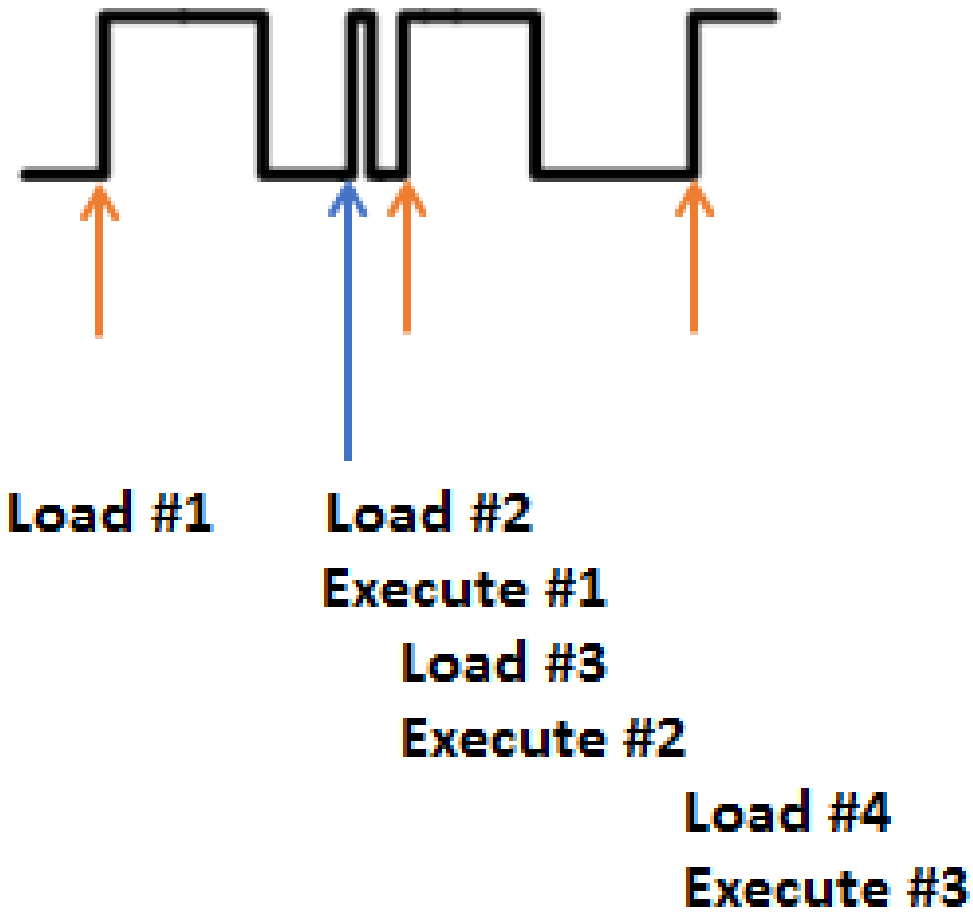


Figure 3.1: Fault Injection Clock Glitch

3.2 AES

The Advanced Encryption Standard (AES), also known as Rijndael is a symmetric cryptographic algorithm using a block cipher to preserve the confidentiality of information trans-

missions. AES is capable of using either a 128, 192, or 256 bit key sizes. AES is based on the design of a subsitution-permutation network (SPN) to provide confusion and diffusion in order to encrypt a fixed block of data. The AES standard chosen was AES-128 which uses 10 rounds for encryption and decryption. The first step in encrypting a block of plaintext into ciphertext is to do a Key Expansion to calculate round keys derived form the cipher key according to the AES key schedule. The initial round then includes an AddRoundKey which performs a bitwise XOR operation of each byte with the byte of the round key. The next 9 rounds include each of the following operations sequentially: SubBytes, ShiftRows, MixColumns, and AddRoundKey. SubBytes is a substitution step where each byte is replaced with another byte of data using a lookup table. The ShiftRows operation performs a transposition by shifting certain bytes cyclically a certain number of steps. The MixColumns operation does a mixing operation on the columns of the state. AddRoundKey does the same operation as the initial AddRoundKey. The final round of AES128 includes a SubBytes, ShiftRows, and AddRoundKey. The decryption process is done in a computationally inverse order to recover the plaintext from the ciphertext with the secret key. This process is shown in Figure 3.2.

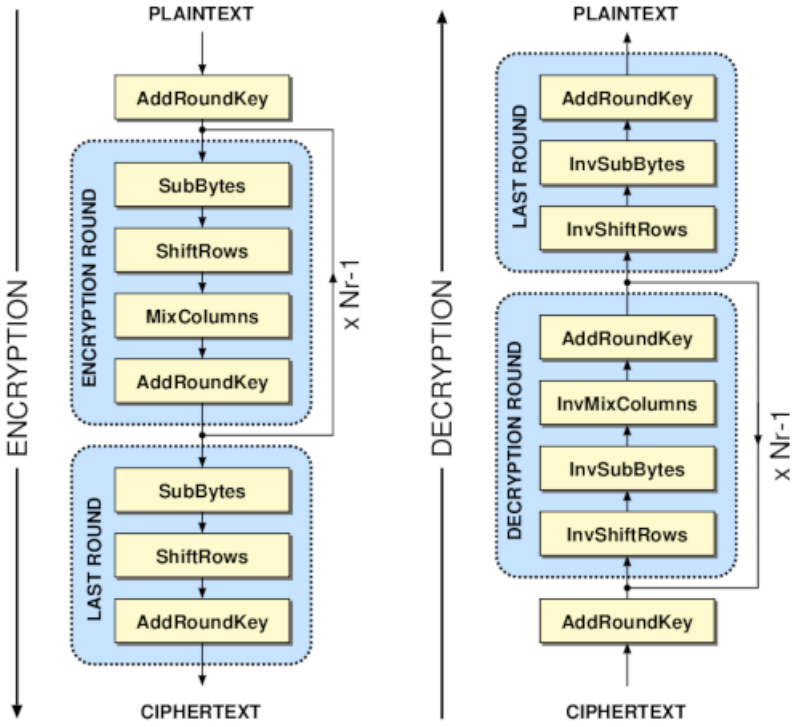


Figure 3.2: Advanced Encryption Algorithm overview

The attack performed on AES in this project consists of running through many instances of clock glitching, collecting unexpected output from the AES algorithm. Most of the time the incorrect output will be entirely different from expected, but sometimes the faulty output will only be a few bytes different from expected. In these cases, the changing output bytes

can be mapped back to a change in an input byte, which is especially easy when attacking the final round of AES, as it lacks a mix columns phase which would otherwise provide it with diffusion. Given knowledge of what the input to that round may have been which generated the faulty output, bytes can be recovered from the round key corresponding the bytes which changed from the correct output to the faulty output. With faulty outputs where only a few bytes change, the entire round key can be uncovered using this method. Once the round key is uncovered, one can work backwards through the AES key distribution algorithm to find the original AES key, therefore breaching the entire system.

4 EXPERIMENTAL PROCEDURE

4.1 Equipment Used

- Hardware

  Chip Whisperer CW308 STM32F3 Target Board

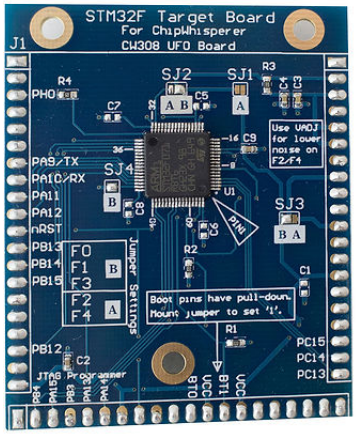


Figure 4.1: CW308 Target Board

Chip Whisperer CW1200

Figure 4.2: CW1200

Siglent Digital Oscilloscope SDS 1104X-E

Figure 4.3: Siglent Digital Oscilloscope

Personal Computer

- Software

  Python3

  VMWare Workstation Pro

  Jupyter Notebooks

4.2 Procedure

4.2.1 Chip Whisperer Setup

Downloading the Debian virtual machine (VM) is the first step. The virtual machine instance has the dependencies and drivers necessary for the Chip Whisperer hardware. The VM must be set up for a Jupyter instance to be made available, the port necessary for the host PC to access the instance through an internet browser. Once the correct IP address with port is loaded in a web browser the Jupyter instance is accessible. Creating a new ".ipynb" file allows the user to start interacting with the Chip Whisperer device.

On the hardware side the Chip Whisperer 1200 device is connected using a standard NewAE 20-pin connector pinout cable to the STM32F3 Target Board CW308. With the Jupyter instance running the initial parameters must be set. Parameters include Scopetype, Platform, and Crypto Target, for this specific application the settings used are OPENADC, CW308_STM32F3, and TINYAES128C respectively. The correct target must be selected so the ChipWhisperer Pro can flash the target board. The STM32 version is selected and the target is flashed.

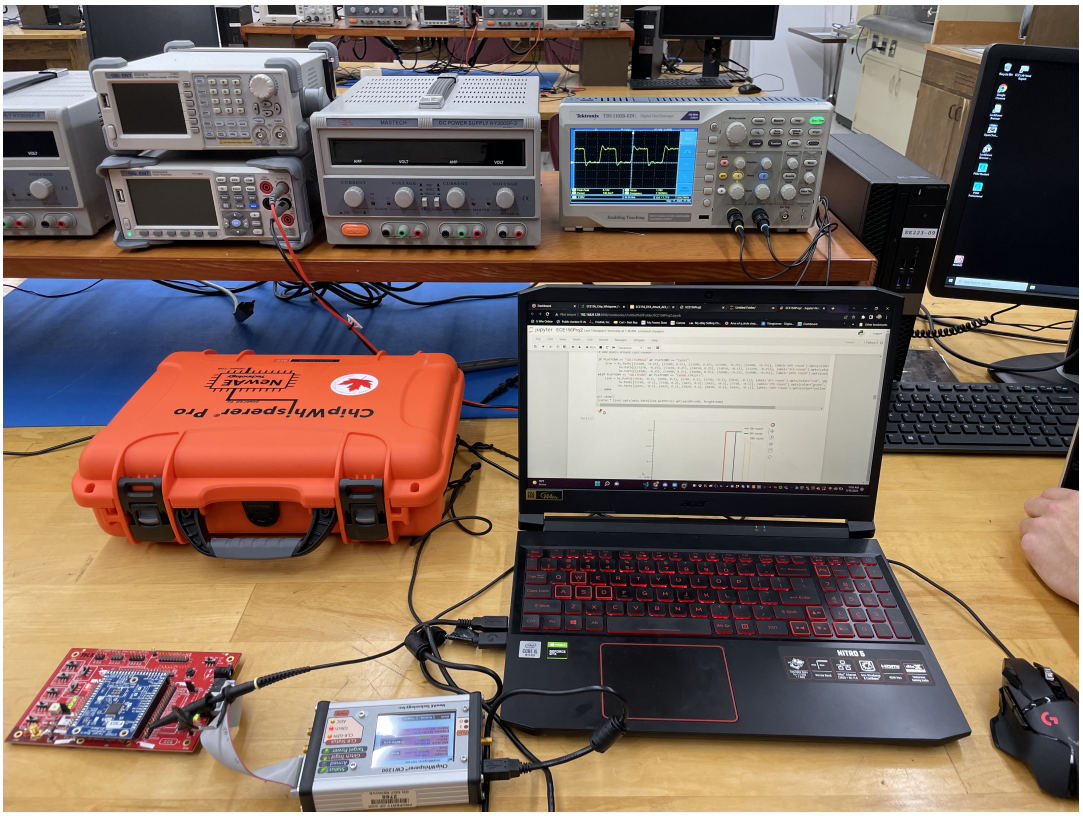


Figure 4.4: Physical Setup

4.2.2 Python Code

The Python program to run the DFA on the Chip Whisperer to break AES was already made available by New AE Technologies on their Github [3]. Some parameters had to be changed in order to utilize the CW STM32F3 Target Board and ensure the program runs with the proper dependencies in the correct directory. For a DFA attack a constant plaintext is needed. For the first execution the Chip Whisperer API is used. Full AES is captured to see which round(s) will be glitched by tuning $scope.glitch.ext_offset$ Testing the first execution outputs the golden reference output. The Plaintext, Key, and Ciphertext are also printed onto the screen. Using the holoviews library the full AES execution can be displayed. The 10 AES-128 rounds can be seen in Figure 4.4 where the 8th, 9th, and 10th rounds surrounded by a box in red, green and yellow respectively.

Once it is established that the AES is working properly the testing of the glitches can begin. Probing the Clock pin of the Chip Whisperer UFO Board the glitch can be observed in Figure 5.1 and Figure 5.2 where two clock cycles can be seen before the glitch and after the glitch. The Oscilloscope is only used to observe that the glitch is working, when implementing the actual attack it is advised to do it all with or all without the oscilloscope attached.

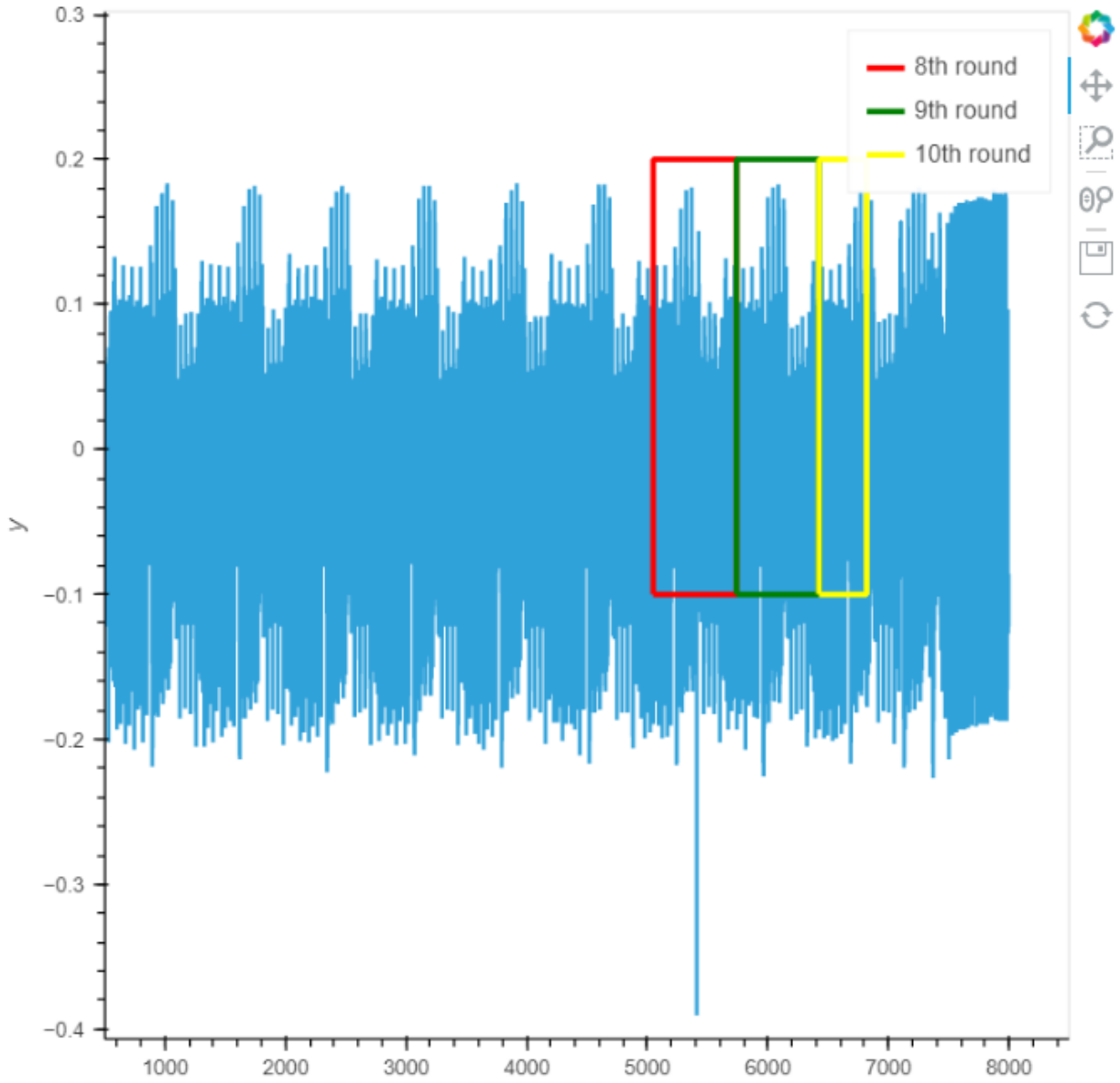


Figure 4.5: Full AES Operation

The Campaign setup can be set using GlitchController. The data will be categorized into a lot of different groups which will make understanding the logged material much easier.

Next step is to Attack the 9th round. This attack the chip whisperer tries to glitch the 9th round MixCollumns. The following operations are between these two parts of AES:

- 8th round AddRoundKey
- 9th round SubBytes
- 9th round ShiftRows

The MIN_STEP parameter was set to 25/64. The parameter is further adjusted by multiplying a constant to the MIN_STEP variable. This variable is used to set the width_range and the offset_range. These make up the glitch settings, The glitch campaign is then ran. The goal is to collect as many *interesting* outputs as possible. These *interesting* outputs are simply 16-byte output different from the reference output.

Using the *phoenixAES* library the DFA attack against the collected ciphertexts can occur. phoenixAES requires the list of *interesting* outputs and the reference output in order to try and crack the bytes of data.

In the first attack it is assumed the fault was injected between the last two MixColumn operations and we look for ciphertexts only partially corrupted. In a successfully recovery the full 10th round key would be recovered. Tuning the *width_range* can yield better results, but it can still be a form of brute force effort. In the end, a *width_range* of 2*MIN_STEP to 5*MIN_STEP was used to find as many useful faults as possible.

```
Plaintext: 000102030405060708090a0b0c0d0e0f
Key:       2b7e151628aed2a6abf7158809cf4f3c
Ciphertext:50fe67cc996d32b6da0937e99bafec60
```

Figure 4.6: Plaintext & Key used to produce the corresponding Ciphertext

5 ANALYSIS

5.1 Experimental Results

Figure 5.1: Clock of CW during normal operation

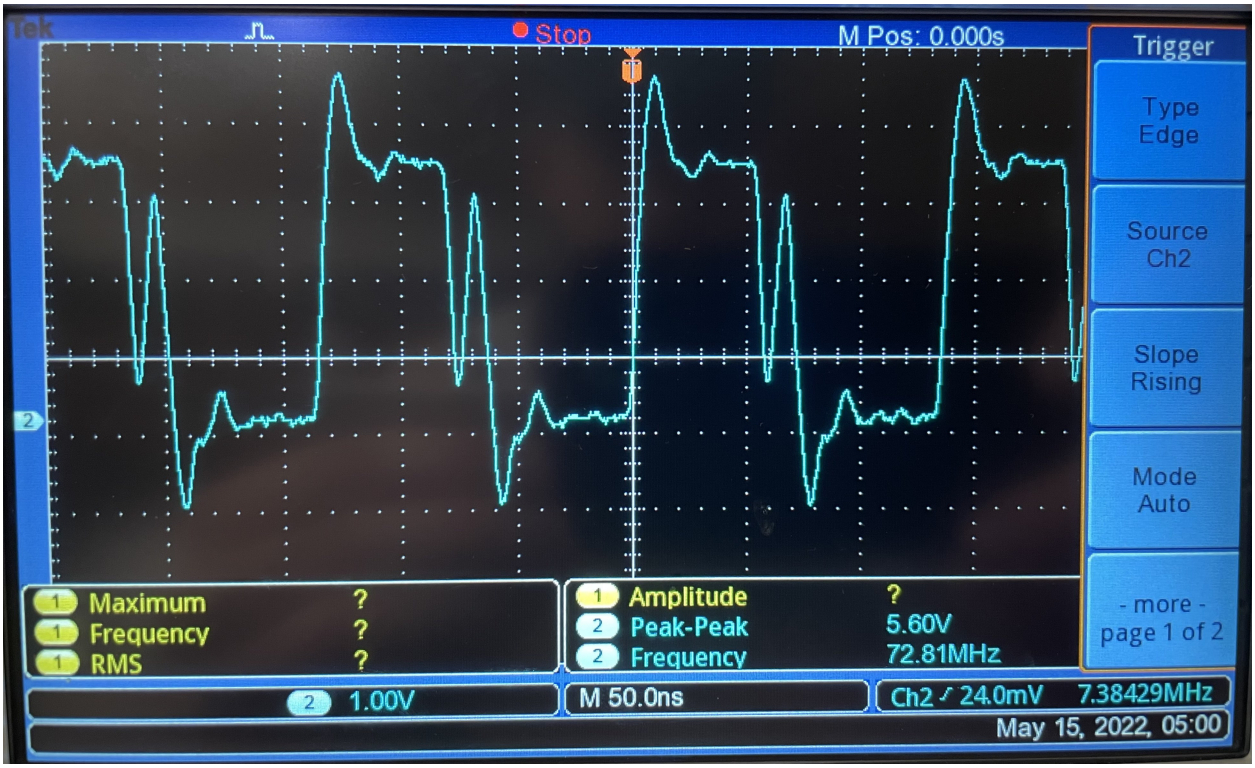


Figure 5.2: Clock of CW with clock glitch

Figure 5.1 shows the clock on the oscilloscope of the Chip Whisperer without any clock glitches and Figure 5.2 shows the clock using an oscilloscope on the Chip Whisperer with clock glitches. The clock glitch was performed on each clock cycle for 2 seconds in order to ensure that the clock glitching is performed correctly with the CW1200 on the CW Target Board. This is important because the clock glitch width during each clock cycle is what prevents the mixing of the columns during AES so that is possible to analyze possible key bytes during a specific round of AES.

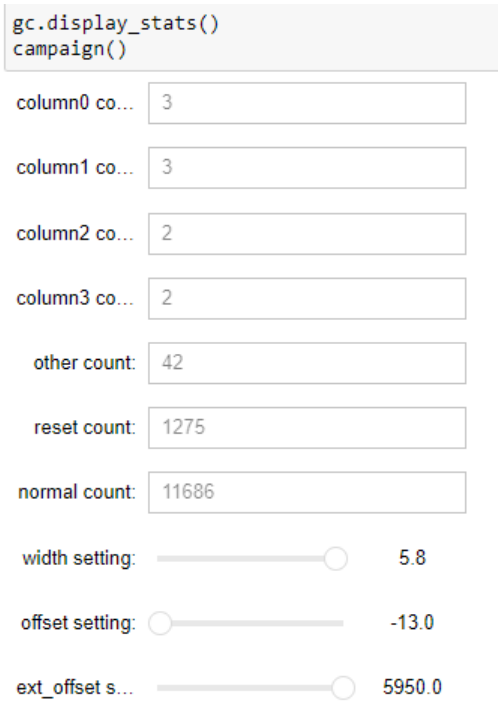


Figure 5.3: Statistics of Campaign

Figure 5.3 shows the statistics of the clock glitching campaign. Column0-Column4 represent the block of AES round keys and each column has 8 bytes of the round key in it. The total count of each column represents how many times a round key was recovered inside that column. Since there are a total of 10 occurrences of recovered round keys in all the columns that means that two different kinds of glitches produced the same corresponding round key. There were a total of 11686 iterations performed of clock glitching with DFA to recover the entire round key. Different parameters for the glitch width and offsets were tried, but failed to produce a complete recovered round key. A parameter of 1 to 6 width for the clock glitch was set to produce the results below.

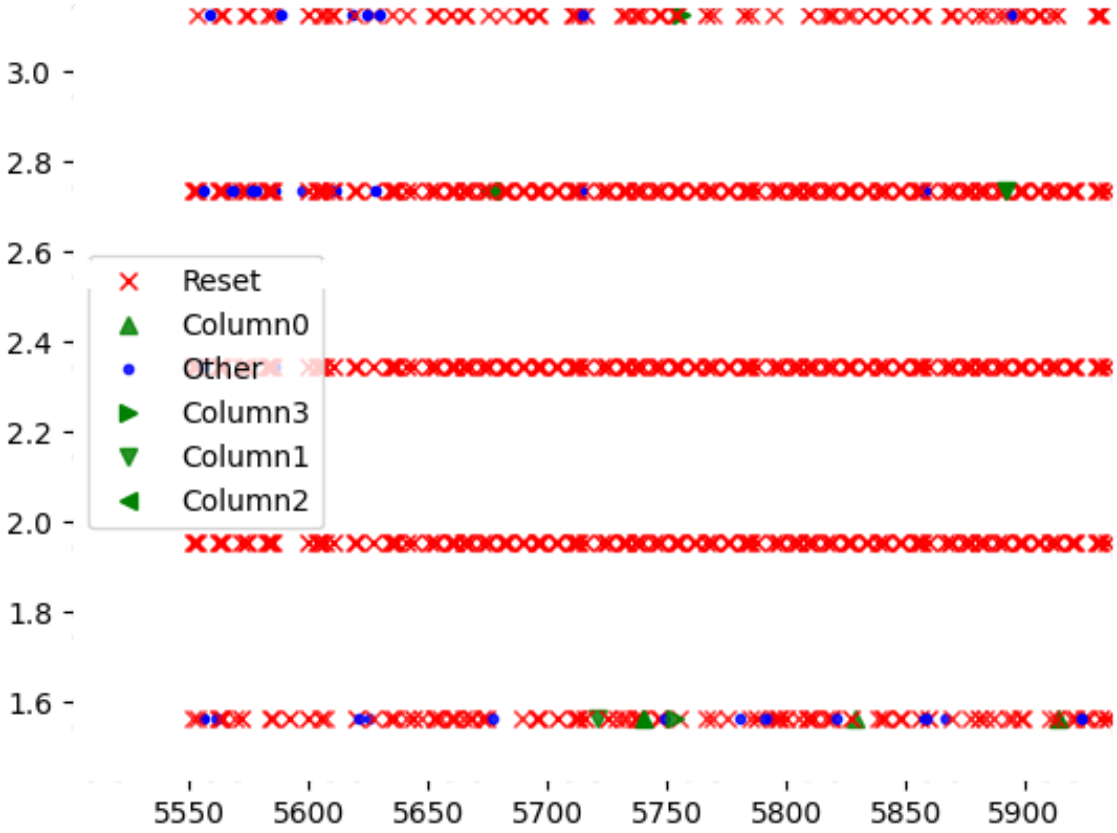


Figure 5.4: Occurrences of partial recovered Round Keys with varying Clock Glitch Widths and External Offsets

Figure 5.4 shows the occurrences of recovered round keys during the campaign. On the y-axis it shows the width of clock glitch and on the x-axis it shows the bit offset for that corresponding glitch width. Once 2 or more bytes of the round key is recovered and tested to make sure it is computationally correct according to the AES algorithm during the end of the 9th round, it uses those correct Round Key bytes to test new ciphertexts to find other bytes of the round key. If a ciphertext fails to result in a correct round key byte, it will reset the encryption and test the next ciphertext with a different clock glitch width and offset. The most common occurrence of a successful clock glitch to recover the round key was with a width of 1.5625.

```
50fe6751996d94b6daf537e976afec60: group 3
bbfe67cc996d322bda0996e99b50ec60: group 0
Round key bytes recovered:
D0............89....0C....63....
aafe67cc996d328dda09b5e99b83ec60: group 0
50fea1cc993932b6be0937e99bafecc1: group 2
50feb5cc99fd32b6d10937e99bafece2: group 2
Round key bytes recovered:
D0..F9....EE..89E1..0C....63..A6
506567cc526d32b6da0937de9baf6160: group 1
Round key bytes recovered:
D014F9..C9EE..89E1..0CC8..630CA6
504667ccee6d32b6da09372d9baf7b60: group 1
50fe6799996d44b6da0a37e9dcafec60: group 3
Round key bytes recovered:
D014F9A8C9EE2589E13F0CC8B6630CA6
Last round key #N found:
D014F9A8C9EE2589E13F0CC8B6630CA6
```

Figure 5.5: Crack of AES Round Key in Round 10

Figure 5.5 shows the ciphertext result with corresponding round key bytes recovered. The orignal ciphertext we want is 50fe67cc996d32b6da0937e99bafec60. Upon completion of recovering the entire Round key, the resulting ciphertext is shown as this with the entire recovered round key.

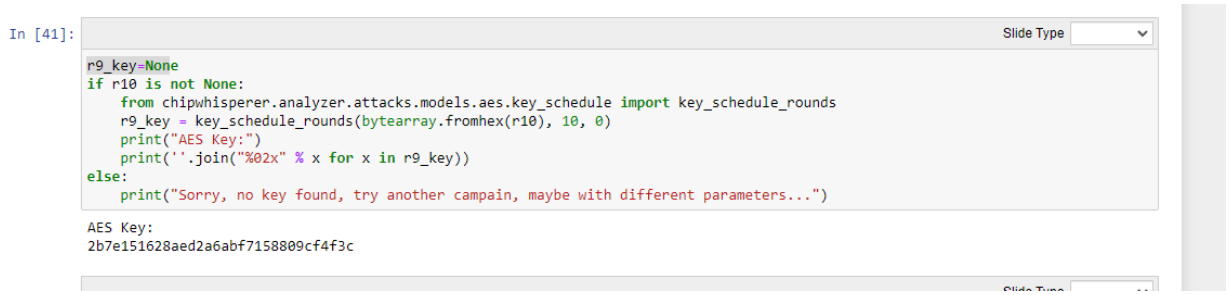


Figure 5.6: Recovered AES Key from AES Round Key based on Analysis of Key Scheduling Algorithm

Figure 5.6 shows the result of recovering the original AES Key with the AES Round Key from Round 10 with the corresponding python code implementing this. The

chipwhisperer.analyzer.attacks.models.key_schedule API is a proprietary package provided by Chip Whisperer that uses other AES utilities to recover the original AES key. If the round key from round 10 is successfully recovered using clock glitching and DFA, then it will work backwards through each round of AES-128 from Round 10 to Round 0 using the AES-128 algorithm. Since the Round Key in Round 0 is the AES-128 secret key, then it has been successfully recovered. Figure 5.6 shows that the AES key was successfully recovered and when compared to Figure 4.6 it can be verified.

5.2 Data Analysis

5.2.1 Attack

Before the AES attack had even taken place, sample clock glitching was analyzed using an oscilloscope, as shown in Figures 5.1 and 5.2. This simple analysis revealed that the clock glitching which was to be used in this project was in fact operating correctly, as the glitched waveform shown in Figure 5.2 roughly follows the expected appearance of clock glitching shown in Figure 3.1. As previously mentioned, clock glitching is when the clock input is modified slightly to execute new instructions at the wrong time, which leads to all sorts of difficult to predict glitches. This concept is crucial for this project, since it is used to glitch the AES algorithm, and guaranteeing the the clock glitching is operational is the first step to ensuring that any data can be collected by the fault injection attack performed on AES.

As shown in Figure 5.4, the attack was able to successfully create glitches in the AES algorithm. While the vast majority of them just caused the algorithm to reset, several of them ran through the full execution and generated incorrect AES ciphertext. Of these occurrences, the best ones to have for this attack are the ones where a single column of data has been modified, as represented by the green arrow icons in Figure 5.4. These instances allowed the attack to be performed as when a single column is modified, it can be more easily mapped back to the input text to that given round, which then allows for more bytes to be

discovered from the AES round key. Shown in Figure 5.5 are all of the faults which allowed for the round key to be found are shown, along with the corresponding bytes recovered with each, finally showing the AES round key at the end.

From this round key, the original AES key could be found, as shown in Figure 5.6. This works by essentially moving backwards through the key scheduler to find the round 0 key based on the round 10 key which was found. With this process, the AES key was successfully uncovered, as shown in Figures 4.6 and 5.6, where the actual key shown in Figure 4.6 perfectly matches the key found using the DFA attack shown in Figure 5.6. With this information, it is safe to say that given that the AES key is fairly static, and that an attacker has physical access to the clock and output to the AES device, this attack is viable for breaching the AES system.

5.2.2 Defense

One major reason why this attack is so effective on the final round of AES is that the final round does not have a mix columns phase, unlike all previous rounds. The mix columns phase adds a layer of diffusion to the phase, so that the output of the round is harder to trace back from the input. One solution to this vulnerability would just be to incorporate some phase which allows for diffusion into the final AES round, and a mix columns phase would be the most straightforward solution to this. With this diffusion, more ciphertext bits will change given a 1-bit change in the round input, and since it will be harder to trace back the bit that changed, it will also be harder to find the corresponding round key.

As for AES as a whole, the most straightforward preventative measure against this sort of attack is simply to switch keys often. This attack requires many iterations of fine tuning the specifications of the clock glitching, and can easily get into the thousands of iterations just to even get part of the round key, as shown in Figure FIXME. By frequently switching keys, even just every thousand uses or so, an attacker using this procedure would not be able to breach the AES key before a new key is used.

Another defense against this attack would simply be to keep the device running the AES algorithm in a physically secure location. The DFA attack performed in this project is built around clock glitching to cause improper ciphertext output, and without access to the device's clock this attack would not be possible. As touched upon in the literature review, Dehbaoui et al. have been able to successfully perform similar glitches using electromagnetic pulses [2], which is another attack which is restricted by vicinity to the hardware being attacked. In keeping the device running AES in a secure physical location, DFA attacks such as these can be mitigated.

6 CONCLUSION

The main objective of this project was to use fault injection with clock glitching to breach the AES algorithm, and this objective was successfully reached throughout the course of this project. The use of clock glitching allowed for faulty outputs to be generated, and by using differential fault analysis where the outputs were mapped to corresponding input bytes, the AES key was able to be uncovered. This attack requires the right scenario to be effective, but is definitely viable in that situation. To prevent this situation, the system can be improved in three main ways: by adding a layer of diffusion (such as mixing columns) to the last layer of AES, by generating new AES keys often, or by keeping the device running the AES algorithm in a physically secure location. Overall, this fault injection attack on AES has proven to be a threat in the right environment, but there are simple steps which can be taken to mitigate the viability of this attack which should be taken into account when implementing AES.

7 REFERENCES

[1] A. Shylendra, P. Shukla, S. Bhunia and A. R. Trivedi, "Fault Attack Detection in AES by Monitoring Power Side-Channel Statistics," 2020 21st International Symposium on Quality Electronic Design (ISQED), 2020, pp. 219-224, doi: 10.1109/ISQED48828.2020.9136981.

[2] A. Dehbaoui, J.-M. Dutertre, B. Robisson, and A. Tria, "Electromagnetic transient faults injection on a hardware and a software implementations of AES," 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, 2012.

[3] New AE Technology, "Chipwhisperer-jupyter/Lab 1_3B - DFA Attack on AES.ipynb at master newaetech/chipwhisperer-jupyter," GitHub, Apr. 20, 2022. https://github.com/newaetech/chipwhisperer-jupyter/blob/master/courses/fault201/Lab201/Lab1_3B - DFA Attack on AES.ipynb (accessed May 14, 2022).

8 APPENDIX

DFA_Attack.ipynb - Runs Entire DFA Attack on AES-128

```
SCOPETYPE = "OPENADC"
#PLATFORM = "CWLITEXMEGA"
PLATFORM = "CW308_STM32F3"
CRYPTO_TARGET = "TINYAES128C"
import sys
import os
!{sys.executable} -m pip install phoenixAES
print(os.getcwd())
os.chdir('/home/vagrant/work/projects/chipwhisperer/hardware/victims/
    firmware/simpleserial-aes/')

%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd hardware/victims/firmware/simpleserial-aes
make PLATFORM=$1 CRYPTO_TARGET=$2

import os
os.chdir('/home/vagrant/work/projects/chipwhisperer/jupyter/Setup_Scripts')
print(os.getcwd())
%run "Setup_Generic.ipynb"

os.chdir('/home/vagrant/work/projects/chipwhisperer/
    hardware/victims/firmware/simpleserial-aes/')
fw_path = "simpleserial-aes-{}.hex".format(PLATFORM)
cw.program_target(scope, prog, fw_path)

if PLATFORM == "CWLITEXMEGA":
    def reboot_flush():
        scope.io.pdic = False
        time.sleep(0.1)
        scope.io.pdic = "high_z"
        time.sleep(0.1)
        #Flush garbage too
        target.flush()
else:
    def reboot_flush():
        scope.io.nrst = False
        time.sleep(0.05)
```

```
38          scope.io.nrst = "high_z"
39          time.sleep(0.05)
40          #Flush garbage too
41          target.flush()
42
43 ktp = cw.ktp.Basic()
44 ktp.fixed_text = True
45 ktp.fixed_key = True
46 key, text = ktp.next()
47
48 scope.clock.adc_src = "clkgen_x1"
49 if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
50     scope.adc.samples = 20000
51 elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
52     scope.adc.samples = 8000
53
54 # make sure glitches are disabled (in case cells are re-run)
55 scope.io.hs2 = "clkgen"
56
57 trace = cw.capture_trace(scope, target, text, key)
58 goldciph = trace.textout
59 print("Plaintext: {}".format(text.hex()))
60 print("Key:       {}".format(key.hex()))
61 print("Ciphertext:{}".format(goldciph.hex()))
62
63 reset_target(scope)
64
65 from Crypto.Cipher import AES
66 aes = AES.new(bytes(key), AES.MODE_ECB)
67 goldciph2 = aes.encrypt(bytes(text))
68 print("Expected ciphertext:  {}".format(goldciph2.hex()))
69
70 import holoviews as hv
71 from holoviews import opts
72 import matplotlib.pyplot as plt
73
74 hv.extension('bokeh')
75 curve = hv.Curve(trace.wave).opts(width=600, height=600)
76
77 # add boxes around last rounds
78
```

```
79 if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
80     line = hv.Path([(11600, -0.25), (11600, 0.25), (13200, 0.25),
81         (13200, -0.25), (11600, -0.25)],
82         label='8th round').opts(color="red", show_legend=True) * \
83             hv.Path([(13250, -0.25), (13250, 0.25), (14850, 0.25),
84                 (14850, -0.25), (13250, -0.25)],
85                 label='9th round').opts(color="green", show_legend=True) * \
86             hv.Path([(14900, -0.25), (14900, 0.25), (16000, 0.25),
87                 (16000, -0.25), (14900, -0.25)],
88                 label='10th round').opts(color="yellow", show_legend=True)
89 elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
90     line = hv.Path([(5050, -0.1), (5050, 0.2), (5740, 0.2),
91         (5740, -0.1), (5050, -0.1)],
92         label='8th round').opts(color="red", show_legend=True) * \
93             hv.Path([(5740, -0.1), (5740, 0.2), (6425, 0.2),
94                 (6425, -0.1), (5740, -0.1)], label='9th round').opts(
95                 color="green", show_legend=True) * \
96             hv.Path([(6425, -0.1), (6425, 0.2), (6820, 0.2),
97                 (6820, -0.1), (6425, -0.1)], label='10th round').opts(
98                 color="yellow", show_legend=True)
99     pass
100
101 plt.show()
102 (curve * line).opts(opts.Path(line_width=3)).opts(width=600, height=600)
103
104 import time
105 if scope._is_husky:
106     scope.glitch.enabled = True
107
108 # These width/offset numbers are for CW-Lite/Pro; for CW-Husky,
109 #convert as per Fault 1_1:
110 def test_glitches():
111     if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
112         scope.io.hs2 = "glitch"
113         scope.glitch.clk_src = 'clkgen'
114         scope.glitch.width=3.5
115         scope.glitch.offset=34
116         scope.glitch.trigger_src='continuous'
117     elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
118         scope.io.hs2 = "glitch"
119         scope.glitch.clk_src = "clkgen"
```

```
120         scope.glitch.width = -10.15625
121         scope.glitch.offset = -39.84
122         scope.glitch.trigger_src='continuous'
123
124 def stop_test_glitches():
125     scope.glitch.trigger_src='ext_single'
126
127 test_glitches()
128 time.sleep(2)
129 stop_test_glitches()
130
131 print(scope.glitch)
132
133 # Initial glitch parameters
134 # These width/offset numbers are for CW-Lite/Pro; for CW-Husky,
135 #convert as per Fault 1_1:
136 if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
137     scope.io.hs2 = "glitch"
138     scope.glitch.clk_src = 'clkgen'
139     scope.glitch.width=3.5
140     scope.glitch.offset=34
141     scope.glitch.trigger_src='ext_single'
142     scope.glitch.ext_offset = 13000
143 elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
144     scope.io.hs2 = "glitch"
145     scope.glitch.clk_src = "clkgen"
146     scope.glitch.width = -10.15625 + 1
147     scope.glitch.offset = -39.84
148     scope.glitch.ext_offset = 5400
149     scope.glitch.repeat = 3
150     scope.glitch.trigger_src='ext_single'
151
152 # reset target
153 reset_target(scope)
154 time.sleep(0.1)
155
156 trace = cw.capture_trace(scope, target, text, key)
157
158 curve = hv.Curve(trace.wave)
159 curve *= hv.Path([(scope.glitch.ext_offset, 0.25),
160     (scope.glitch.ext_offset, -0.3)]).opts(color="red")
```

```
161 curve.opts(width=600, height=600)
162 from collections import namedtuple
163 # named tuples to make it easier to change the scope of the test
164 Range = namedtuple('Range', ['min', 'max', 'step'])
165
166 import math
167
168 # get control over logging in order to be able to
169 #mask target execution errors, which can easily happen when
170 #glitching the target!
171 import logging
172 logging.basicConfig(level=logging.WARN)
173
174 # Let's be prepared for user-provided ranges: rounding and checking
175 #consistency by ourselves.
176 def apply_ranges():
177     global width_range, width_range_steps
178     global offset_range, offset_range_steps
179     global extoffset_range, extoffset_range_steps
180     width_step_sign = width_range.step/abs(width_range.step)
181     offset_step_sign = offset_range.step/abs(offset_range.step)
182     width_range = Range(width_range.min, width_range.max,
183         round(width_range.step / MIN_STEP) * MIN_STEP)
184     offset_range = Range(offset_range.min, offset_range.max,
185         round(offset_range.step / MIN_STEP) * MIN_STEP)
186     if abs(width_range.step) < MIN_STEP:
187         step = width_step_sign*MIN_STEP
188         logging.error('width_range.step too small,
189             adjusting to {}'.format(step))
190         width_range = Range(width_range.min, width_range.max, step)
191     if abs(offset_range.step) < MIN_STEP:
192         step = offset_step_sign*MIN_STEP
193         logging.error('offset_range.step too small,
194             adjusting to {}'.format(step))
195         offset_range = Range(
196             offset_range.min, offset_range.max, step)
197     width_range_steps = math.ceil((
198         width_range.max-width_range.min)/width_range.step)
199     offset_range_steps = math.ceil((
200         offset_range.max-offset_range.min)/offset_range.step)
201     extoffset_range_steps = math.ceil((
```

```
202             extoffset_range.max-extoffset_range.min)/extoffset_range.step)
203         if width_range_steps < 0:
204             step = -width_range.step
205             logging.error('width_range.step has wrong sign,
206                 adjusting to {}'.format(step))
207             width_range = Range(width_range.min, width_range.max, step)
208             width_range_steps = -width_range_steps
209         if offset_range_steps < 0:
210             step = -offset_range.step
211             logging.error('offset_range.step has wrong sign,
212                 adjusting to {}'.format(step))
213             offset_range = Range(offset_range.min, offset_range.max, step)
214             offset_range_steps = -offset_range_steps
215         if extoffset_range_steps < 0:
216             step = -extoffset_range.step
217             logging.error('extoffset_range.step has wrong sign,
218                 adjusting to {}'.format(step))
219             extoffset_range = Range(extoffset_range.min,
220                 extoffset_range.max, step)
221             extoffset_range_steps = -extoffset_range_steps
222
223 GLITCH_RESULTS_FILEPATH='/tmp/glitch_outputs.csv'
224 TRACES_FILEPATH='/tmp/glitch_traces.npy'
225
226 GLITCH_RESULTS_FILEPATH=None
227 TRACES_FILEPATH=None
228
229 # get control over logging in order to be able to mask target execution
230 #errors, which can easily happen when glitching the target!
231 import logging
232 import chipwhisperer.common.results.glitch as glitch
233
234 # can
235 gc = glitch.GlitchController(groups=["column0", "column1", "column2",
236     "column3", "other", "reset", "normal"],
237     parameters=["width", "offset", "ext_offset"])
238
239 #check if a single column is glitched
240 def check_column_glitch(glitched_ct, gold_ct, column):
241     column_lookup = [[0, 13, 10, 7], [4, 1, 14, 11], [8, 5, 2, 15],
242         [12, 9, 6, 3]] #shift rows
```

```
243     for byte in column_lookup[column]:
244         if glitched_ct[byte] == gold_ct[byte]:
245             return False
246     return True
247
248 outputs = []
249 results = []
250 obf = []
251 def campaign():
252     # Initial glitch parameters
253     global outputs
254     global results
255     global obf
256
257     #reset results arrays
258     outputs = []
259     results = results = [['target output', 'width', 'offset',
260         'extoffset', 'column_fault']]
261     obf = []
262
263     #glitch setup
264     scope.io.hs2 = "glitch"
265     scope.glitch.clk_src = 'clkgen'
266     scope.glitch.trigger_src = 'ext_single'
267     key, text = ktp.next()
268
269     #make sure correct key is loaded on target
270     reboot_flush()
271     target.simpleserial_write('k', key)
272     target.simpleserial_wait_ack()
273
274     for glitch_setting in gc.glitch_values():
275         # set glitch settings
276         scope.glitch.offset = glitch_setting[1]
277         scope.glitch.width = glitch_setting[0]
278         scope.glitch.ext_offset = glitch_setting[2]
279
280         #do glitch
281         target.flush()
282         key, text = ktp.next()
283         logging.getLogger().setLevel(logging.ERROR)
```

```
284
285         scope.arm()
286         target.simpleserial_write('p', text)
287
288         ret = scope.capture()
289         if ret:
290             print("timeout!")
291             reboot_flush()
292             target.simpleserial_write('k', key)
293             target.simpleserial_wait_ack()
294             continue
295
296         #record output
297         output = target.simpleserial_read_witherrors('r', 16,
298             timeout=100, glitch_timeout=1) #don't care about glitchy text
299
300         #handle invalid output
301         if not output['valid']:
302             gc.add("reset", (scope.glitch.width, scope.glitch.offset,
303                 scope.glitch.ext_offset))
304             reboot_flush()
305             target.simpleserial_write('k', key)
306             target.simpleserial_wait_ack()
307             continue
308
309         data = [bytes(output['payload']).hex(), scope.glitch.width,
310                 scope.glitch.offset, scope.glitch.ext_offset]
311
312         #normal output
313         if output['payload'] == goldciph:
314             gc.add("normal", (scope.glitch.width, scope.glitch.offset,
315                 scope.glitch.ext_offset))
316             data.append(None)
317             results.append(data)
318             continue
319
320         outputs.append(output['payload'])
321
322         #check for a glitch in each column of AES
323         column_glitches = []
324         for column in range(4):
```

```
325             if check_column_glitch(output['payload'], goldciph, column):
326                 column_glitches.append(column)
327
328         #We're looking for single column glitches here
329         if len(column_glitches) == 1:
330             gc.add("column{}".format(column_glitches[0]), (scope.glitch.width
331                 scope.glitch.offset, scope.glitch.ext_offset))
332             obf.append(output['payload'])
333             data.append(column_glitches[0])
334         else:
335             gc.add("other", (scope.glitch.width, scope.glitch.offset,
336                 scope.glitch.ext_offset))
337             data.append(0xFF)
338
339         #for display in ascii table
340         results.append(data)
341
342 import holoviews as hv
343 from holoviews import opts
344 hv.extension('bokeh')
345 curve = hv.Curve(trace.wave).opts(width=600, height=600)
346
347 # add boxes around last rounds
348 if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
349     line = hv.Path([(11600, -0.25), (11600, 0.25), (13200, 0.25),
350         (13200, -0.25), (11600, -0.25)], label='8th round').opts(
351             color="red", show_legend=True) * \
352             hv.Path([(13250, -0.25), (13250, 0.25), (14850, 0.25), (14850, -
353                 (13250, -0.25)], label='9th round').opts(color="green",
354                 show_legend=True) * \
355             hv.Path([(14900, -0.25), (14900, 0.25), (16000, 0.25), (16000, -
356                 (14900, -0.25)], label='10th round').opts(color="yellow",
357                 show_legend=True)
358 elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
359     line = hv.Path([(5050, -0.1), (5050, 0.2), (5740, 0.2), (5740, -0.1),
360             (5050, -0.1)], label='8th round').opts(
361                 color="red", show_legend=True) * \
362             hv.Path([(5740, -0.1), (5740, 0.2), (6425, 0.2), (6425, -0.1),
363                 (5740, -0.1)], label='9th round').opts(color="green",
364                 show_legend=True) * \
365             hv.Path([(6425, -0.1), (6425, 0.2), (6820, 0.2), (6820, -0.1),
```

```
366                     (6425, -0.1)], label='10th round').opts(
367                     color="yellow", show_legend=True)
368     pass
369
370 (curve * line).opts(opts.Path(line_width=3)).opts(width=600, height=600)
371
372 MIN_STEP=25/64
373
374 # for scope.glitch.width:
375 # These width/offset numbers are for CW-Lite/Pro; for CW-Husky,
376 #convert as per Fault 1_1:
377 if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
378     width_range = Range(7*MIN_STEP, 8*MIN_STEP, MIN_STEP)
379 #     width_range = 7*MIN_STEP
380
381     offset_range = Range(0.2, 2.4, 0.5*MIN_STEP)
382     extoffset_range = Range(12800, 14100, 5)
383     glitch_repeat = 5
384 elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
385     gc.set_range("width", 1, 6)
386     gc.set_range("offset", -13, -13)
387     gc.set_range("ext_offset", 5550, 5950)
388     gc.set_global_step(0.4)
389     scope.glitch.repeat = 1
390
391 gc.display_stats()
392 campaign()
393
394 from terminaltables import AsciiTable
395 table = AsciiTable(results)
396 print(table.table)
397
398 %matplotlib notebook
399 gc.results.plot_2d(plotdots={"column0":"+g", "column1":"pr", "column2":"bo",
400                              "column3":"sk", "normal":None, "other":None,
401                              "reset":None},
402                    x_index=2, y_index=0)
403
404 %matplotlib notebook
405 gc.results.plot_2d(plotdots={"column0":"^g", "column1":"vg", "column2":"<g",
406                              "column3":">g", "normal":None, "other":"b.",
```

```
407                                 "reset":"rx"},
408                         x_index=2, y_index=0)
409
410 import phoenixAES
411 r10=phoenixAES.crack_bytes(outputs, goldciph, encrypt=True, verbose=2)
412 r10=phoenixAES.crack_bytes(obf, goldciph, encrypt=True, verbose=2)
413 r9_key=None
414 if r10 is not None:
415     from chipwhisperer.analyzer.attacks.models.aes.key_schedule
416             import key_schedule_rounds
417     r9_key = key_schedule_rounds(bytearray.fromhex(r10), 10, 0)
418     print("AES Key:")
419     print(''.join("%02x" % x for x in r9_key))
420 else:
421     print("Sorry, no key found, try another campain,
422             maybe with different parameters...")
423
424 os.chdir('/home/vagrant/work/projects/chipwhisperer/jupyter/Helper_Scripts')
425 print(os.getcwd())
426 import holoviews as hv
427 from holoviews import opts
428
429 %run "AES_differential_plotter.ipynb"
430 curve = None
431 if r9_key is not None:
432     ad=AesDiff(intext=text, key=r9_key, encrypt=True)
433     for c in outputs[:20]:
434         ad.add_glitch(c)
435     curve=ad.plot_diff_bits()
436     if curve:
437         curve
438         curve *= hv.Path([(0, 8), (10, 8)], label='8 bits faulted').opts(
439                     color="red", show_legend=True)
440     pass
441 curve.opts(width=600, height=600)
442
443 import holoviews as hv
444 from holoviews import opts
445 curve=None
446 if r9_key is not None:
447     curve=ad.plot_diff_bytes()
```

```
448     curve
449     curve *= hv.Path([(0, 1), (10, 1)], label='1 byte faulted').opts(
450                 color="red", show_legend=True)
451 curve.opts(width=600, height=600)
452
453 import holoviews as hv
454 from holoviews import opts
455 print(os.getcwd())
456 %run "AES_differential_plotter.ipynb"
457 curve = None
458 if r9_key is not None:
459     ad=AesDiff(intext=text, key=r9_key, encrypt=True)
460     for c in obf[:]:
461         ad.add_glitch(c)
462     curve=ad.plot_diff_bits()
463     if curve:
464         curve *= hv.Path([(0, 8), (10, 8)], label='8 bits faulted').opts(
465                         color="red", show_legend=True)
466 curve.opts(width=600, height=600)
467
468 curve=None
469 if r9_key is not None:
470     curve=ad.plot_diff_bytes()
471     curve *= hv.Path([(0, 1), (10, 1)], label='1 byte faulted').opts(color="r
472 curve.opts(width=600, height=600)
473
474 # for scope.glitch.ext_offset:
475 if PLATFORM == "CWLITEXMEGA" or PLATFORM == "CW303":
476     extoffset_range = Range(11800, 12400, 3)
477 elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
478     gc.set_range("ext_offset", 5050, 5250)
479 gc.display_stats()
480 campaign()
481
482 from terminaltables import AsciiTable
483 table = AsciiTable(results)
484 print(table.table)
485
486 outputs2=phoenixAES.convert_r8faults_bytes(outputs, goldciph, encrypt=True)
487 r10=phoenixAES.crack_bytes(outputs2, goldciph, encrypt=True, verbose=2)
488
```

```
489 if r10 is not None:
490     from chipwhisperer.analyzer.attacks.models.aes.key_schedule
491         import key_schedule_rounds
492     key = key_schedule_rounds(bytearray.fromhex(r10), 10, 0)
493     print("AES Key:")
494     print(''.join("%02x" % x for x in key))
495 else:
496     print("Sorry, no key found, try another campain,
497             maybe with different parameters...")
498
499 %run "AES_differential_plotter.ipynb"
500 curve=None
501 if key is not None:
502     ad=AesDiff(intext=text, key=key, encrypt=True)
503     for c in outputs[:100]:
504         ad.add_glitch(c)
505     curve=ad.plot_diff_bits()
506     if curve:
507         curve
508         curve *= hv.Path([(0, 8), (10, 8)], label='8 bits faulted').opts(
509                     color="red", show_legend=True)
510 curve.opts(width=600, height=600)
511
512 curve = None
513 if key is not None:
514     curve=ad.plot_diff_bytes()
515     if curve:
516         curve.opts(width=600, height=600)
517 curve
518
519 scope.dis()
520 target.dis()
521 assert key == [43, 126, 21, 22, 40, 174, 210, 166, 171, 247, 21,
522                     136, 9, 207, 79, 60], "Failed to break r8"
523 assert r9_key == [43, 126, 21, 22, 40, 174, 210, 166, 171, 247, 21,
524                     136, 9, 207, 79, 60], "Failed to break r9"
```

California State University, Fresno

Lyles College of Engineering

Electrical and Computer Engineering Department

TECHNICAL REPORT

Project Title: HMAC Algorithm Implementation based on SHA-256

Course Title: ECE156 - Fundamentals of Cryptography and Computer Network Security

Date Submitted: May 19, 2022

Prepared By:	Sections Written:
Brian Irineo (Lead)	Abstract, edited Theoretical Background, Experimental Procedure, Experimental Results and Analysis, Conclusion, Code Implementation
Jorge Lopez	Theoretical Background

INSTRUCTOR SECTION

Comments: Technical advisor: Dr. Hayssam El-Razouk

Final Grade: Team Member 1: _____

Team Member 2: _____

TABLE OF CONTENTS

TABLE OF CONTENTS **2**

1. ABSTRACT **3**

2. THEORETICAL BACKGROUND **3**

3. EXPERIMENTAL PROCEDURE **8**

3.1. Equipment Used 8

3.2 Procedure 8

3.2.1 SHA-256 Implementation 8

3.2.2 HMAC Implementation 10

4. ANALYSIS **11**

4.1 Experimental Results 11

4.1.1 SHA-256 Implementation Results 11

4.1.2 HMAC Implementation Results 11

4.2 Data Analysis 13

4.2.1 SHA-256 Implementation Analysis 13

4.2.2 HMAC Implementation Analysis 13

5. CONCLUSION **13**

7. REFERENCES **14**

1. ABSTRACT

Hash functions are a type of mathematical function that compresses information of various sizes down to a fixed size. In contrast to encryption algorithms, hash algorithms are one way. As a result, it is computationally infeasible for two inputs to ever output the same hash value. Hashing provides data integrity as part of the three pillars of cyber security, the CIA triad. Hash based message authentication codes, HMAC, are used in tandem with hashing functions in order to satisfy the authentication pillar as well. In this project, a secure hash algorithm, SHA-256, and a secret key will be applied to a string of plain text in order to generate a message tag. This generated message tag will then be compared to the output of an online HMAC generator tool in order to verify proper implementation of the HMAC algorithm on SHA-256.

Keywords: hash functions, encryption, algorithms, hash value, CIA triad, HMAC

2. THEORETICAL BACKGROUND

Hashing is the process in which an input, *M*, is converted, or "hashed" into output of fixed length, $h=H(M)$ where *h* is the output of the hashing function *H* [1]. The output of the hashed function is also known as the hash value or hash code. Cryptographic functions are hash functions with one-way and collision-free properties[1]. The one-way property states that it is impossible to retrieve the input of the hash function given the output. The collision-free property states that each input produces a unique output, or not two inputs will map to the same output. Good hash functions also have collision resistance where if given the same two digests, it's impossible to determine the input block[1].These properties make cryptographic hashing functions useful for ensuring the integrity of the message.

Message Authentication Codes (MAC) are used to authenticate the source of a message and that the message has not been altered. The formula $MAC = C(K,M)$ calculates the MAC of a given message. Where *K* is a shared secret key between the sender and receiver, *M* is the message, and *C* is the MAC function[2]. The MAC function, *C*, is typically a symmetric block cipher. In this case however, the MAC must fulfill the one-way property. The output of the MAC function is then appended to the end of the message that is being sent. The receiver then performs their own MAC on the received message using the same secret key as the sender. If the received MAC and the calculated MAC are the same then the receiver can be certain the message has not been altered and the source of the message is legitimate. It is assumed that the sender and receiver have a pre established and secure communication channel.

SHA-256 is a hashing algorithm that takes an input of any length and outputs a stream of fixed length[1]. The maximum input size of SHA-256 is 2^{64} bits and the output is a constant 256 bit block. The first step in SHA-256 is to first convert the input into a binary stream. After

converting the input stream a 1 followed by 0's are padded to the end of the bit stream until the bit stream is of length 64 bits less than the nearest multiple of 512. The second step is to append the length of the input stream. To do this, 64 bits are added to the padded input. These 64 bits contain the length of the original message. These first two steps produce a block of size that is always divisible by 512.

After preparing the input stream, the hash buffer must be initialized. In this stage, 8 registers are initialized as shown in the figure below. These values are the first 64 bits of the square roots of the first 8 prime numbers. These variables will hold the value of the hashes throughout the execution of the algorithm and eventually hold the final hash value.

```
unsigned long  H0 = 0x6a09e667;
unsigned long  H1 = 0xbb67ae85;
unsigned long  H2 = 0x3c6ef372;
unsigned long  H3 = 0xa54ff53a;
unsigned long  H4 = 0x510e527f;
unsigned long  H5 = 0x9b05688c;
unsigned long  H6 = 0x1f83d9ab;
unsigned long  H7 = 0x5be0cd19;
```

Figure 2.1: Initial Hash Values

Round Constants also need to be declared. These round constants are put into an array of 64 elements. These constants are the first 32 bits of the fraction parts of the cube roots of the first 64 prime numbers. These constants are necessary in the compression cycle calculations.

```
// SHA-256 uses the following constants as defined by NIST
// They represent the first 32 bits of the fractional parts of the cube roots of the first 64 primes
unsigned long k[64] = {
    0x428a2f98,0x71374491,0xb5c0fbcf,0xe9b5dba5,0x3956c25b,0x59f111f1,
    0x923f82a4,0xab1c5ed5,0xd807aa98,0x12835b01,0x243185be,0x550c7dc3,
    0x72be5d74,0x80deb1fe,0x9bdc06a7,0xc19bf174,0xe49b69c1,0xefbe4786,
    0x0fc19dc6,0x240ca1cc,0x2de92c6f,0x4a7484aa,0x5cb0a9dc,0x76f988da,
    0x983e5152,0xa831c66d,0xb00327c8,0xbf597fc7,0xc6e00bf3,0xd5a79147,
    0x06ca6351,0x14292967,0x27b70a85,0x2e1b2138,0x4d2c6dfc,0x53380d13,
    0x650a7354,0x766a0abb,0x81c2c92e,0x92722c85,0xa2bfe8a1,0xa81a664b,
    0xc24b8b70,0xc76c51a3,0xd192e819,0xd6990624,0xf40e3585,0x106aa070,
    0x19a4c116,0x1e376c08,0x2748774c,0x34b0bcb5,0x391c0cb3,0x4ed8aa4a,
    0x5b9cca4f,0x682e6ff3,0x748f82ee,0x78a5636f,0x84c87814,0x8cc70208,
    0x90befffa,0xa4506ceb,0xbef9a3f7,0xc67178f2
};
```

Figure 2.2: Round Constants

The final preprocessing step is to take the padded message and parse it into blocks. The blocks are of 512 bits and denoted as M^1, M^2 …. M^n . These blocks are further subdivided into 16 32-bit words known typically as M^i_n. SHA-256 requires a message schedule of 32 words. To that end, the rest need to be computed using the equation seen in Figure 2.3. Where *W* is the 32 bit word that will be created for the round constants. The sigma functions in Figure 2.4 are necessary when calculating the expansion of the message schedule. ROTR and SHR denote a bitwise rotation to the right and a bitwise shift to the right, respectively.

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

Figure 2.3: Message Schedule Generation

$$\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)$$
$$\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)$$

Figure 2.4: Sigma Function Definitions

The hashing process begins with the compression rounds. The compression rounds modify the 64 words in the message schedule that were calculated in the previous preprocessing stage. Figure 2.5 shows the steps of permutation and substitutions carried out through the hashing process for SHA-512[1]. To make this formula applicable to SHA-256, all that needs to be done is to replace the number 512 by 256. Variables *a-h* are the values of the hash buffer or updated working variables, *Ch(e,f,g*) is the choice function, *Maj(a,b,c)* is the majority function, and *T1* and *T2* are temporary words.

$$
\begin{aligned}
T_1 &= h + \mathrm{Ch}(e,\ f,\ g) + \left(\sum_1^{512} e\right) + W_t + K_t \\
T_2 &= \left(\sum_0^{512} a\right) + \mathrm{Maj}(a,\ b,\ c) \\
h &= g \\
g &= f \\
f &= e \\
e &= d + T_1 \\
d &= c \\
c &= b \\
b &= a \\
a &= T_1 + T_2
\end{aligned}
$$

Figure 2.5: Updating Working Variables [1]

The operations of $\mathbf{\Sigma}_0^{256}$, where x is equal to a, and $\mathbf{\Sigma}_1^{256}$, where x is equal to e, are defined in Figure 2.6. The definitions of the change and majority functions are shown in Figure 2.7**.** Finally, the intermediate hash values need to be calculated. This is done by adding the final values of the working variables after the 64 rounds in the compression cycle to the initial hash values as shown in Figure 2.8. The final digest/hash is produced by concatenating these final intermediate hash values.

$$
\begin{aligned}
\sum_0^{\{256\}}(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\
\sum_1^{\{256\}}(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)
\end{aligned}
$$

Figure 2.6: Upper Sigma Function Definitions [1]

$$
\begin{aligned}
\mathrm{Ch}\,(e,f,g) &= (e\ \mathrm{AND}\ f) \oplus (\mathrm{NOT}\ e\ \mathrm{AND}\ g) \\
&\quad \textit{the conditional function: If e then f else g} \\
\mathrm{Maj}\,(a,b,c) &= (a\ \mathrm{AND}\ b) \oplus (a\ \mathrm{AND}\ c) \oplus (b\ \mathrm{AND}\ c) \\
&\quad \textit{the function is true only of the majority (two or three) of the arguments are true}
\end{aligned}
$$

Figure 2.7: Change and Majority Function Definitions [1]

$$H_0^{(i)} = a + H_0^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)}$$
$$H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_7^{(i)} = h + H_7^{(i-1)}$$

Figure 2.8: Calculation of Intermediate Hash Values [1]

Hashed Message Authentication Codes utilize cryptographic hash functions to produce a MAC. The formula $HMAC=H[(K^+ \text{ XOR } opad) \,||\, H[(K^+ \text{ XOR } ipad) \,||\, M]$ [2]. Where H is a hash function and K^+ is the shared secret key padded with 0s to fit the block size of the hash function. *Opad* is equal to 0x5C and *ipad* is equal to 0x36. Figure 2.9 shows the block diagram for the HMAC algorithm.

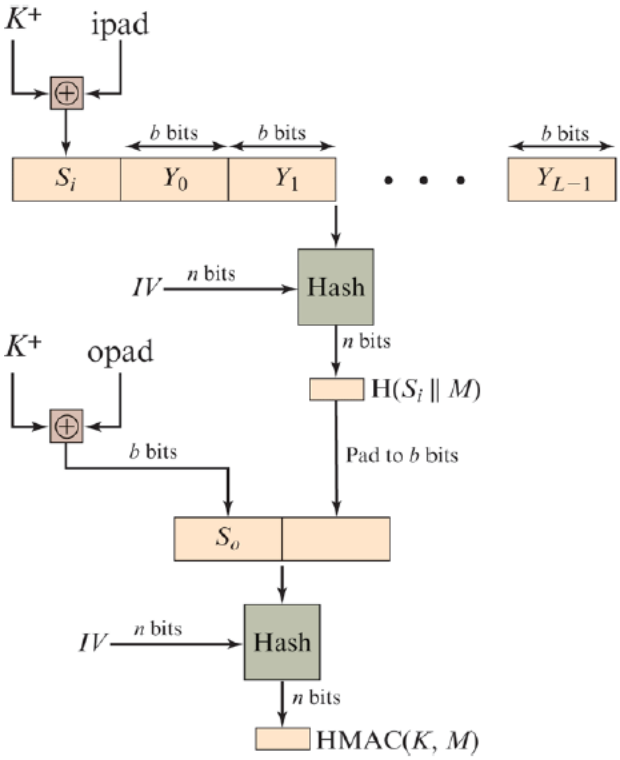


Figure 2.9: HMAC Algorithm [2]

Following the formula and the block diagram, the order of the operations of an HMAC algorithm can be observed. First, the key, *K,* is padded by appending 0's to the left of the key until the size of the padded key is the same as the block size,b, of the hash function. Then, the padded key is logically XORed with *ipad* (which is 0x36 repeated b/8 times) which produces output S_i, the inner secret key. The message blocks are then appended to the end of S_i . The hash function, *H*, is performed on this new appended stream of blocks. The outer secret key is calculated by logically XORing the padded secret key with *opad,* resulting in the outer secret key block S_0. Next, the outer S_0 is appended to the padded hashed value of S_i. and *M*. Finally, the appended stream is hashed using the chosen hash function resulting in the final HMAC digest.

3. EXPERIMENTAL PROCEDURE

3.1. Equipment Used

- ➢ Visual Studio Code

3.2 Procedure

3.2.1 SHA-256 Implementation

First SHA256 was implemented. This required the creation of the following functions: show_as_hex, convert_to_binary, pad_to_512 bits, convert_words, and compute_hash. Next the following definitions were created as necessary for round compression in the compute_hash function : ROTRIGHT(word,bits), SSIG0(x), SSIG1(x), CH(x,y,z), MAJ(x,y,z), EP0(x), and EP1(x). The definitions of these functions were taken from SHA-256 documentation and included calculations such as right rotation of bits, logical shifting, combinatorial logic, and conditional logic. The specifics were as mentioned in the background and as shown in the attached cpp files through the submission link.

The first function I created was the show_as_hex function. The purpose of this function was to take an unsigned long integer as an input and return a string of its hexadecimal representation just to make it easier to analyze outputs of the code. This utilized the bitset library. A bitset variable, bs, was initialized with the unsigned long input to a size of 32 bits. This was converted back to an unsigned long integer with the same amount of bits as bitset variable, bs. This converted long integer was inserted into a string stream filled with 0s and converted to hex. This string stream was then extracted into a string variable, which was returned by the function.

The convert_to_binary function took in a string input and output a vector of unsigned long integers where each element in the vector represented the decimal values of each ASCII character (i.e. each element represented a byte of the converted string). This was done by initializing the vector of unsigned long integers called block, then iterating the following steps over the length of the input string. A bitset variable 8 elements wide was created and called temp. The input was parsed through utilizing the c_str member function to parse through the string, setting temp to each character in that string through each iteration. Once a single character was extracted, that character was converted to its unsigned long integer representation using

bitset's to_ulong member function. The converted long integer was then appended to the block vector utilizing the push_back member function. Once all iterations over the size of the input string was complete, the final populated vector was returned.

The purpose of the pad_to_512bits function was to perform the preprocessing of the plaintext message. It's important to note that in this version, I only handled cases where the plaintext message was less than 64 characters, or rather the input message was always less than a 512 bit block. This function took in a vector of unsigned long integers where each element in the vector represented the decimal value of a single character in the plain text (the result of the convert_to_binary function) and output another vector of unsigned long integers where each element represents a single byte out of the 512 bit padded block. First, the number of 0s to append to the end of the vector was calculated by utilizing a while loop. The number of 0s, k, was increased by 1 for every iteration of a while loop where the message length + 65 + k mod 512 was not equal to 0. The message length was calculated by multiplying the input vector by 8 since each element represented a byte. Next hex value 0x80 was appended to the output vector, the same as the input vector. This handled the appended 1 at the end of the converted message as well as an additional 7 0s. Variable k was subtracted by 7 to account for this. Then the remaining 0s, k, was appended to the output vector using the push_back member function. For the final 64 bits, the message length was appended by converting the final 64 bits, 8 elements, in the vector to a string, accessing substrings, converting those substrings to long integers, then pushing them back into the output vector. The resulting output vector was a modified input vector with an appended 1, 0s, and the size of the message.

Next, the convert_words function takes in a vector of unsigned long integers where each element represents a byte in a 512 bit block and outputs another vector of unsigned long integers where each element represents a word in a 512 bit block. This was done by looping through all 64 elements in the input vector, logically shifting 4 of those elements to the left in a way that they were grouped together to a single bitset variable (this new bitset variable's value, when converted to hex represents the 8 hex values in a word), and storing this new value to an output vector.

Finally, the compute_hash function took in a vector from the convert_words function and outputs a vector of the same size. First all 64 constants, k, and the initial hash values were initialized. The initial hash values were represented as unsigned long integers labeled H0 through H7. These values were taken as according to SHA-256 documentation. The message schedule was then built up. This was done by creating a new array called W. The first 16 elements of this array were assigned the 16 elements from the input vector. For the remaining 48 elements, the expansion calculation was as follows where t was the indexed element of the W array: SSIG1(W[t-2]) + W[t-7] + SSIG0(W[t-15]) + W[t-16]. This equation was obtained from SHA-256 documentation. After every calculation, the result was modded by 2^32 in order to handle overflow. Working variables temp1, temp2, and a through h were initialized where a through h was initialized with the initial hash values H0 through H7. Once the message schedule was built, i.e. the W array was populated with 64 elements, compression began. For every element in the W array, these working variables were calculated and updated. Temp1 was calculated by summing the following where t is the iteration: h, EP1(e), CH(e,f,g), k[t], and W[t].

Temp2 was calculated by summing the following where t is the iteration: EP1(a) and MAJ(a,b,c). Then the working variables a through g were updated as follows: h was assigned the value of g, g was assigned the value of f, f was assigned the value of e, e was assigned the sum of d and temp1, e was modded by 2^32, d was assigned the value of c, c was assigned the value of b, b was assigned the value of a, a was assigned the sum of temp1 and temp2, and finally a was modded by 2^32. It was important to make these assignments in this particular order in order to avoid overwriting necessary information. Once the compression cycles were completed the intermediate hash values were calculated by taking the initial values H0 through H7, adding them with working variables a through h respectively, and modding by 2^32. These intermediate hash values were then pushed into the output vector. The output vector consisted of 8 elements where each element's value represented a byte of the final digest. Each value when converted to hex, would display a byte of the final digest.

For the main implementation, a string was obtained from the terminal. This string was input into the convert_to_binary function. The result of this function was input into the pad_to_512bits function. The result of this function was input into the convert_words function. Then finally, the result of the word converted vector was input into the compute_hash function. The resulting digest was output to the terminal both as hexadecimal values as well as the integer values of each word representing the final digest.

3.2.2 HMAC Implementation

Our implementation of HMAC was built off of the code for our SHA-256 implementation. It utilized all of the same functions from the SHA-256 implementation with the addition of 2 new functions for padding. These functions were a modified version of the pad_to_512bits function but only appended 0s on either the left side or right side of the input vector.

The compute_hash function was modified so that it took in 2 vectors each 16 elements wide. The first vector would be the appended S0 or Si block calculated as per the HMAC algorithm, and the second vector would be the padded input block. The only alterations made to this function was that the hash algorithm was first applied to the appended S0 or Si vector with the initial hash values determined by the SHA-256 documentation. The resulting intermediate hash values from this computation were then the new initial hash values for the hash computation on the padded message vector. Additionally, instead of outputting a vector of 8 elements where each element represented a word of the final digest, I included code that extracted each hexadecimal value from the intermediate hash values, and output a 64 element wide vector. This was necessary because two rounds of hashing were necessary where the result of the first hashing needed to be padded. Because the padding functions required an input vector where each element represented a byte, the previous output where the vector represented a word no longer sufficed.

For the main implementation, first the ipad and opad variables were initialized. It was previously stated that the value of ipad was 0x36 repeated over 512 bits, and opad was 0x5c repeated over 512 bits. Ipad and opad were created as vectors where each element represented a byte in the 512 bit block.Then, a string for the secret key was initialized with "procrastination". An input from the terminal was also included for the message string.

The secret key and message string were both converted to binary and padded with 0s on the left as necessary. The padded secret key vector was then logical XORed element by element with Ipad in order to generate the 512 bit, Si block, a vector with 16 elements. This Si vector was input along with the padded message vector into the compute_hash function. The result of this hash function was then padded with 0s on the right. Similar to how the Si vector was created, the S0 vector was created by logical XORing the padded secret key vector with the opad vector. The final hashing computation took this opad vector and the padded previous hash.

4. ANALYSIS

4.1 Experimental Results

4.1.1 SHA-256 Implementation Results

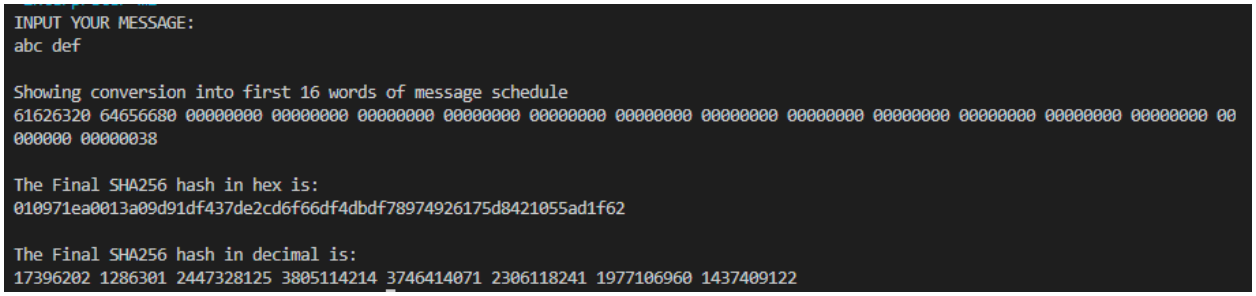


Figure 4.1: Terminal Output of SHA-256 Implementation

4.1.2 HMAC Implementation Results

```
Showing ipad as words in hex
36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36
363636 36363636

Showing opad as words in hex
5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c
5c5c5c 5c5c5c5c

INPUT YOUR MESSAGE:
abc

Showing padded secretkey as words in hex
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 0070726f 63726173 74
696e61 74696f6e

NOW XORing padded secret key with IPAD represented as words in hex format
36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36363636 36464459 55445745 42
5f5857 425f5958


Showing conversion of message into first 16 words of message schedule after padding
61626380 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00
000000 00000018

The SHA256 hash in hex is:
77f235f3af1833d74ddf714a0ff69babc75c02663d92a9949b71f7ae3c82291f

The first hash computation in decimal is:
119 242 53 243 175 24 51 215 77 223 113 74 15 246 155 171 199 92 2 102 61 146 169 148 155 113 247 174 60 130 41 31



NOW XORing padded secret key with OPAD represented as words in hex format
5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c5c5c5c 5c2c2e33 3f2e3d2f 28
35323d 28353332

Showing conversion of padded previous digest into first 16 words of message schedule after padding
77f235f3 af1833d7 4ddf714a 0ff69bab c75c0266 3d92a994 9b71f7ae 3c82291f 00000000 00000000 00000000 00000000 00000000 00000000 00
000000 00000000

The SHA256 hash in hex is:
1df854485dd40d46fae1e006600f1cc145aa7a3647977a534d525c5114eb1bea

The Final SHA256 hash in decimal is:
29 248 84 72 93 212 13 70 250 225 224 6 96 15 28 193 69 170 122 54 71 151 122 83 77 82 92 81 20 235 27 234
```

Figure 4.2: Terminal Output of HMAC Implementation

4.2 Data Analysis

4.2.1 SHA-256 Implementation Analysis

The implementation of SHA-256 was successful. If the input message was the string “abc def”, the terminal output the following digest:
010971ea0013a09d91df437de2cd6f66df4dbdf78974926175d8421055ad1f62

According to online SHA-256 calculators, this was correct. The only issue with this implementation is that it was coded to only operate on messages that were less than 56 characters long. The reason for this was because the space for the last 8 characters was reserved for the length of the message. In order to modify the code such that this implementation would work on messages larger than 56 characters, the pad_to_512bit function would have to be modified so that it would parse through the input vector, extract every 512 bit (in the context of this code, 64 elements) and store each group of 64 elements as a vector in a 2-dimensional array. The final element of the initial input vector is the vector that would have to be padded then pushed into the 2-dimensional vector. The output of this function would then be a 2-dimensional vector, where each element is a vector that represents a 512 bit block. The compute_hash function would then have to be altered to account for this by accessing each block in the 2-dimensional array, calculating the hash, then saving the resulting intermediate hash value as the initial hash value for the computations on the next block in the 2-dimensional array.

4.2.2 HMAC Implementation Analysis

Our implementation of HMAC did not produce the correct results. There were multiple outputs to the terminal in order to determine where the error occurred. The padding of the input block was correct. The generation of the padded ipad, opad, and padded secret key was also correct. The XOR operations for both the inner key and outer key also proved to be correct. From the modified compute_hash function, we included snippets to output what the resulting hash was in hex in a vector as words (unchanged from the SHA-256 implementation) as well as the new output which was a vector created by extracting a single byte out of every word. From the terminal output and showing the padded version of the previous hash, the way each byte was extracted from the digest was correct as well. As a result, the underlying issue must have been in the implementation of the hash computation, even though it was expanded on a correctly working SHA-256 computation function.

5. CONCLUSION

In conclusion, we were able to implement SHA-256 albeit only on a single block. We were also able to validate the correct computations and block expansion of the appended inner and outer secret keys. However, we were unable to output the correct digest. The problem lies within the hash computation on multiple blocks. Although we found the logic and sequence of calculations to be correct, there may be an issue with variables being overwritten or being filled with trash due to poor coding practices.

7. REFERENCES

[1] W. Stallings, “11.5 Secure Hash Algorithm: SHA,” in *Cryptography and network security: Principles and practice*, Hoboken, NJ: Pearson Education, Inc., 2019, pp. 338–358.

[2] W. Stallings, “12.5 Macs Based on Hash Functions: Hmac,” in *Cryptography and network security: Principles and practice*, Hoboken, NJ: Pearson Education, Inc., 2019, pp. 377–380.

ECE 156 - Fundamentals of Cryptography and Computer Network Security

DPA Attack on AES Labs

Dr. Hayssam El-Razouk
Electrical and Computer Engineering Department
California State University, Fresno

Acknowledgement: this lab is based on the open-source tutorial material available through the following link:

https://rtfm.newae.com/Capture/ChipWhisperer-Pro/

1. Introduction to Jupyter

Running Tutorials

Jupyter notebooks are documents that combine Python code, rich text, and additional code (such as Bash), organized into executable blocks. Notebooks allow users to run blocks out of order and multiple times, giving greater control over what's being run than a Python script. To run these notebooks, you'll need to install the Jupyter Notebook application (see Installing ChipWhisperer), which also includes file browser and text editor capabilities.

Below is an example of a Python block. First select the block by clicking on it. A green or blue border should appear around the block, indicating that the block is selected. Next, hit the `>| Run` button on the toolbar at the top of your screen. The next block should now be highlighted.

```
i = 0
```

State is kept between Python blocks. Try running the next block multiple times to see how the printed number changes:

```
i += 1
print(i)
```

Notebooks also allow special blocks to be run. Two special blocks used throughout the ChipWhisperer tutorials are `bash` blocks, which allow bash commands to be run (note: if you're not using a bash-like terminal to run Jupyter, these blocks won't work for you):

```
%%bash
echo "Test bash"
i=0
for j in *; do let "i=i+1"; done
echo "there are" $i "files/folders in this directory"
```

And `run` blocks, which run other notebooks:

```
%run "Helper_Scripts/Test_Run.ipynb"
```

Code blocks can also be edited by clicking within the code portion of the block (the grey area). You can tell if you're in edit mode by the presence of a green border around the block. If you're not in edit mode, the border will be blue instead.

Completing Tutorials

ChipWhisperer tutorials are designed to be run from start to finish, with few changes needed. One change you will likely need to make is to the `Paramater` block. These appear as the first code block in every tutorial and are used to allow tutorials to work with multiple hardware setups. An example `Parameter` block is shown below:

```
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEARM'
CRYPTO_TARGET = 'NONE'
```

These are typically used as follows in the tutorial:

- `SCOPETYPE` - Indicates the capture hardware to use. If you have a ChipWhisperer Lite or Pro, use `'OPENADC'`. If you have a ChipWhisperer Nano, use `'CWNANO'`.
- `PLATFORM` - Selects the target we're attacking. As of ChipWhisperer v5.1.0, tutorials only support at most `'CWLITEARM'` (STM32F3 target), `'CWLITEXMEGA'` (XMEGA target), and `'CWNANO'` (Nano STM32F0 target).
- `CRYPTO_TARGET` - Selects the cryptographic library to use on the target. If unsure, use `'AVRCRYPTOLIB'` for the XMEGA target and `'TINYAES128C'` otherwise. Tutorials that don't have this block or use `'NONE'` don't require a crypto library.

The parameter blocks should always be the first one run during a tutorial.

ECE 156 - Fundamentals of Cryptography and Computer Network Security

DPA Attack on AES Labs

Dr. Hayssam El-Razouk
Electrical and Computer Engineering Department
California State University, Fresno

Acknowledgement: this lab is based on the open-source tutorial material available through the following link:

https://rtfm.newae.com/Capture/ChipWhisperer-Pro/

3. Timing Analysis with Power for Password Bypass

Supported setups:

SCOPES:

- OPENADC
- CWNANO

PLATFORMS:

- CWLITEARM
- CWLITEXMEGA
- CWNANO

This tutorial will introduce you to breaking devices by determining when a device is performing certain operations. It will use a simple password check, and demonstrate how to perform a basic power analysis.

Note this is not a prerequisite to the tutorial on breaking AES. You can skip this tutorial if you wish to go ahead with the AES tutorial.

```
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEXMEGA'
CRYPTO_TARGET = 'NONE'
```

Firmware

Like before, we'll need to setup our `PLATFORM`, then build the firmware:

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../../hardware/victims/firmware/basic-passwdcheck
make PLATFORM=$1 CRYPTO_TARGET=$2
```

Setup

Setup is the same as usual, except this time we'll be capturing 2000 traces.

```
%run "Helper_Scripts/Setup_Generic.ipynb"
```

```
fw_path = '../../hardware/victims/firmware/basic-passwdcheck/basic-passwdcheck-{}.hex'.
```

```
cw.program_target(scope, prog, fw_path)
```

Communicating With The Target

As was mentioned at the beginning of the tutorial, the firmware we loaded onto the target implements a basic password check. After getting a `'\n'` terminated password, the target checks it and enters an infinite loop, so before communicating with it, we'll need to reset it.

We'll be doing this a lot, so we'll define a function that resets the target (this function is also available by running "Helper_Scripts/Setup.ipynb" as we did above):

```
import time
def reset_target(scope):
    if PLATFORM == "CW303" or PLATFORM == "CWLITEXMEGA":
        scope.io.pdic = 'low'
        time.sleep(0.05)
        scope.io.pdic = 'high'
        time.sleep(0.05)
    else:
        scope.io.nrst = 'low'
        time.sleep(0.05)
        scope.io.nrst = 'high'
        time.sleep(0.05)
```

The target sends some text to us upon starting. After running the block below, you should see some text appear.

NOTE The text may appear cutoff, accompanied by a message about data loss. This means that the buffer used to store serial data (128 bytes) from the target is full. This isn't an issue here, since the text is just aesthetic, but keep this in mind if you want to do large transfers of serial data using ChipWhisperer.

```
ret = ""
reset_target(scope)

num_char = target.in_waiting()
while num_char > 0:
    ret += target.read(timeout=10)
    time.sleep(0.05)
    num_char = target.in_waiting()

print(ret)
```

Now we can send the target a password:

```
target.flush()
target.write("h0px3\n")
#target.write("aaaaa\n")
```

And get the response. We sent it the right password (hopx3), so you should see "Access granted, Welcome!":

```
print(target.read(timeout=100))
```

tip

In real systems, you may often know one of the passwords, which is sufficient to investigate the password checking routines as we will do. You also normally have an ability to reset passwords to default. While the reset procedure would erase any data you care about, the attacker will be able to use this 'sacrificial' device to learn about possible vulnerabilities. So the assumption that we have access to the password is really just saying we have access to a password, and will use that knowledge to break the system in general.

Recording Traces

Now that we can communicate with our super-secure system, our next goal is to get a power trace while the target is running. To do this, we'll arm the scope just before we send our password attempt, then record the trace as we've done before.

```
if PLATFORM == "CWNANO":
    scope.adc.samples = 800
else:
    scope.adc.samples = 2000
```

```
ret = ""
reset_target(scope)
num_char = target.in_waiting()
while num_char > 0:
    ret += target.read(timeout=10)
    time.sleep(0.01)
    num_char = target.in_waiting()

print(ret)
scope.arm()
target.flush()
target.write("h0px3\n")
ret = scope.capture()
if ret:
    print('Timeout happened during acquisition')

trace = scope.get_last_trace()
resp = ""
num_char = target.in_waiting()
while num_char > 0:
    resp += target.read(timeout=10)
    time.sleep(0.01)
    num_char = target.in_waiting()
print(resp)
```

Now that we have a trace, we'll plot it:

```
%matplotlib notebook
import matplotlib.pylab as plt

plt.plot(trace)
```

Timing Analysis

Now that we can capture traces, we can begin planning our attack. First we'll make a function to guess a password and return a power trace, since we'll be repeating those steps a lot:

```
def   cap_pass_trace(pass_guess):
    ret = ""
    reset_target(scope)
    num_char = target.in_waiting()
    while num_char > 0:
        ret += target.read(num_char, 10)
        time.sleep(0.01)
        num_char = target.in_waiting()

    scope.arm()
    target.flush()
    target.write(pass_guess)
    ret = scope.capture()
    if ret:
        print('Timeout happened during acquisition')

    trace = scope.get_last_trace()
    return trace
```

Next, we'll try two different passwords and see if the power traces differ by length. We'll then plot both traces on the same figure (with the first in red and the second in blue).

```
%matplotlib notebook
import matplotlib.pylab as plt

# start block

#trace_correct = cap_pass_trace("h0\n")
#trace_wrong   = cap_pass_trace("h9\n")

# end block

#enable the following block and disable the one above - start block

trace_correct = cap_pass_trace("h\n")
trace_wrong   = cap_pass_trace("h0\n")
count = 0
for i in trace_wrong:
    count = count + 1
    if i <= -0.5:
        print(count, i)
count = 0
for i in trace_correct:
    count = count + 1
    if i <= -0.5:
        print(count, i)

# end block
c
```

```
plt.plot(trace_wrong, 'r')
plt.plot(trace_correct, 'g')
```

You should see both traces start and end similarly, but differ elsewhere. If you look closely, you should see that the blue trace looks a lot like the red trace but shifted later in time. We'll use this timing difference to break the password!

Edit the above block to try different passwords and see how it changes for different lengths and number of correct characters.

Go back to the original guesses (`"\n"` and `"h\n"`) and find some distinct spikes that get shifted in time. Your target may differ, but in my case, there were some distinct spikes of about -0.25 at 229 in red and 265 in blue. The plot is interactive, so you can zoom in and move around using the buttons on the right side of the plot. Record their locations, value, and the difference in location (in my case, 229, 265, -0.25, and 36).

Using distinct peaks may not always work. Instead of distinct peaks you can use where the two traces start to diverge. At the beginning the power traces are similar even though the number of characters correct is different. However, there is a point where they start to become significantly different. If you can find this spot you can use this spot to do the timing analysis instead. The difference in location should be the same as when using distinct peaks.

An even easier way to see this is to simply plot *every* possible first letter. If there is some sort of timing attack, we should see everything take one path, except for a single outlier. Let's try plotting a few traces this way and hope we get an interesting outlier.

```
%matplotlib notebook
import matplotlib.pylab as plt

trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
for c in trylist:
    next_pass = c + "\n"
    trace = cap_pass_trace(next_pass)
    plt.plot(trace)
```

Attacking a Single Letter

Now that we've located a distinctive timing difference, we can start building our attack. We'll start with a single letter, since that will quickly give us some feedback on the attack.

The plan for the attack is simple: keep guessing letters until we no longer see the distinctive spike in the original location. To do this, we'll create a loop that:

- Figures out our next guess
- Does the capture and records the trace
- Checks if sample 229 is larger than -0.2 (replace with appropriate values)

To make things a little easier for later, we'll make a function that will return whether our spike is (guess incorrect) or isn't (guess correct) in the right location:

```
def checkpass(trace, i):
    if PLATFORM == "CWNANO":
        #There's a bit of jitter
        return (trace[228 + 11*i] < 3 and trace[227 + 11*i] < 0.3)
    elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
        return trace[249 + 40*i] > -0.25
        #return trace[884 + 40 * i] > -0.2
    elif PLATFORM == "CW303" or PLATFORM == "CWLITEXMEGA":
        return trace[162 + 40 * i] > -0.4
```

The below loop finds the first correct character, prints it, then ends. You should see "Success: h" after a while.

```
trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
password = ""
for c in trylist:
    next_pass = password + c + "\n"
    trace = cap_pass_trace(next_pass)

    if checkpass(trace, 0):
        print("Success: " + c)
        break
```

Note that you will likely need to change the values in `checkpass`, or simply define your own `checkpass()` function.

Attacking the Full Password

Now that we can guess a single character, attacking the rest is easy; we just need to repeat the process in another loop, move the check point (this is the change is location you recorded earlier), and update our guess with the new correct letter.

After updating the below script and running it, you should see parts of the password printed out as each letter is found.

```
trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
password = ""
for i in range(5):
    for c in trylist:
        next_pass = password + c + "\n"
        trace = cap_pass_trace(next_pass)
        if checkpass(trace, i):
            password += c
            print("Success, pass now {}".format(password))
            break
```

That's it! You should have successfully cracked a password using the timing attack. Some notes on this method:

- The target device has a finite start-up time, which slows down the attack. If you wish, remove some of the `printf()`'s from the target code, recompile and reprogram, and see how quickly you can do this attack.

- The current script doesn't look for the "WELCOME" message when the password is OK. That is an extension that allows it to crack any size password.
- If there were a lock-out on a wrong password, the system would ignore it, as it resets the target after every attempt.

Attack via SAD Match

As you've seen, for simple timing attacks like this. However, looking at traces in this way has a number of disadvantages:

- If the offset between the correct and incorrect guesses changed, we would have to manually incorporate that into the attack
- If the offsets change (say changing the optimization level), we have to find the offsets between the traces again
- The attack is fairly finnicky and may require some trial and error

Luckily, there's a few ways of finding time offsets between parts of traces that are more reliable and require much less manual work. For this section, we will be focusing on using a Sum of Absolute Difference (SAD) match, which is a method of measuring the difference between two signals to find if a target trace is or isn't shifted in time from a reference trace.

The SAD match itself is a simple calculation:

$$\sum_{j=0}^{J} |t_{ref,j} - t_{target,j}|$$

Where $t_{ref,j}$ is a single point of the reference trace, $t_{target,j}$ is a single point of the target trace, and j is the point along the trace we're taking the difference at. This is performed along the length of the reference trace, J. Put simply, we're subtracting the two traces, taking the absolute value, then adding these absolute differences. If this value is low, the traces are very similar. If the value is high, they're very different. Our strategy will be as follows:

- Capture a reference trace and find a unique portion
- Guess another character and slide the reference along the trace, calculating the SAD at each offset until we find one below a certain threshold
- Repeat this until we find a character with a different offset than the reference - this is the correct character
- Repeat with the rest of the characters until we've broken the password

First, we'll need to make a function to calculate the offset based on a SAD match, which takes in the reference trace, the target trace, and a threshold to match at. Try to write this function yourself, but if you get stuck, `PA_SPA_1_answers.py` has a working version. For maximum compatability with the tutorial, make this function return `None` if it doesn't find a match.

```
#def find_offset_SAD(ref, target_trace, threshold):
from PA_SPA_1_answers import find_offset_SAD
```

Let's test our function on some simple lists:

```
a = list(range(50))
b = [5, 6, 7]
offset = find_offset_SAD(b, a, 0.01)
assert offset == 5, "Incorrect offset from SAD function"

b = [8.5, 9, 10]
offset = find_offset_SAD(b, a, 1)
assert offset == 8, "Incorrect offset from SAD function with nonzero threshold"
```

Choosing a Reference

Now that we have a working function to find an offset using a SAD match, let's capture a reference trace, as well as a trace with a correct guess:

```
%matplotlib notebook
import matplotlib.pyplot as plt
ref_trace = cap_pass_trace("a\n")
correct_trace = cap_pass_trace("h\n")
plt.plot(ref_trace, 'r')
plt.plot(correct_trace - 0.5, 'g')
```

We've got two requirements for the reference:

1. It should be fairly unique. For example, the section you select shouldn't match any later section in the same trace
2. It must be a portion that shifts in time. For example, the very beginning of the power trace probably doesn't shift in time here, but later sections definitely should.

For example, in the following trace, the black section doesn't shift in time, making it a poor choice. Meanwhile, the purple section does shift in time, but isn't unique: it's very likely that it will match with another part of the trace. The blue selection, however, is both unique and moves in time.

Finally, selecting our reference:

```
%matplotlib notebook
if PLATFORM == "CWNANO":
    #There's a bit of jitter
    ref = ref_trace[300:500]
elif PLATFORM == "CWLITEARM" or PLATFORM == "CW308_STM32F3":
    ref = ref_trace[770:1000]
elif PLATFORM == "CW303" or PLATFORM == "CWLITEXMEGA":
    ref = ref_trace[110:265]

plt.plot(ref)
```

As a quick test, we should try the SAD match on another incorrect guess as well as the original reference trace: they should both have the same offset here. This also gives us a good opportunity to find a good threshold. If the threshold is too high, the trace will match early. If it's too low, it won't match at all:

```
trace = cap_pass_trace("b\n")
offset1 = find_offset_SAD(ref, ref_trace, 5)
offset2 = find_offset_SAD(ref, trace, 5)

assert offset1 == offset2, "Mismatched offsets for incorrect guesses. Adjust threshold
```

Now let's see if we can guess a correct character using this technique. If the offset is higher than what we get from the reference trace, we know we've got a correct character:

```
original_offset = find_offset_SAD(ref, ref_trace, 5)
trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
for c in trylist:
    next_pass = c + "\n"
    trace = cap_pass_trace(next_pass)
    offset = find_offset_SAD(ref, trace, 5)
    if offset is None:
        print("Threshold likely too low")
        break
    elif offset == 0:
        print("Threshold likely too high")
        break
    if offset > offset1:
        print(c)
        break
```

Extending this to the rest of the password isn't too hard either. Simple iterate through all the characters. Again try writing this code yourself. One caveat here is that this technique will likely fail on the last password, since the part of the trace we're checking won't occur if we guess the password correctly. To work around this, we can simply check the response to see if we've got the whole password correct using `target.read()`. Again, if you're really stuck, a working guess function can be found in `PA_SPA_1_answers.py`

```
#def guess_password_SAD(cap_pass_trace, find_offset, ref, original_offset, threshold, t
from PA_SPA_1_answers import guess_password_SAD
password = guess_password_SAD(cap_pass_trace, find_offset_SAD, ref, offset1, 5, target)
```

With that, you should now have successfully broken the password again! Keep in mind that this is just one application of the SAD match: for example, it's quite useful for resynchronizing traces if random jitter is used as a countermeasure against side channel attacks, as is the case in tutorial `PA_CPA_3`. It is also available as a powerful trigger for the CW1200 Pro - in a typical target, we might not have access to a simple IO pin to trigger off of.

Keep in mind that this is just one of many possible ways to perform this attack - a similar one is performed in PA_Multi_1 using correlation instead of SAD.

Conclusion

This tutorial has demonstrated the use of the power side-channel for performing timing attacks. Two techniques were used to break a password with a timing vulnerability.

```
scope.dis()
target.dis()
```

Tests

```
assert (password == "h0px3\n"), "Failed to break password, got {}.\nIf on Nano, may nee
```

California State University, Fresno
Lyles College of Engineering
Electrical and Computer Engineering Department

TECHNICAL REPORT

Work Title: PUF-Based Sensor in Temperature Monitoring System

Course Title: ECE186B - Senior Design II

Date Submitted: May 17, 2022

Prepared By:	Sections Written:
Trevor Ridley	**Trevor Ridley**
Julius Andrade	**Julius Andrade**
Cristian Luna	**Cristian Luna**

INSTRUCTOR SECTION

Comments: Technical advisor: Dr. Hayssam El-Razouk

Final Grade: Team Member 1: _____

Team Member 2: _____

Team Member 3: _____

TABLE OF CONTENTS

Section	Page
Title Page ……	1
Table of Contents ……	2
Progress Log ……	3
1. Abstract ……..……	5
2. Literature Review ……...……	5
3. Theoretical Background ……...……	7
3.1 Required Hardware and Software……	7
3.2 Technical Specifications ……	8
3.3 Background ……..	12
3.3.1 SRAM Physically Unclonable Functions ……	13
3.3.2 PUFs as Temperature Sensors ……	13
3.3.3 Kill Switches in Electric Vehicles ……	13
3.3.4 Controlling the 23K256 SRAM ……	13
3.3.5 HC-05 Bluetooth Communication ...……	13
3.3.6 Arduino MEGA 2560 …….	14
3.3.7 Altera DE2-115 ……..	14
3.3.8 TXS0108E Logic Level Converter ……..	14
4. Hardware and Software Unit Testing …...……	15
4.1 Arduino Mega 2560 Serial UART Test ……	15
4.2 Arduino Mega 2560 DHT11 Temperature Sensor Test ……	16
4.3 Altera DE2-115 Serial UART Test ……	18
4.4 GeeekPi TXS0108E Logic Converter Test ……..……	21
4.5 HC-05 Bluetooth Module Test with Android ...……	22
4.6 Arduino Mega 2560 Ethernet 2 Shield Webserver Test ……	23
4.7 Arduino Mega 2560 Vibration Motor Test ……	24
5. Methodology ……	26
5.1 Design Overview ……	26
5.2 Hardware Alternatives …….	26
5.3 Top-level Design Block Diagram ……	27
5.4 Design Algorithms ....……	28
6. Implementation Plan ……	32
7. Implementation ……	33
7.1 Serial CRP transfer between Arduino and DE2-115 ……	33
7.2 Connecting DE2-115 to Serial SRAM using SPI ……	34

7.3 Performing bit masking to increase SRAM startup-value reliability … 37
7.4 Voltage Variance Noise Mitigation ……..…… 40
7.5 Implementing SRAM-PUF Response Generation ...…… 41
7.6 Creating an Arduino SD-Card Memory Database to Store CRPs …… 42
7.7 PUF Enrollment Phase at Room Temperature …… 45
7.8 PUF Correlation Phase ……. 45
7.9 PUF Sensing Phase …… 48
7.10 Output to Vibration Motors ……. 49
7.11 Output to Digital Speaker ……. 49
7.12 Android App Bluetooth Warning …… 49
7.13 HC-05 Bluetooth Module Secure Connection …… 49

8. Conclusion …… 50

9. Contributions …… 51

10. References …… 52

11. Appendix A: Arduino Enrollment Code …… 55

12. Appendix B: Arduino Correlation Code …… 60

13. Appendix C: Arduino Sensing Code …… 66

14. Appendix D: DE2-115 C Code …… 73

15. Appendix E: C++ Hex to Text Code …… 91

PROGRESS LOG

Date	Section #	Student Name	Summary of Changes
2/15/2022	3.3	Trevor Ridley	Changed background info from DRAM PUF to SRAM PUF
2/15/2022	7.1	Trevor Ridley	Documented serial CRP transfer implementation
2/15/2022	7.2	Trevor Ridley	Documented connecting DE2-115 to SPI SRAM
2/24/2022	7.3	Trevor Ridley	Documented bit masking methodology
2/24/2022	7.4	Trevor Ridley	Documented voltage regulator reliability analysis
2/26/2022	7.5	Trevor Ridley	Documented SRAM-PUF response generation
3/6/2022	7.6	Trevor Ridley	Documented Arduino SD-Card database implementation
2/20/2022	4.6	Julius Andrade	Documented setting up a web server got it to run on Arduino
2/20/2022	N/A	Julius Andrade	Did research on setting a web server and some HTML
2/21/2022	4.6	Julius Andrade	Created HTML web server for unit testing
2/21/2022	4.6	Julius Andrade	Testing server capabilities unit test
2/22/2022	4.6	Julius Andrade	Tested multiple attacks to take down server unit test
3/1/2022	N/A	Julius Andrade	Changed the server to a database on SD to read CRPs
3/8/2022	N/A	Julius Andrade	Debugging reading and masking CRPs
3/16/2022	N/A	Julius Andrade	Started testing SRAM with heat for CRP reliability
3/22/2022	N/A	Julius Andrade	Tried new source of heat to test for reliability
3/29/2022	N/A	Julius Andrade	Tested and gathered data for how temp affected reliability
4/5/2022	7.8	Trevor Ridley	PUF Correlation Phase Documentation
4/8/2022	N/A	Julius Andrade	Ran heat test again to gather better data
4/19/2022	N/A	Cristian Luna	Record demonstration and UART Serial Test Result
4/19/2022	N/A	Cristian Luna	Helped debugging Sensing Phase
4/19/2022	N/A	Cristian Luna	Circuit setup and implementation of vibration motor
4/19/2022	N/A	Cristian Luna	Handled Arduino and Eclipse software during correlation data collection

3/16/2022	N/A	Cristian Luna	Took handwritten notes detailing temperature-correlation relationship
4/5/2022	7.9	Trevor Ridley	PUF Sensing Phase Documentation
4/25/2022	7.10	Cristian Luna	Output to Vibration Motor Documentation
4/19/2022	N/A	Julius Andrade	Tested emergency triggers (vibrating motor, bluetooth, and speaker)
4/25/2022	7.11	Cristian Luna	Output to Digital Speaker Documentation
4/25/2022	7.12	Julius Andrade	Bluetooth warning message to android phone documentation
4/26/2022	N/A	Julius Andrade	Tested temp values and recorded values with thermometer final test trial
4/26/2022	N/A	Cristian Luna	Recorded final test trial
4/26/2022	N/A	Trevor Ridley	Used heat gun in final test trial
5/10/2022	7.13	Trevor Ridley	Bluetooth Secure Connection Configuration Documentation
5/10/2022	N/A	Trevor Ridley	Added DE2-115, Arduino, V-Converter, and Bluetooth to Background
5/10/2022	N/A	Trevor Ridley	Updated Table of Contents
5/10/2022	3.1-3.8	Trevor Ridley	Added Subsections to Background
5/10/2022	5.2	Trevor Ridley	Added Hardware Alternatives Section to Justify Use of FPGA
5/10/2022	8	Trevor Ridley	Updated Conclusion Section
5/3/2022	N/A	Trevor Ridley	Setup AT command mode for bluetooth module
5/3/2022	N/A	Julius Andrade	Wrote bluetooth AT command code setup for encryption
5/3/2022	N/A	Cristian Luna	Helped set up bluetooth hardware circuit
2/15-5/3/2022	N/A	Cristian Luna	Set up Quartus Prime Project and Eclipse Code on Laptop
2/15-5/3/2022	N/A	Cristian Luna	Setup Arduino IDE C Code on Laptop
4/26/2022	N/A	Cristian Luna	Setup vibration motor circuit for final testing
4/19/2022	N/A	Julius Andrade	Modified arduino code for final testing for emergency triggers
4/19/2022	N/A	Trevor Ridley	Fixed errors preventing correct Arduino code functionality
3/19/2022	N/A	Trevor Ridley	Bitmask generation and processing time optimization

1. ABSTRACT

As smart vehicle technology progresses, security continues to be a concern for vehicles and their sensors. Two major cybersecurity risks for sensors include spoofing, where an attacker poses as one of the vehicle's sensors, and physical attacks, where an attacker gains physical access to the sensor. Physically Unclonable Functions (PUFs) are a recent technology that protects against these attacks in addition to possessing potential for use in sensing temperature. This protection results from the unique fingerprint of the PUF, while the temperature sensing capability relies on the relationship between the PUF's temperature and its reliability. Taking advantage of these characteristics of PUFs allows a more secure temperature sensor to be realized. In this work we implement a PUF temperature sensor utilizing an SRAM PUF. The designed system utilizes PUF temperature readings to trigger safety features such as warning the driver via bluetooth, vibrating the steering wheel, and turning off the vehicle at a threshold lithium ion vehicle battery temperature. The expected result is that the system will be more resistant to spoofing and physical attacks while still providing the functionality expected of temperature sensors. This work is significant because it provides a novel application of SRAM PUF temperature sensors in an electric vehicle temperature monitoring system.

2. LITERATURE REVIEW

Networking of smart devices, commonly known as the Internet of Things (IoT), supports a broad range of applications in both industrial and commercial sectors [1]. Within these smart devices, sensors are implemented which are important components in the IoT that help a wide spectrum of applications from healthcare to monitoring critical infrastructure [1]. IoT devices are often resource constrained in terms of cost, computational capacity and power. The desire to keep the devices low cost and long lasting, the appropriate hardware protection logic and power consumption overheads are counterproductive. In this case, a Physical Unclonable Function (PUF) sensor where the PUF itself acts as a sensor or is integrated with a sensor will sense the physical measurements from any transducer capable of converting a physical to a voltage signal. PUF is a lightweight hardware security primitive which is increasingly employed to serve as hardware trust anchors of constrained devices. The PUF response unreliability is mainly induced by environmental parameter shifts, such as supply voltage and temperature. The capability of the PUF sensor to sense an electrical signal, a voltage, leads to a potential universal PUF sensor architecture when other Physical Quantities can be converted to a voltage through a transducer [1].

The authors of [2] propose a "new low-cost physical unclonable function (PUF) based temperature sensor for secure remote temperature sensing." The focus of [2] seems to be chiefly on security, since challenge response pairs are established at different temperatures for the purpose of authentication. [2] states that the design "protects the integrity of the sensed information by preventing falsified sensor data and masquerade sensing nodes." The PUF used for this implementation was a ring oscillator PUF.

PUFs have the capability to sense environmental changes such as temperature. [3] proposes a general implementation DRAM PUF-based temperature sensor for use in “off-the-shelf commodity devices”. In addition, [3] provides the results of testing the “feasibility” of the proposed DRAM PUF-based sensor. [3] also states that Arbiter and Ring Oscillator PUFs require a dedicated circuit to implement, and that due to the difficulty of finding such circuits in typical devices, interest is generally directed more towards DRAM and SRAM PUFS.

As IoTs continue to develop, so do the attacks exploiting their various vulnerabilities. [4] PUFs complement IoT by providing an added layer of security. The work done in [4] proposes “a novel weak PUF design using thermistor temperature sensors.”[4] The authors state that they “believe that creating a PUF from components that are already commonly found in cyber-physical systems could improve the viability of integrating PUFs with these devices.” Components that are especially common, such as DRAM and SRAM, would seem particularly promising given this assertion.

Work has been done to utilize the PUF unreliability resulting from environmental variables (such as temperature) to increase PUF security [5]. Knowing that a PUF challenge-response pair changes at a given temperature, that new challenge-response pair was considered valid, but only at that temperature [5]. As a result, every challenge response pair is evaluated not just on its surface correctness, but also based on the current temperature of the PUF [5]. The fact that the physical characteristic fingerprint of the PUF changes is thereby made into an advantage [5]. In a general sense this work focused on improving the security of PUF-based sensors [5].

The thesis in [6] offers a detailed overview of the effects of temperature variation on asynchronous PUFs implemented on FPGA. The work also details the general concepts behind various types of PUFs. The work includes a testing methodology with a significant range of temperatures. The work states, “To determine the robustness, the proposed Asynchronous PUF design is validated by testing at 4 different locations on 13 FPGAs at different temperatures: room temperature, 0 C, 20 C, 45 C, and 70 C.”

The work in [7] provides a design for a DRAM PUF that is “intrinsically reconfigurable”. This work is primarily focused on security but contains helpful information on the inverse relationship between PUF reliability and temperature. The work in [7] states that “the number of bit-flips at lower authentication temperatures is substantially smaller than that at higher temperatures (50◦C and 60◦C).” The work in [7] also states that “Higher temperatures result in an exponential rise in the number of bit-flips, which can potentially hinder the authentication process.” While such unreliability is a concern for security applications, it is necessary in order to implement a PUF temperature sensor.

Source [8] provides information about ECU's, electronic control units used in modern electric cars. Based on the information provided it is clear that the control units make use of SRAM, and so it is feasible that they could make use of a PUF-sensor for temperature measurements.

In [9] the use of PUFs in ECU's was proposed for the purpose of secure device authentication. This work demonstrates the applicability of PUFs for security purposes in modern vehicles. This work does not explicitly mention adding temperature sensing capabilities to these PUFs, but it seems a reasonable secondary use for the PUFs alongside security.

The typical lithium ion battery used in modern electric cars has an acceptable temperature range that includes temperatures as high as approximately 60 degrees Celsius [12]. Temperatures higher than this can lead to performance degradation and thermal runaway [12].

SRAM start-up behavior is addressed at length in [14]. Simulation is performed on an SRAM PUF to test its susceptibility to different environmental parameters such as temperature and voltage. One key finding is that voltage variation is the parameter with the highest impact on SRAM PUF start-up bits.

One important parameter when utilizing SRAM start-up values in a design is data retention time, and the work done in [21] provides some useful information regarding six transistor SRAM retention time. The work includes a graph depicting varying retention times and their corresponding temperatures. According to the graph one can expect a retention time of around 1 millisecond at around 25 Celsius.

The subsequent sections of this document will describe the theoretical background needed to understand the work, the methodology used, and the conclusions drawn.

3. THEORETICAL BACKGROUND

3.1 Required Hardware and Software

Hardware:

- Arduino MEGA 2560 **$17.49**
  Alternative: Raspberry PI 4 Model B (used as computer/server) **$61.80**
- Arduino Ethernet Shield 2 **$24.49**
  Alternative: RS485 CAN HAT for Raspberry Pi 4B (for connecting FPGA and R-PI) **$16.69**
- Mini Vibration Motors (for vibrating steering wheel) **$12.99**
- L298N Motor Driver Controller **$10.69**
- DHT11 Arduino Temperature Sensor **$8.99**
- GeeekPi TXS0108E logic level converter **$8.99**
- DSD TECH HC-05 Bluetooth Module **$9.99**
- 23K256 Serial SRAM **$1.20**
- Altera DE2-115 FPGA Board (Already Owned)

- Phone for testing text warnings (Already Owned)
- Laser Thermometer (Already Owned)
- Heat gun (Already Owned)

Software:

- Arduino IDE
- Android Studio IDE
- Quartus Prime Lite 17.1 with Nios 2 Software Build tools for Eclipse

Justification:

- For the purposes of running a server and performing simple computations the arduino is adequate. Its 256KB flash memory is sufficient storage for a CRP database.
- The DE2-115 FPGA includes built-in DRAM, making it a good candidate for implementing a DRAM PUF. It also includes general purpose I/O pins for use communicating with the Arduino and the RO-PUF.
- The vibration motors are small but 10,000RPM should be powerful enough to vibrate a steering wheel sufficiently.
- The Laser Thermometer is fairly accurate and we will also use a temperature sensor connected directly to the FPGA for taking accurate temperature measurements in the correlation phase.

3.2 Technical Specifications

Arduino MEGA 2560:

- 256KB Flash Memory
- 5V operating voltage
- 54 digital I/O pins (including some for UART comms)
- An image of the Arduino MEGA 2560 is shown below in **Figure 3.1**.

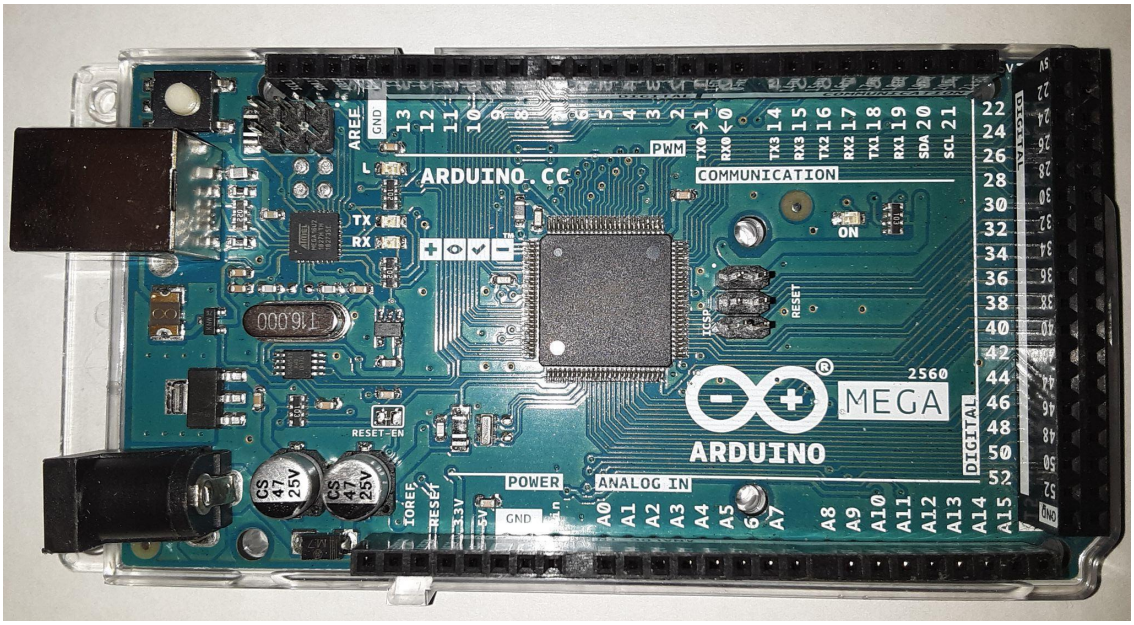


Figure 3.1: Arduino MEGA 2560

Arduino Ethernet Shield 2:

- Ethernet Controller: W5500 with internal 32K buffer
- 5V operating voltage
- 10/100MB connection speed
- Connects with Arduino via SPI port
- An image of the Arduino Ethernet Shield 2 is shown below in **Figure 3.2**.

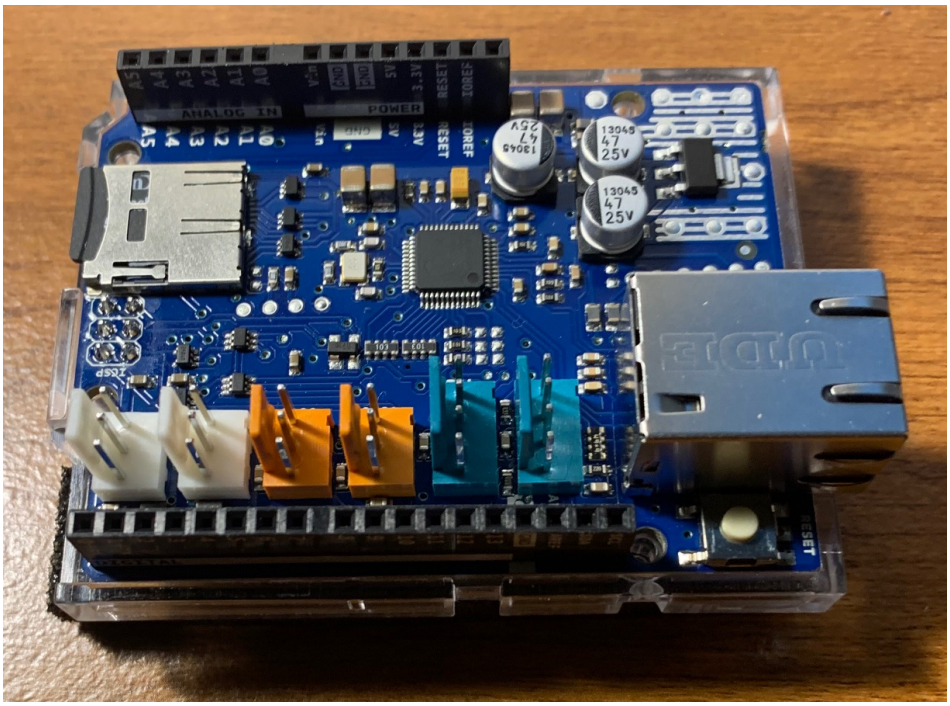


Figure 3.2: Arduino Ethernet Shield 2

Altera DE2-115:

- Two 64MB DRAM
- 2MB SRAM
- 2 Gigabit Ethernet PHY with Rj45 connectors
- 40-pin Expansion header with diode protection
- 3.3V operating voltage
- An image of the Altera DE2-115 is shown on the following page in **Figure 3.3**.

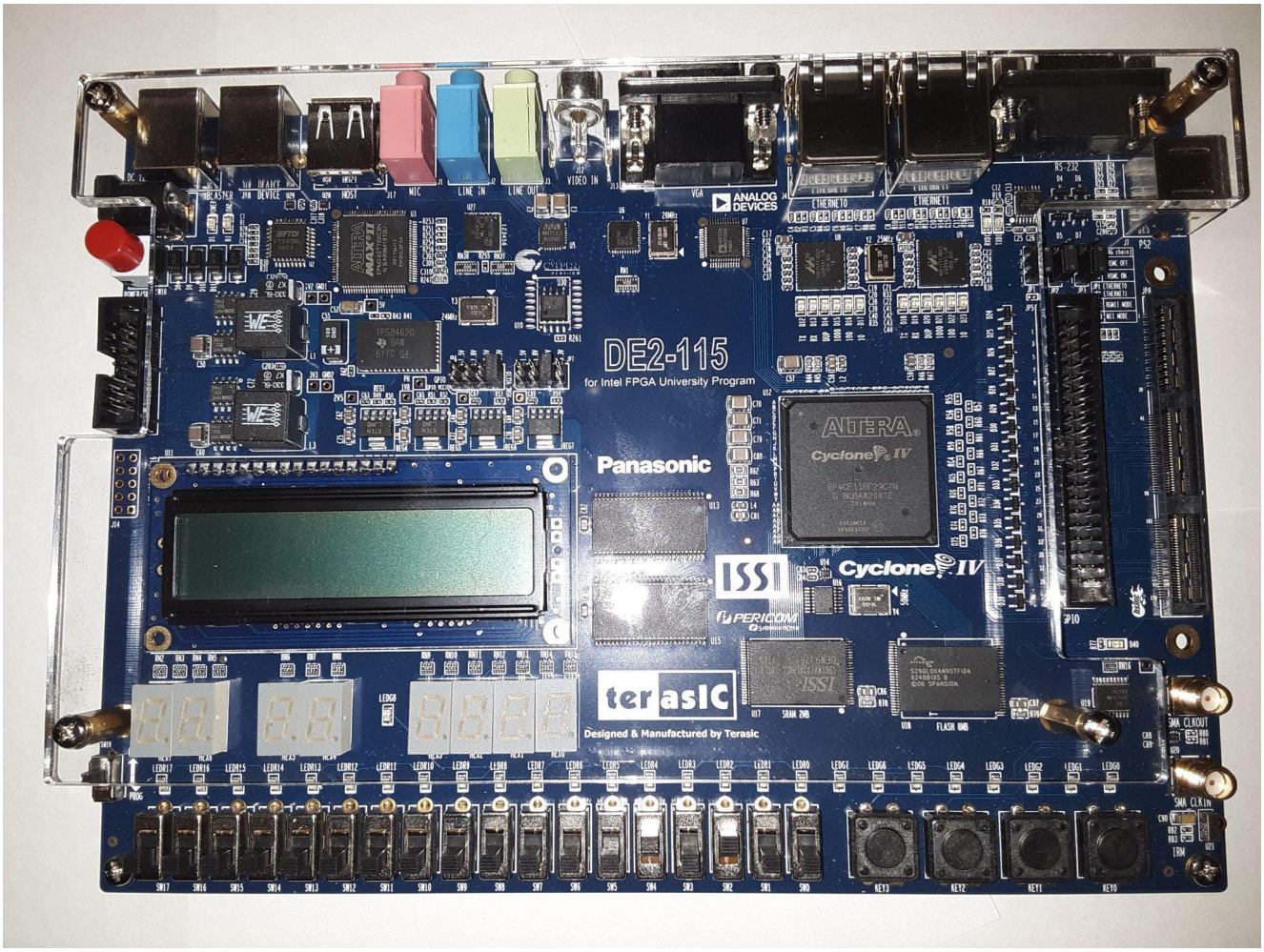


Figure 3.3: Altera DE2-115

GeeekPi TXS0108E logic level converter:

- Converts 1.2V-3.6V to 1.65V-5.5V bidirectionally
- An image of the TXS0108E logic level converter is shown below in **Figure 3.4**.

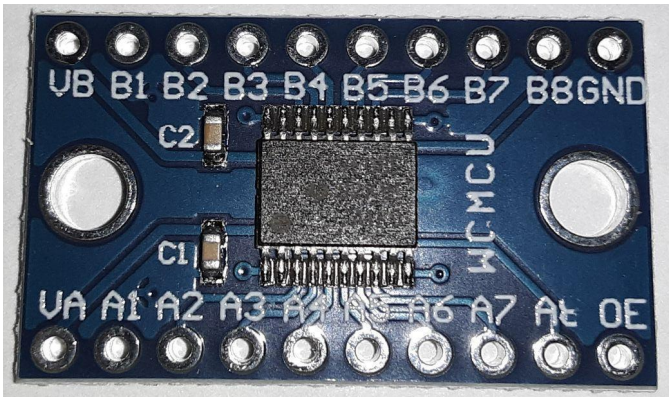


Figure 3.4: GeeekPi TXS0108E Logic Level Converter

DSD TECH HC-05 Bluetooth Module:

- Working voltage 3.6V to 6V
- Default baud rate of 9600
- An image of the HC-05 bluetooth module is shown below in **Figure 3.5**.

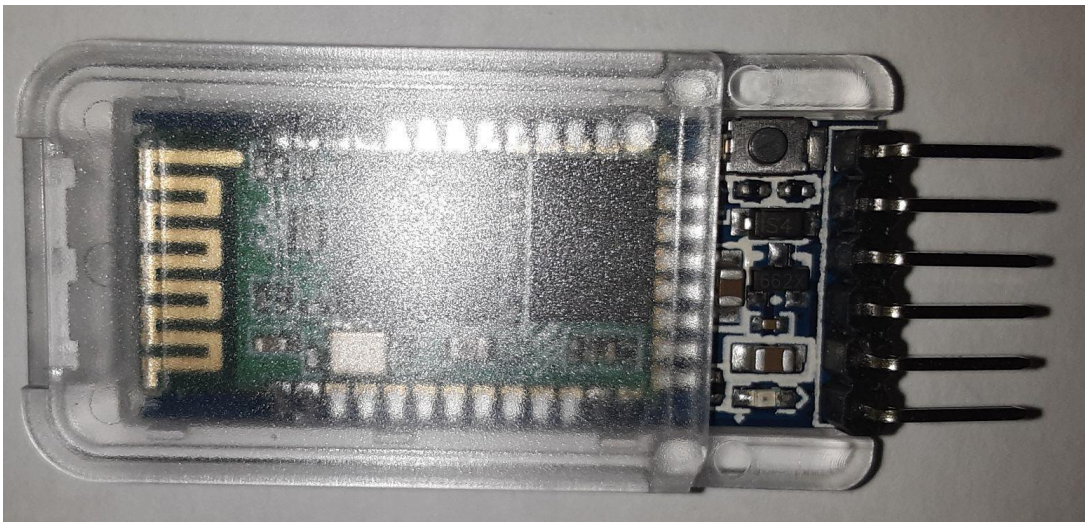


Figure 3.5: DSD TECH HC-05 Bluetooth Module

Microchip 23K256 Serial SRAM:

- Working voltage 3.6V to 6V
- Maximum clock frequency of 20MHz
- Supported temperature range of -40℃ to +85℃, sufficient for lithium ion battery temperatures
- An image of the Microchip 23K256 Serial SRAM DIP package is shown below in **Figure 3.6**.

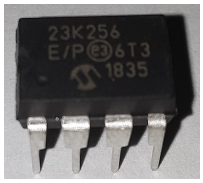


Figure 3.6: 23K256 Serial SRAM DIP

3.3 Background

3.3.1 SRAM Physically Unclonable Functions

Physically unclonable functions work by taking in challenge data, manipulating that data using a function derived from the unique physical variations of the printed circuit board, and outputting a response that can be used for authentication of the PUF device. While this is true for all PUFs in general, the specifics of the PUF implementation depend on the type of PUF being implemented. Implementing an SRAM PUF involves making use of the semi-random variations in SRAM data upon startup. The random 0's and 1's in the SRAM memory depend upon the random variations and biases in the bistable flip-flops corresponding to the SRAM memory cells. A simplified SRAM cell with two inverters is shown below in **Figure 3.7**.

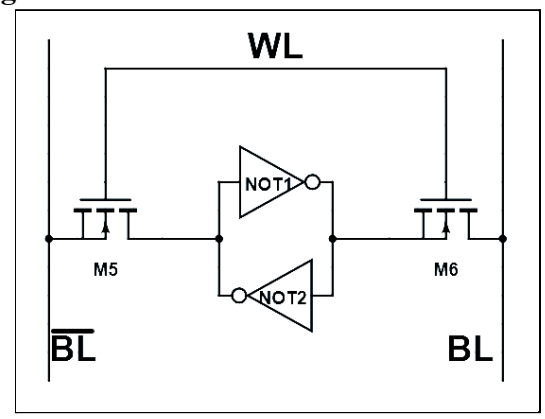


Figure 3.7: SRAM Memory Cell

The SRAM cell in **Figure 3.6** shows two bit lines denoted as BL and BL not, as well as a word line denoted as WL. For the purposes of implementing an SRAM PUF, the most important thing to understand is that when the SRAM is powered on the voltage passes through either the top or bottom inverter first, causing the bit to assume a value of either 1 or 0. Most cells tend to be biased towards one side or the other, assuming the same value with relative consistency. These are the ideal cells to use for SRAM PUF implementation, as they produce consistent responses for provided challenges.

Another key requirement one must be aware of when implementing an SRAM PUF is the need to power cycle the memory between reading the SRAM. This requires control of the voltage supply to the SRAM chip, and should be taken into consideration during design.

3.3.2 PUFs as Temperature Sensors

A PUF can also sense temperature by taking advantage of the inverse relationship between its challenge-response reliability and the temperature of the environment. In other words, as the surrounding environment heats up the PUF's challenge-response pairs (CRPs) change, since the physical characteristics of the PUF change with temperature. Knowing that the reliability decreases as temperatures rise, a set of challenge-response pairs can be established at room temperature, and as the temperature rises fewer and fewer of those pairs will remain the same. By repeatedly checking the number of correct CRPs at different temperatures, it can be determined what reliability rate applies to a given temperature. Then, the PUF can be used as a temperature sensor by checking its reliability rate at a given temperature and outputting the temperature that correlates to that reliability rate.

3.3.3 Kill Switches in Electric Vehicles

A kill switch can work in many different ways to stop an electric vehicle. One way is by diverting the electric charge which would normally power the electric traction motor. It is important to have a kill switch in any vehicle because it is a safety mechanism used to shut off machinery in an emergency when it cannot be shut down in the usual manner.

3.3.4 Controlling the 23K256 SRAM

The 23K256 SRAM is a 6-transistor static random access memory chip controlled using SPI(Serial Peripheral Interface). This is done by sending commands and their associated parameters to the SRAM in order to control a memory register within the SRAM that determines the current configuration and functionality. There is a need to monitor the signals being sent to the SRAM in order to ensure they match those for the desired command, and for this purpose an internal logic analyzer should be used.

3.3.5 HC-05 Bluetooth Communication

Bluetooth is a wireless communication standard that uses a radio frequency to share data over a short distance, eliminating the need for wires [22]. The HC-05 is a specific bluetooth module that is accessed using a universal asynchronous receiver-transmitter serial connection. The module itself is capable of security features such as authentication and encryption, but these features must be configured in the device's AT command mode in order to enable the desired functionality. In practical use the module is typically inserted upright in a breadboard, where it can then be controlled by sending

high and low voltage signals to its six pins. In order to enter AT command mode to modify security settings, the KEY pin should be held high and the small button above it pressed down when the module is powered via the VCC pin. There are three relevant AT commands for the purposes of this project. The first is “AT”, which elicits a serial response of “OK” from the module, indicating that AT command mode is functioning correctly. The second is “AT+SENM=3,2”, which sets the security features to link safety mode and sets encryption mode to both point-to-point and broadcast. The third is “AT+PSWD=PW”, where PW would be replaced with a custom password. This sets the password used for connecting securely to the bluetooth module. The default password is “1234”. Messages from the HC-05 were received using the “Bluetooth Serial Terminal” android app from the play store. After enabling the HC-05 security features it is necessary to first connect the android to the bluetooth module in the android phone settings before it can be accessed from within the terminal app.

3.3.6 Arduino MEGA 2560

The Arduino Mega 2560 is a popular microcontroller board that is typically used for projects involving digital electronics and/or circuits. It can be paired with an Arduino Ethernet Adapter for convenient access to micro SD cards, allowing storage and retrieval of data. In addition, it provides a large number of general purpose pins for sending and receiving signals, allowing for controlling motors, digital speakers, and devices such as the HC-05 bluetooth module. The Arduino IDE is required in order to code and download sketches to the board.

3.3.7 Altera DE2-115

The Altera DE2-115 is a development board with diverse functionality and a Cyclone IV FPGA. The board is equipped with a large number of general purpose input/output pins in addition to a slot for an SD card and many other onboard interfaces. The hardware system for the board can be defined using platform designer(formerly known as Qsys) in the Quartus Prime 17.1 IDE, and the desired functionality for the board can be achieved using C within the Nios II Software Build Tools for Eclipse, which can also be accessed from within Quartus Prime 17.1.

3.3.8 TXS0108E Logic Level Converter

Logic Level Converters are typically used to convert the voltage level of signals. The TXS0108E Logic Level Converter can be used to convert between 3.3 Volts and 5 Volts bidirectionally. This is necessary to ensure that the 5 Volt logic signals from the Arduino MEGA pins do not damage the 3.3 Volt pins on the DE2-115 during serial communication. It is possible to perform serial communication between the boards without using the converter, since the absolute maximum recommended voltage on the DE2-115 pins is 4.2 Volts, which means the excess voltage is only 0.8 Volts. However this still risks damage to the pins and is not recommended.

4. HARDWARE AND SOFTWARE UNIT TESTING

4.1 Arduino Mega 2560 Serial UART Test

A series of unit tests were performed prior to beginning implementation of the combined system in order to achieve a functional understanding of the hardware and software being used. The first unit test involved testing the Serial UART on the Arduino Mega 2560. This simple test was done by connecting the TX1 pin to the RX1 pin, then sending and receiving data using a serial connection between the two pins. The setup is shown below in **Figure 4.1**.

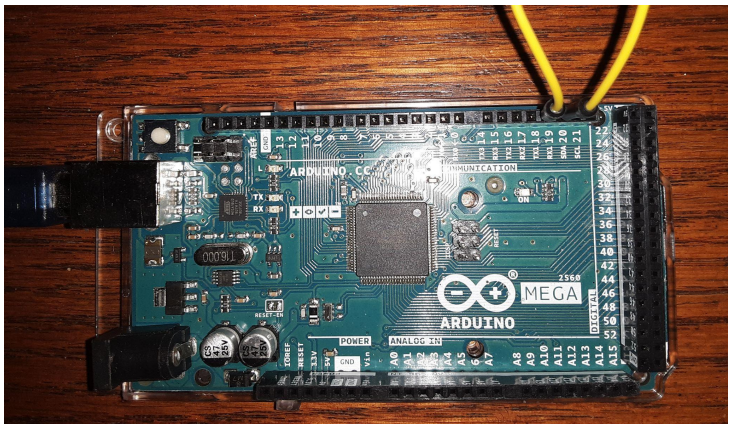


Figure 4.1: Mega 2560 UART Serial Test

The code used for the test is shown below in **Figure 4.2**. This code was typed into the Arduino IDE and loaded onto the 2560 board.

```
void setup() {
Serial.begin(9600);
Serial1.begin(9600);
}
void loop() {
// Check for received Characters from the computer
if (Serial.available())
{
// Write what is received to the default serial port
Serial1.write(Serial.read());

//wait 200 ms
//delay(200);

//print what was sent in serial monitor
Serial.print(Serial1.read());

//print space to seperate codes
Serial.print(' ');
}
}
```

Figure 4.2: Mega 2560 UART Serial Test Code

The function of the code is to send any input to the serial console through the TX1 pin and read out the result received from the RX1 pin. The results of the test are shown below in **Figure 4.3**.

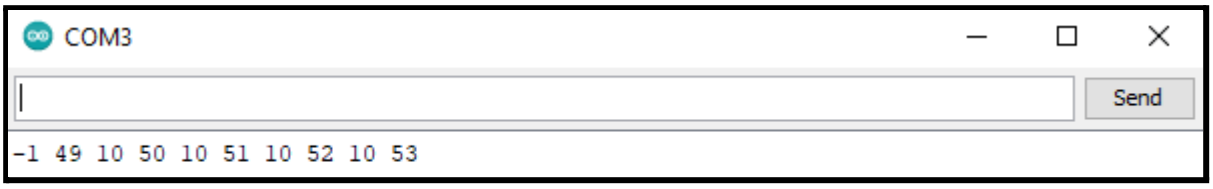


Figure 4.3: Mega 2560 UART Serial Test Results

In the above test the numbers 1, 2, 3, 4, and 5 were entered into the console one by one. The result was that the decimal ascii codes for those numbers were read from the RX1 pin receiving the data. Thus the serial test for the Mega 2560 was successful.

4.2 Arduino Mega 2560 DHT11 Temperature Sensor Test

The second unit test performed was for a DHT11 temperature sensor. The sensor was first connected to the 2560 board through its 3 pins. With the DHT11 held upright, the three pins from left to right correspond to voltage input, data output, and ground. The setup is shown below in **Figure 4.4.**

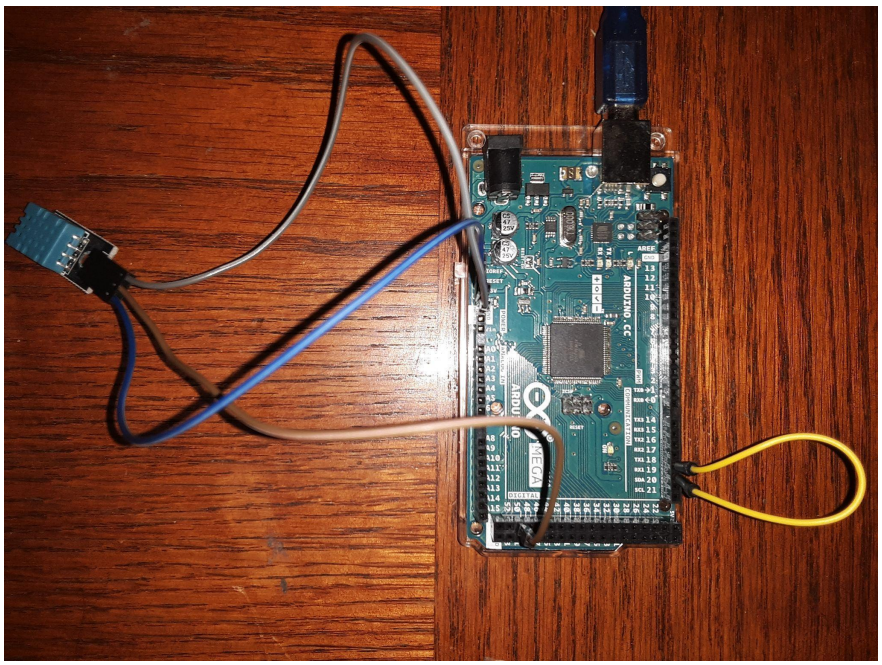


Figure 4.4: Mega 2560 DHT11 Temperature Sensor Test

Some basic test code was used to test the temperature sensor and read out sensed values. The code used and the resulting output is shown below in **Figure 4.5**.

Temp_Sensor_Test1

```
//Arduino Mega 2560 DHT11 Temperature sensor test
#include "DHT.h"
#define DHTPIN 50     // Digital pin connected to the DHT sensor
#define DHTTYPE DHT11   // DHT 11
// Initialize DHT sensor.
DHT dht(DHTPIN, DHTTYPE);
void setup() {
  Serial.begin(9600);
  Serial.println(F("DHTxx test!"));
  dht.begin();
}
void loop() {
  delay(2000);  // Wait a few seconds between measurements.
  float t = dht.readTemperature();  // Read temperature as Celsius (the default)
  float f = dht.readTemperature(true);  // Read temperature as Fahrenheit (isFahrenheit = true)
  if (isnan(t) || isnan(f)) {  // Check if any reads failed and exit early (to try again).
    Serial.println(F("Failed to read from DHT sensor!"));
    return;
  }
  Serial.print(F("%  Temperature: "));
  Serial.print(t);
  Serial.print(F("°C "));
  Serial.print(f);
  Serial.print(F("°F \n"));
}
```

COM3

```
%  Temperature: 23.00°C 73.40°F
```

Autoscroll Show timestamp

Figure 4.5: Mega 2560 DHT11 Temperature Sensor Test Code and Results

The code shown in **Figure 4.5** sets up the sensor, stores measured values repeatedly, and outputs them to the terminal. The results can be seen in the COM3 terminal on the right side of the image.

4.3 Altera DE2-115 Serial UART Test

The next test performed involved testing serial UART on the Altera DE2-115 development board. This test proved more complex than testing serial on the Arduino, since the DE2-115 test required a hardware system to first be developed in Quartus Prime using Qsys. This step had to be performed first in order to code the UART test using Quartus Prime Tools for Eclipse. First, the hardware system was defined in Quartus Prime using Qsys, as shown in **Figure 4.6** below.

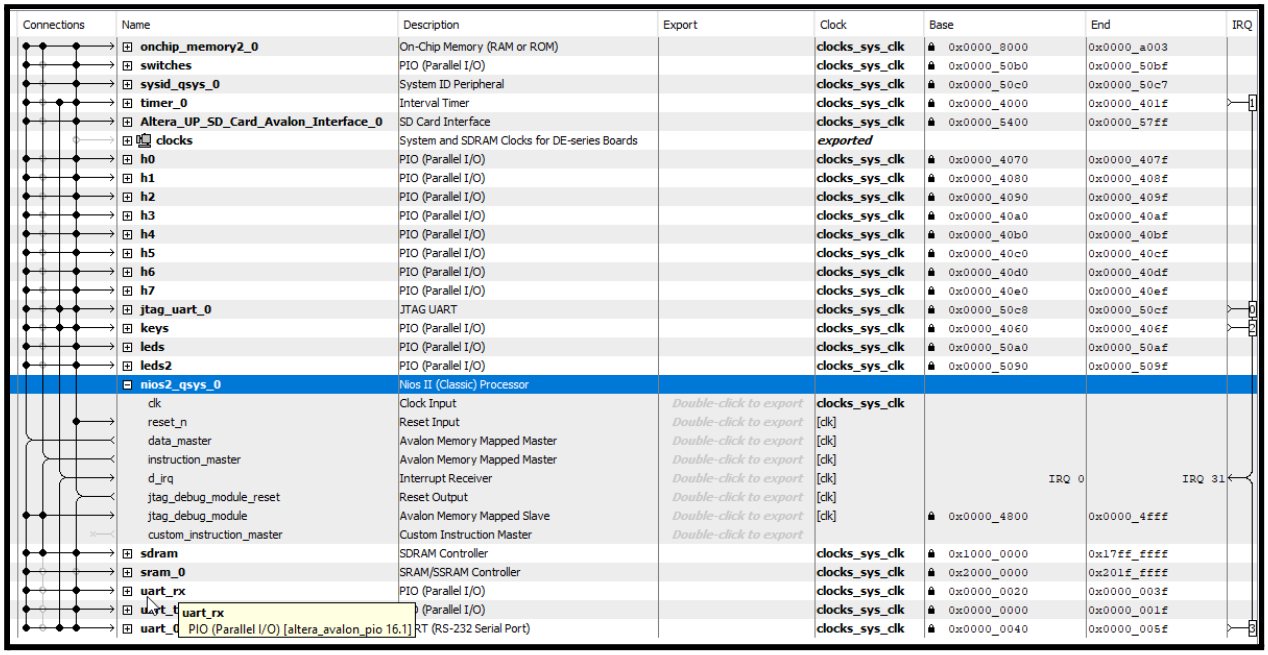


Figure 4.6: DE2-115 UART Test Qsys

In addition to the basic hardware components such as memory and the Nios II processor, the UART RS-232 serial port IP core was added to the design. This was necessary in order to realize UART through the GPIO pins of the DE2-115. The HDL for this hardware system was then generated in Qsys, and the corresponding top level verilog module was written in Quartus Prime. The top level verilog module served to define which pins would correspond to which inputs and outputs.

The hardware system used for the test also contained support for hexadecimal displays, sdram, sram, and a number of other IP cores such as SD-card. As a result the verilog module was lengthier than was strictly necessary for the purposes of this test, but it functioned as intended nonetheless. This verilog module is shown below in **Figure 4.7**.

```
 3  module HW4 (CLOCK_50, SW, KEY, LEDR, LEDG, DRAM_ADDR,DRAM_BA,DRAM_CAS_N,DRAM_CKE,
 4              DRAM_CS_N,DRAM_DQ,DRAM_DQM,DRAM_RAS_N,DRAM_WE_N,DRAM_CLK, HEX0, HEX1,
 5              HEX2, HEX3, HEX4, HEX5, HEX6, HEX7, SD_DAT, SD_CMD, SD_CLK,
 6              SRAM_ADDR, SRAM_DQ, SRAM_CE_N, SRAM_OE_N, SRAM_WE_N, SRAM_UB_N, SRAM_LB_N, GPIO);
 7   input CLOCK_50;
 8   input [17:0] SW;
 9   input [3:0] KEY;
10   output [17:0] LEDR;
11   output [7:0] LEDG;
12   output DRAM_CAS_N,DRAM_CKE,DRAM_CS_N,DRAM_RAS_N,DRAM_WE_N,DRAM_CLK;
13   output [12:0] DRAM_ADDR;
14   output [3:0] DRAM_DQM;
15   output [1:0] DRAM_BA;
16   output [6:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5,HEX6,HEX7;
17   inout [31:0] DRAM_DQ;
18   inout [0:0] SD_CMD, SD_CLK;//SD_DAT, SD_DAT3,
19   inout [3:0] SD_DAT;
20   inout [15:0] SRAM_DQ;
21   output [19:0] SRAM_ADDR;
22   output [0:0] SRAM_CE_N;
23   output [0:0] SRAM_OE_N;
24   output [0:0] SRAM_WE_N;
25   output [0:0] SRAM_UB_N;
26   output [0:0] SRAM_LB_N;
27   inout [3:0] GPIO;
28   wire [2:0]k1;
29   assign k1 = {KEY[3],KEY[2],KEY[1]};
30   // Instantiate the Nios II system module generated by the Qsys tool:
31  ECE186 NiosII (
32   .switches_export(SW),
33   .leds_export(LEDR),
34   .leds2_export(LEDG),
35   .sdram_wire_addr(DRAM_ADDR),
36   .sdram_wire_ba(DRAM_BA),
37   .sdram_wire_cas_n(DRAM_CAS_N),
38   .sdram_wire_cke(DRAM_CKE),
39   .sdram_wire_cs_n(DRAM_CS_N),
40   .sdram_wire_dq(DRAM_DQ),
41   .sdram_wire_dqm(DRAM_DQM),
42   .sdram_wire_ras_n(DRAM_RAS_N),
43   .sdram_wire_we_n(DRAM_WE_N),
44   .sdram_clk_clk(DRAM_CLK),
45   .clk_clk(CLOCK_50),
46   .h0_export(HEX0),
47   .h1_export(HEX1),
48   .h2_export(HEX2),
49   .h3_export(HEX3),
50   .h4_export(HEX4),
51   .h5_export(HEX5),
52   .h6_export(HEX6),
53   .h7_export(HEX7),
54   .keys_export(~k1),
55   .sd_card_b_SD_cmd(SD_CMD),
56   .sd_card_b_SD_dat(SD_DAT[0]),
57   .sd_card_b_SD_dat3(SD_DAT[3]),
58   .sd_card_o_SD_clock(SD_CLK),
59   .sram_wire_DQ(SRAM_DQ),
60   .sram_wire_ADDR(SRAM_ADDR),
61   .sram_wire_LB_N(SRAM_LB_N),
62   .sram_wire_UB_N(SRAM_UB_N),
63   .sram_wire_CE_N(SRAM_CE_N),
64   .sram_wire_OE_N(SRAM_OE_N),
65   .sram_wire_WE_N(SRAM_WE_N),
66   .rx_export(GPIO[0]),
67   .tx_export(GPIO[1]),
68   .uart_0_rxd(GPIO[2]),
69   .uart_0_txd(GPIO[3]),
70   .reset_reset(~KEY[0]));
71   endmodule
72
```

Figure 4.7: Top-level Verilog Module for DE2-115 UART Test

In the above verilog module the most important lines for the purpose of the UART test were lines 68 and 69, which define the RX and TX serial pins, defining them as general purpose input output pins 2 and 3. Thus, the GPIO pins 2 and 3 were used for testing serial UART on the DE2-115.

The last step of testing UART on the DE2-115 required writing some C language code in the Eclipse IDE included within Quartus Prime. A simple C program was written that would send and receive the byte “x” from pin 3 to pin 2. In other words the DE2-115 was communicating with itself using UART. The received data was then output to the terminal. The connection between the pins and the program test code are shown below in **Figure 4.8** and **Figure 4.9** respectively.

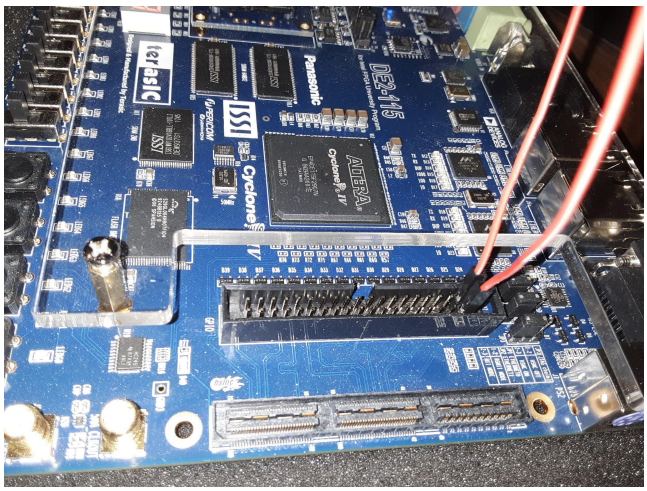


Figure 4.8: DE2-115 UART Test Pin Connection

```
1
2  #include <stdio.h>
3  #include "system.h"
4  #include "altera_avalon_uart_regs.h"
5
6  void Time_Delay(int x)
7  {
8      while (x>0)
9      {x--;}
10 }
11
12 void demo1(){
13  char a = 'x';
14  int b;
15  IOWR_ALTERA_AVALON_UART_TXDATA(UART_0_BASE, a);
16  b = IORD_ALTERA_AVALON_UART_RXDATA(UART_0_BASE);
17  printf("%d", b);
18 }
19
20 int main()
21 {
22   while (1)
23   {
24   demo1();
25   Time_Delay(5000000);
26   }
27 }
28
```

Figure 4.9: DE2-115 UART Test Code

The result of the DE2-115 UART test is shown below in **Figure 4.10**.

Figure 4.10: DE2-115 UART Test Results

The test results shown in the Nios II console show that the RX register repeatedly received a value of 120, which corresponds to the decimal ascii value for "x". Thus, the DE2-115 UART serial test was successful.

4.4 GeeekPi TXS0108E Logic Converter Test

An additional test was necessary in order to verify that the logic converter was converting voltage levels properly, thereby preventing any damage to the DE2-115 during serial communication with the Mega 2560. This damage can occur due to the 2560 having 5V operating voltage and the DE2-115 having a 3.3V operating voltage. A simple test was performed using the TXS0108E logic converter along with a multimeter as shown in **Figure 4.11** below.

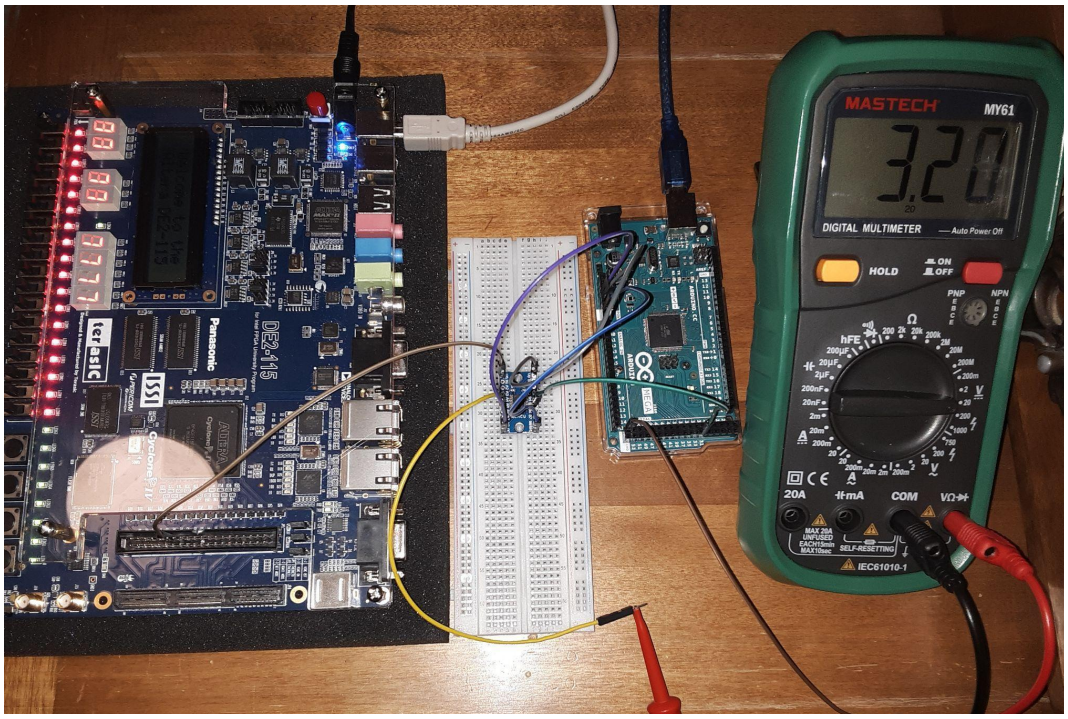


Figure 4.11: TXS0108E Logic Converter Test

As shown in Figure 4.11, the 5V output of the Mega 2560 is converted to 3.20V, which is close enough to the ideal 3.3V for use with the DE2-115 pins. The test was therefore a success.

4.5 HC-05 Bluetooth Module Test with Android

A unit test for the HC-05 Bluetooth module was performed by sending bluetooth data to a generic receiver app on an Android phone. The name of the Android app used was Serial Bluetooth Terminal, although any generic bluetooth terminal app should be adequate for testing purposes. The pins of the HC-05 were connected in such a way that only the middle 4 pins were used. The first and last pins of the module were not used. The middle four pins from left to right were connected to a 3.3V source, ground, TX2, and RX2 respectively. This setup is shown below in **Figure 4.12**.

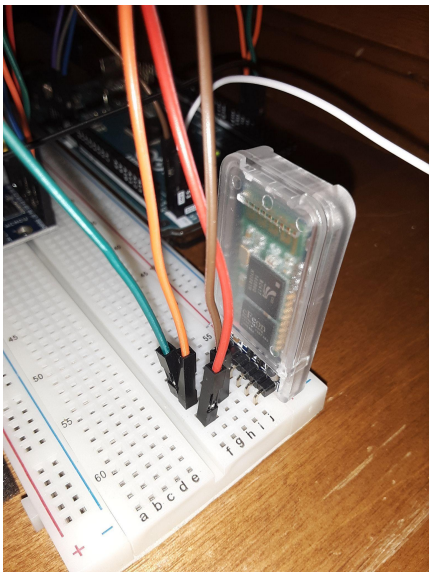


Figure 4.12: HC-05 Bluetooth Module Test Setup

After this setup was complete a simple Arduino program was used to send data to the Android. This program is shown below in **Figure 4.13**.

Bluetooth_Test1

```
char state = 'A';

void setup() {
  // put your setup code here, to run once:
  Serial.begin(9600);
  Serial2.begin(9600);
}

void loop() {
  // put your main code here, to run repeatedly:
  if (Serial.available()>0){
    state = Serial.read();
  }
  delay(1000);
  Serial2.print(state);
}
```

Figure 4.13: HC-05 Bluetooth Module Test Code

The bluetooth terminal Android app was then connected to the available bluetooth signal from the hc-05, and was able to receive data transferred via bluetooth, as shown in **Figure 4.14** below.

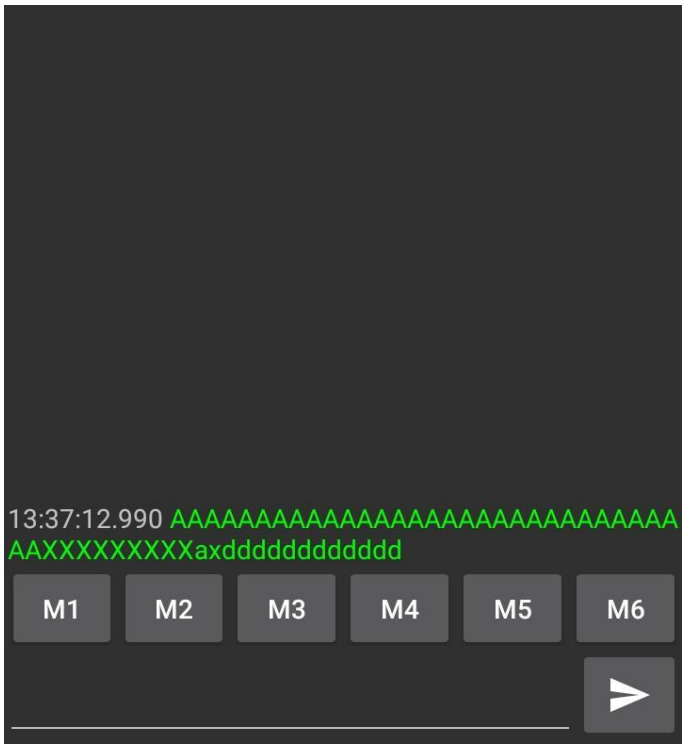


Figure 4.14: HC-05 Bluetooth Data Received in Android App

Thus this simple test of a bluetooth connection to the Android phone was successful.

4.6 Arduino MEGA 2560 WebServer test with Arduino Ethernet 2 Shield

The way to perform a unit test for this web server setup was to first attach the Ethernet shield to the Arduino Mega. The pins of the shield must connect to the power supply of the board along with the digital and analog pins to be able to communicate with the board, as shown in **Figure 4.15** below. This allowed the arduino to connect to the internet.

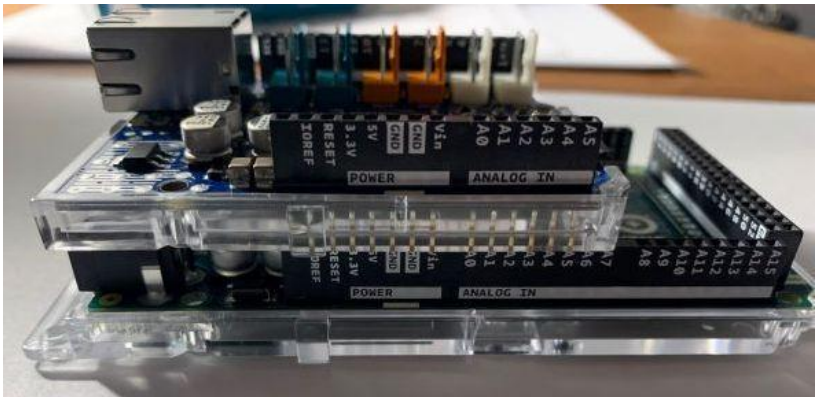


Figure 4.15: Connection between the Ethernet Shield and Arduino Mega

Now for the programming of the board to be able to communicate with the shield as the code shown below in **Figure 4.16** a mac address of the shield is added so that it is able to pick up an IP address within the network. The first thing on the code for the arduino to do is to pick up the webpage as it was implemented on html language which is in the micro SD card that is identified as index.htm file. The code is implemented to state whether it reads the page or not. The next step is to read the file index.htm to be able to pick up the content and add it to the network with the given ip that it is assigned. Note that although this test was performed, in the actual implementation no web server was used. Data was simply stored directly to the SD card through the Arduino Ethernet Shield as described in **Section 7.6** of this report.

```
#include <SPI.h>
#include <SD.h>



void setup()
{
  //Ethernet.begin(mac);
  server.begin();
  Serial.begin(9600);

  Serial.println(Ethernet.localIP());

  Serial.println("Initializing SD card...");
  if (!SD.begin(4)) {
    Serial.println("ERROR - SD card initialization failed!");
    return;
  }
  Serial.println("SUCCESS - SD card initialized.");
  if (!SD.exists("index.htm")) {
    Serial.println("ERROR - Can't find index.htm file!");
  }
  Serial.println("SUCCESS - Found index.htm file.");
}
```

Figure 4.16: Arduino Ethernet Web Server Test

4.7 Arduino MEGA 2560 Vibration Motor Test using L298N Motor Drive Controller

A Vibration Motor Test was performed using an Arduino MEGA with a L298N motor drive controller and a micro DC motor. A motor drive controller was used for this test in order for the pins on the Arduino board to not burn out by powering the DC motor. The motor drive controller was powered using the 5V power pin on the Arduino MEGA. The tests can be seen below in **Figure 4.17** and **Figure 4.18**. The code used to test the motor driver and vibration motor on the Arduino is shown in **Figure 4.19**.

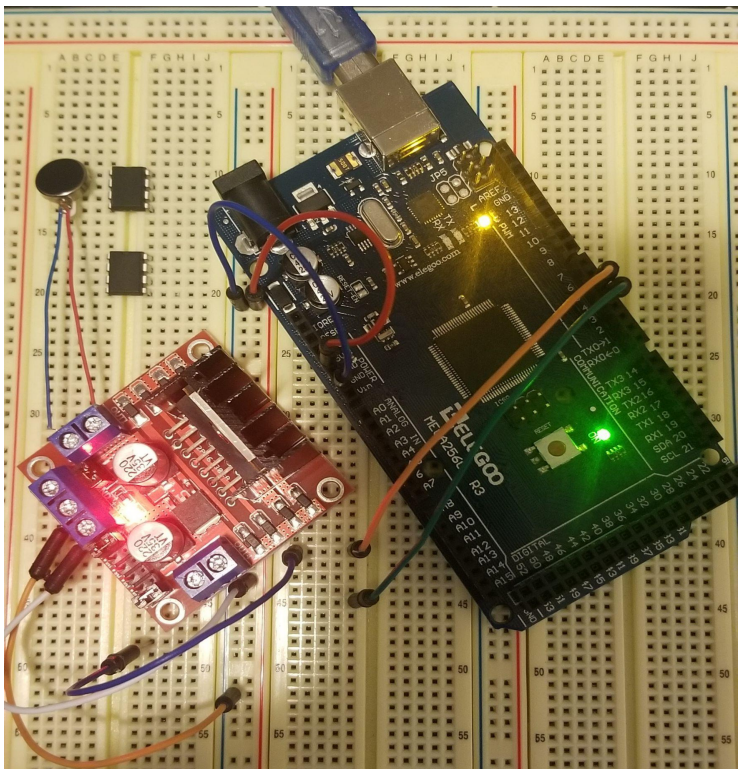


Figure 4.17: Vibration Motor Test

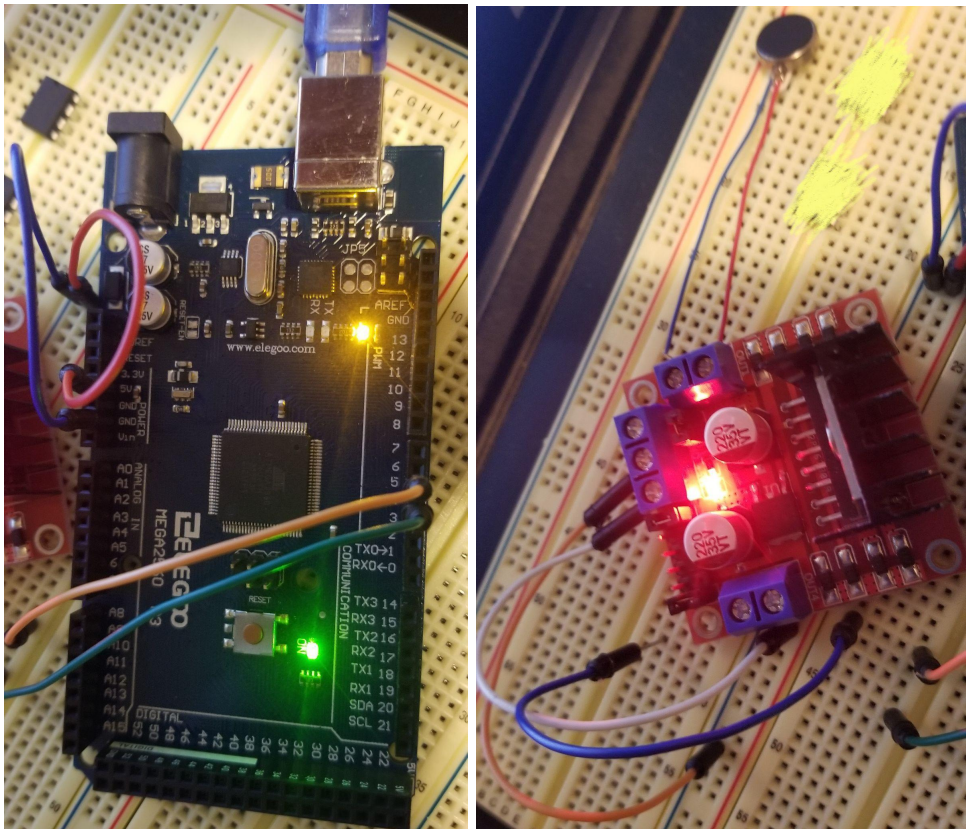


Figure 4.18: Arduino MEGA 2560 & L298N Motor Drive Controller w/ Micro mini DC motor

Figure 4.19: Vibration Motor Test code

5. METHODOLOGY

5.1 Design Overview

The first step of the project methodology is to create the database needed for creating Challenge Response Pairs and storing them. The second step is to establish a wired serial connection between the Arduino and the DE2-115 using UART. It is then necessary to implement the control signal logic for the SRAM memory on the DE2-115's Cyclone IV FPGA. Next it is necessary to complete the enrollment phase for the PUF. The reliability of the PUF will then be recorded from the acceptable temperature range to the warning range to the shutoff threshold. The resulting data is then stored in the CRP database. Lastly, the Arduino must be coded to calculate the temperature by determining the reliability of the SRAM PUF that is being controlled using the FPGA. Based on the sensed temperature that is derived from the reliability, outputs are then sent from the Arduino to the steering wheel/phone.

5.2 Hardware Alternatives

The main reason the design team decided to use the DE2-115 and its Cyclone IV FPGA in this project was that initially the intention was to use the onboard SRAM to implement the SRAM PUF. The team later became aware that this was not possible because SRAM PUF implementations require the ability to power cycle the SRAM at will, and it is not possible to power cycle the DE2-115 onboard SRAM exclusively without restarting the power of the entire board. For this reason the external serial SPI SRAM chip was used, allowing for control over when the memory had access to power. The intention behind using the Arduino was initially to use it for implementing an offsite server for storing CRP's securely, but it was later determined that it was more efficient to have the system isolated within the vehicle in order to avoid server accessibility issues. One cheaper alternative device that could replace

both the DE2-115 and the Arduino MEGA 2560 in this design is a Raspberry PI. This would reduce costs while still providing the desired functionality. In addition, the higher clock speed of a recent Raspberry Pi model may result in shorter measurement time for reliability and by extension temperature. Temperature measurements would therefore be more accurate, since less time would be required to determine reliability, decreasing the span of time in which the temperature could fluctuate.

5.3 Top-level Design Block Diagram

The top-level design for this methodology is shown below in **Figure 5.1**.

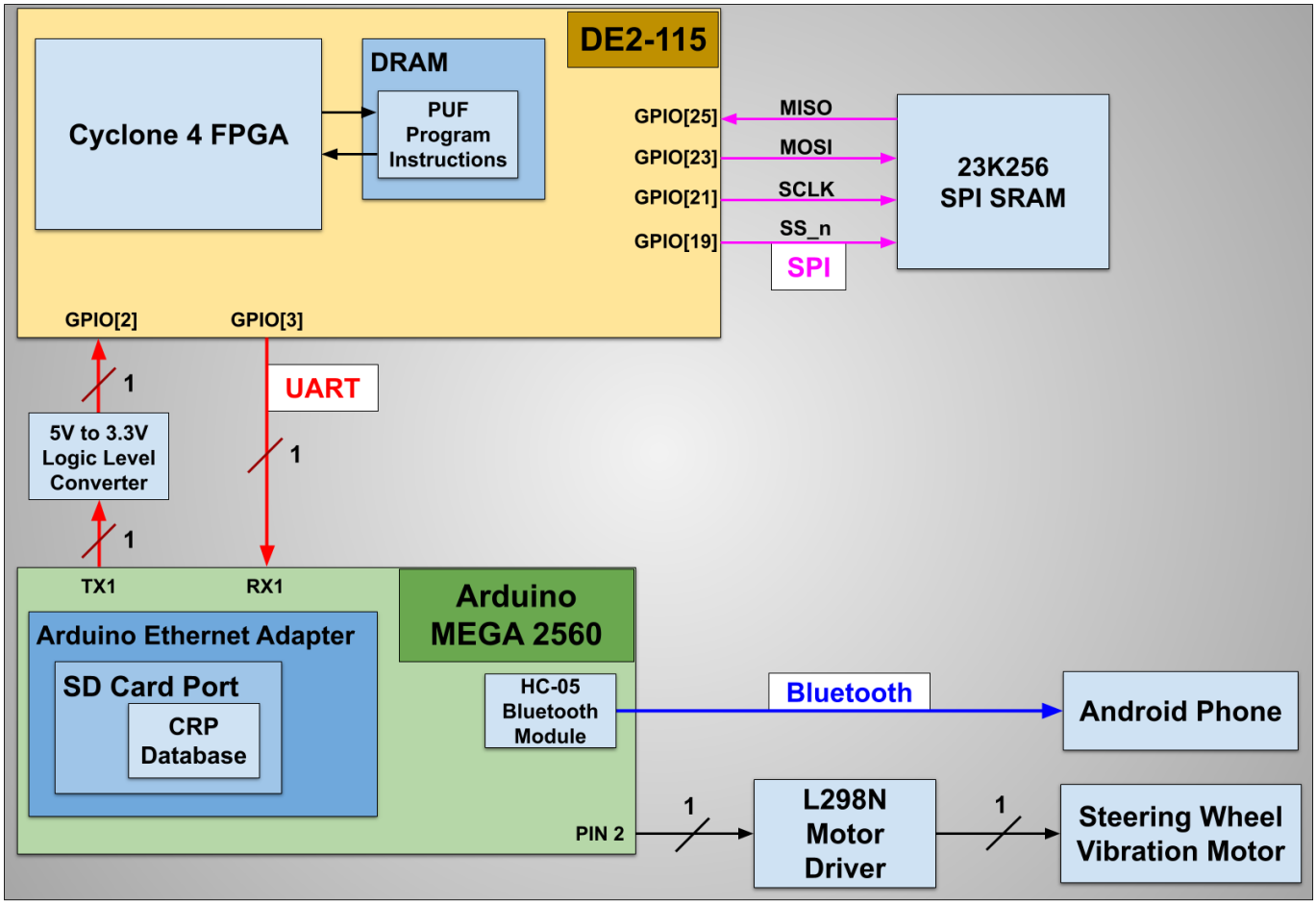


Figure 5.1: Top-level Design

Figure 5.1 shows how the Cyclone 4 FPGA in the DE2-115 board interfaces with the Arduino Mega 2560 using UART serial communication. The Arduino Mega 2560 contacts the server database, sends challenges to the DE2-115 via serial, and receives PUF responses which it then stores in the server. The DE2-115 receives challenges and uses its offboard PUF implementation to generate responses which are then sent back to the Arduino. Note that a 5V to 3.3V conversion occurs when sending data from the Arduino to the DE2-115. This prevents damaging the 3.3V DE2-115 logic pins with a 5V

signal from the Arduino. Accessing the serial 23K256 SRAM is accomplished using a serial peripheral interface. Setup of this interface must be completed prior to PUF implementation in order to provide the FPGA with access to the SRAM memory. Once the enrollment phase is complete the Arduino will still need access to the DE2-115 in order to actively sense the temperature based on the PUF reliability. The Arduino also must connect to the necessary output devices, such as vibrational motors in the steering wheel and the driver's phone. Connections between the driver's phone and the arduino will be established using bluetooth, and the vibration motors will be controlled through a L298N motor Driver connected directly to the pins of the Arduino.

5.4 Design Algorithms

The algorithm used to sense temperature using the PUF was divided into three phases; enrollment, correlation, and sensing. The algorithm for the enrollment phase is shown below in **Figure 5.2.**

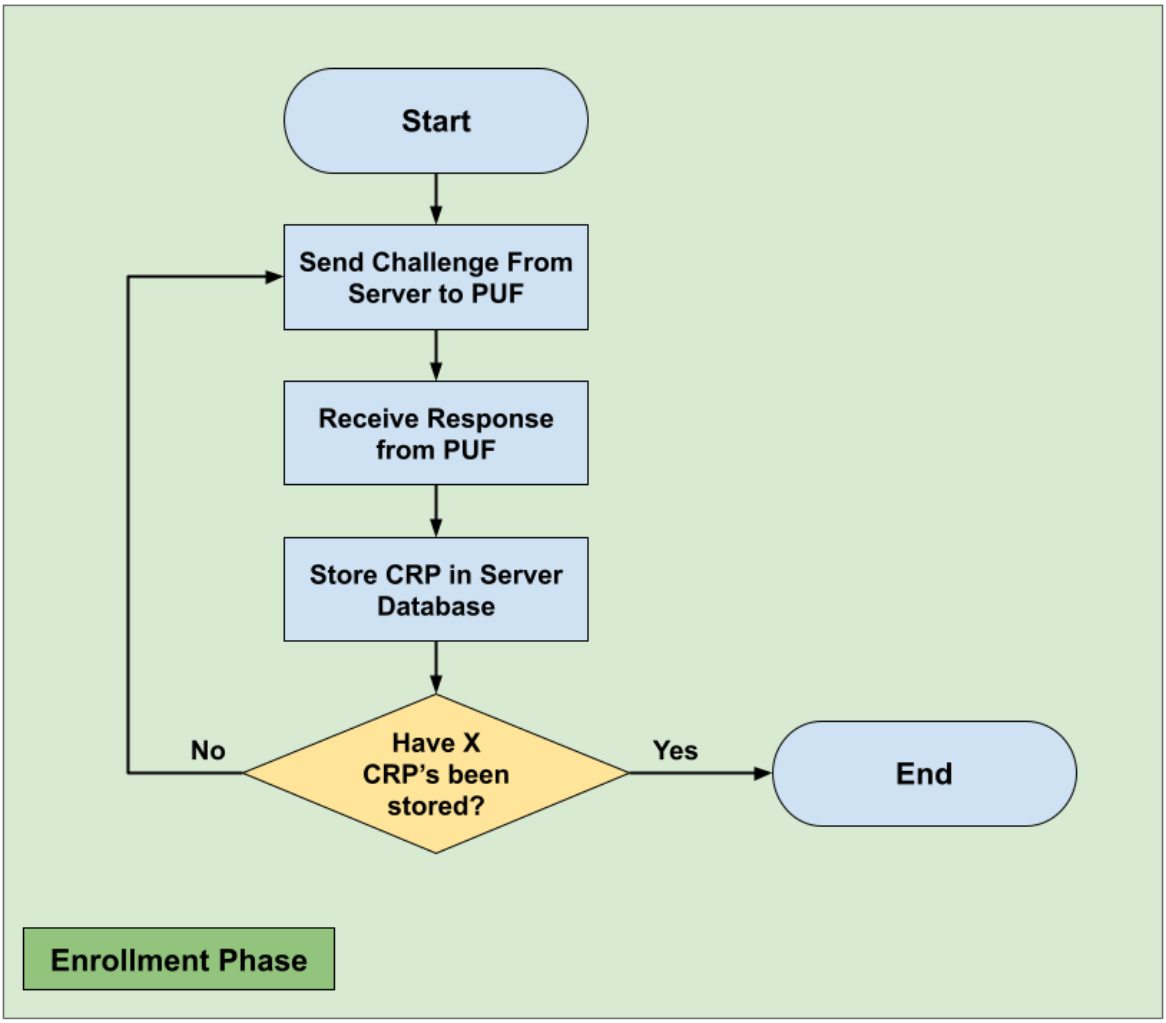


Figure 5.2: Enrollment Phase Flow Chart

Figure 5.2 demonstrates how the enrollment phase algorithm, where Challenge Response Pairs are generated and stored in the server database. This phase continues until a sufficient number of CRPs have been stored in the database. The algorithm shown in **Figure 5.2** is accomplished using the architecture shown previously in **Figure 5.1**. In other words, the Arduino sends challenges from its server to the PUF onboard the DE2-115, then receives responses from the DE2-115. These responses are then stored in the server database along with the corresponding challenge. This process repeats until a sufficient number of CRPs (represented by the letter X in **Figure 5.2**) are stored in the database.

Once the enrollment phase is complete the correlation phase shown below in **Figure 5.3** begins.

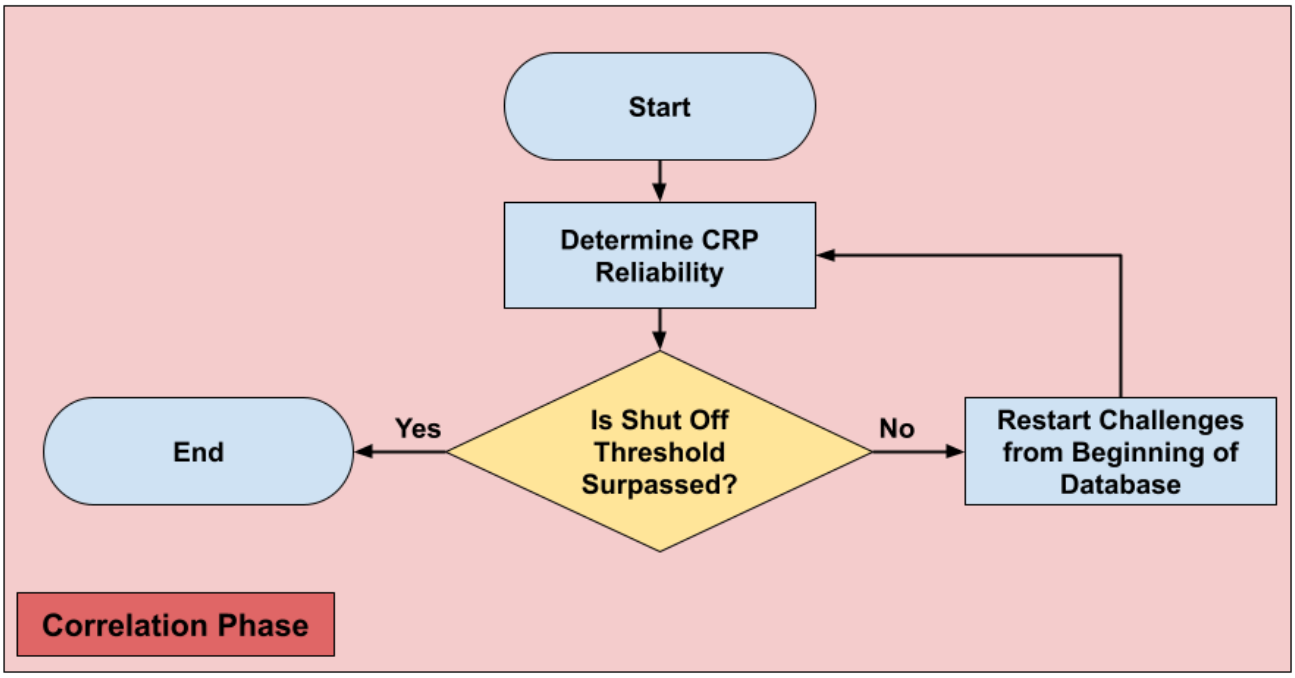


Figure 5.3: Correlation Phase Flow Chart

In the correlation phase shown in **Figure 5.3** the correlation between PUF reliability and temperature is established by repeatedly testing the reliability and measuring the corresponding PUF temperature. After determining CRP reliability the temperature of the PUF is manually increased by 1 degree Celsius, then the reliability is determined again. This process continues until the temperature of the PUF is above the shutoff threshold temperature. The reason for this is to ensure that enough data is being recorded to take into account any erratic behavior at the high end of the temperature range.

Determining Challenge-response pair reliability is a key process that is repeated in both the correlation and sensing phases. In the correlation phase reliability must be determined in order to establish a relationship between it and PUF temperature. In the sensing phase reliability must be determined in

order to subsequently determine the corresponding temperature from the database. A flow chart detailing the process of determining CRP reliability is shown below in **Figure 5.4**.

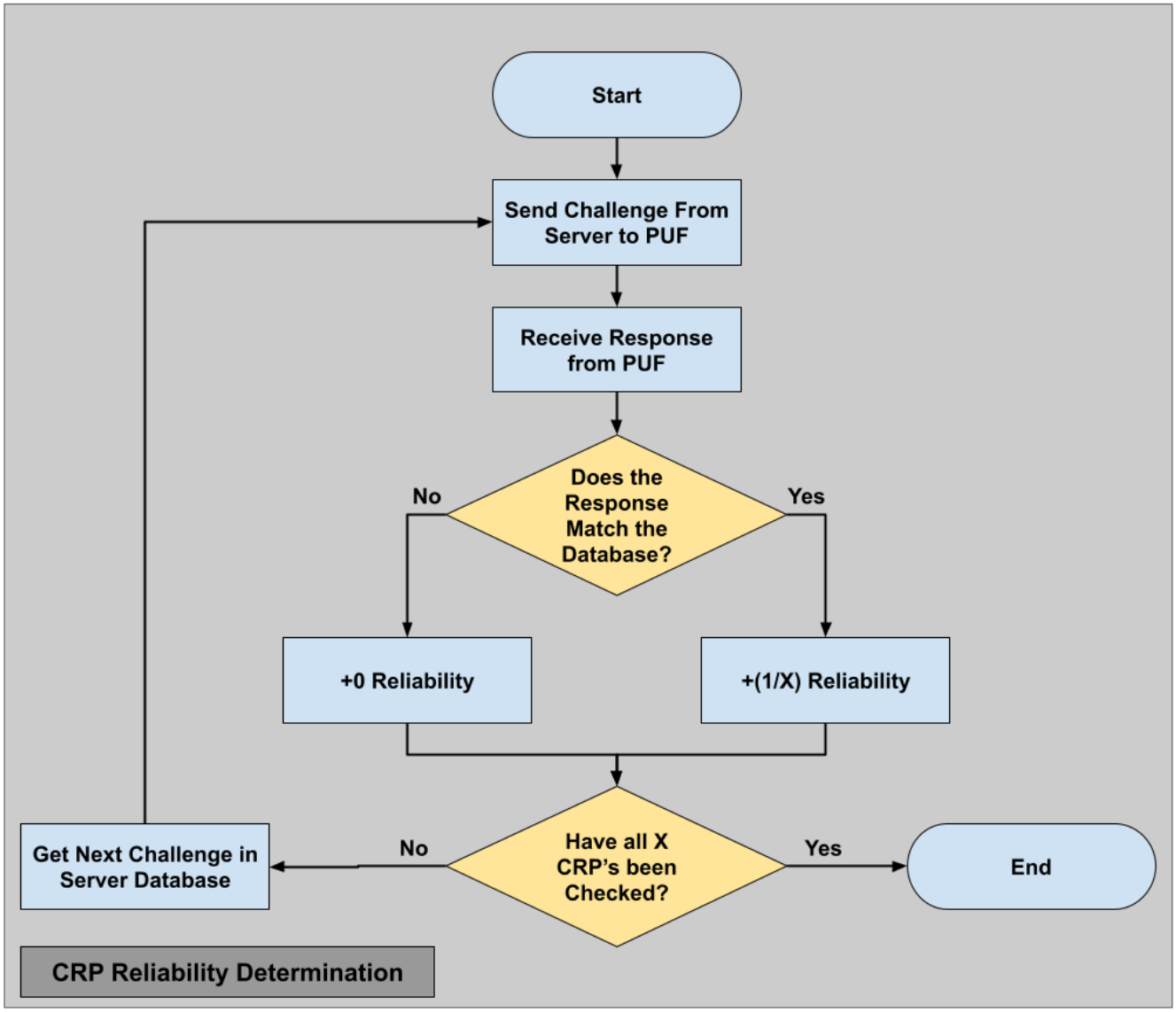


Figure 5.4: CRP Reliability Determination Flow Chart

A dataset that describes the relationship between temperature and reliability is created and stored in the database, allowing temperature to be determined only from reliability in the final phase of the methodology. After every X comparisons the temperature is increased by 1 degree Celsius, up until the PUF temperature is above the shut off threshold of 60 degrees Celsius. The reason for this is to ensure that enough data is being recorded to take into account any erratic behavior at the high end of the temperature range.

After completion of the correlation phase, the final phase can begin. This final phase is referred to as the sensing phase, the algorithm for which is shown below in **Figure 5.5**.

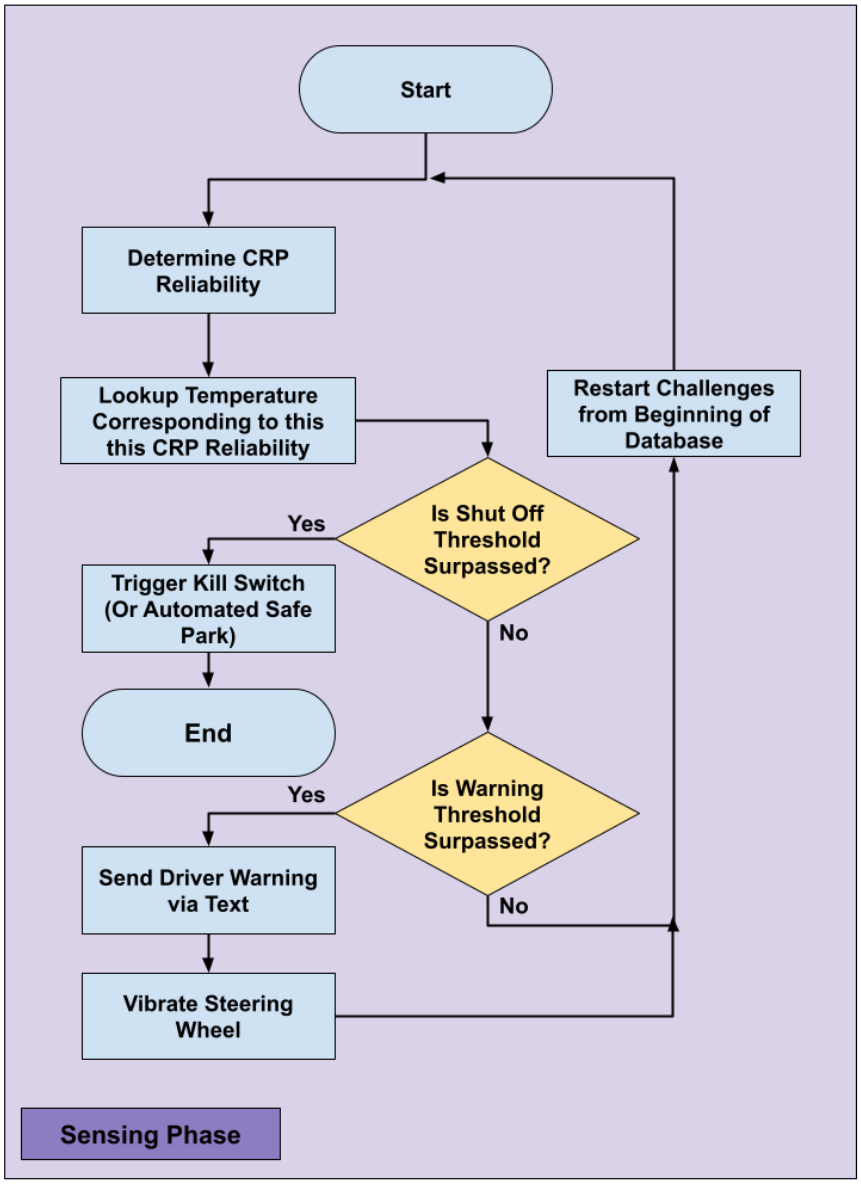


Figure 5.5: Sensing Phase Flow Chart

In **Figure 5.5** above the reliability of the PUF is constantly tested and the corresponding temperature is then calculated using the correlation data stored in the database. After checking the temperature safety features may be triggered based on whether the temperature has exceeded certain thresholds. Once the CRPs have all been checked the next thing is to check the temperature corresponding to the reliability and determine whether it exceeds the shut off or warning threshold. If the shut off is triggered then the battery power will begin to shut down. If the warning is triggered then a text is sent to the driver's phone advising to proceed with caution and pull over to regulate the battery temperature. In addition

the steering wheel motors vibrate when the warning temperature is exceeded. Otherwise the system will continue to monitor the temperature without triggering any reactive safety measures.

6. IMPLEMENTATION PLAN

The implementation of the project is expected to span a period of around four months. During this time the project will progress from theoretical proposal to finished implementation. Major milestones will be used to mark the progress being made towards achieving the completed project functionality. These milestones include implementing the SRAM PUF, implementing serial transfer of CRPs between the Arduino and the DE2-115, and coding the database that will be used to store the CRPs. In general the architecture of the design must be complete before the enrollment and other algorithm phases can begin. For this reason, connecting the different hardware units and getting them to work together as intended will be the first priority. Once this is complete the process of storing CRPs and taking temperatures will begin, and will mainly involve software coding of the Arduino and DE2-115. The subsequent priority is to set up the final outputs of the system, such as triggering the vibration motor, emitting a tone from a digital speaker, and sending a message to an android phone. Once the temperature monitoring system is functioning correctly as a whole, the next priority is to begin work coding a custom Android application in Java. This application will be coded to include built-in security in the application layer. A chart detailing an approximate timetable for the project is shown below in **Figure 6.1**.

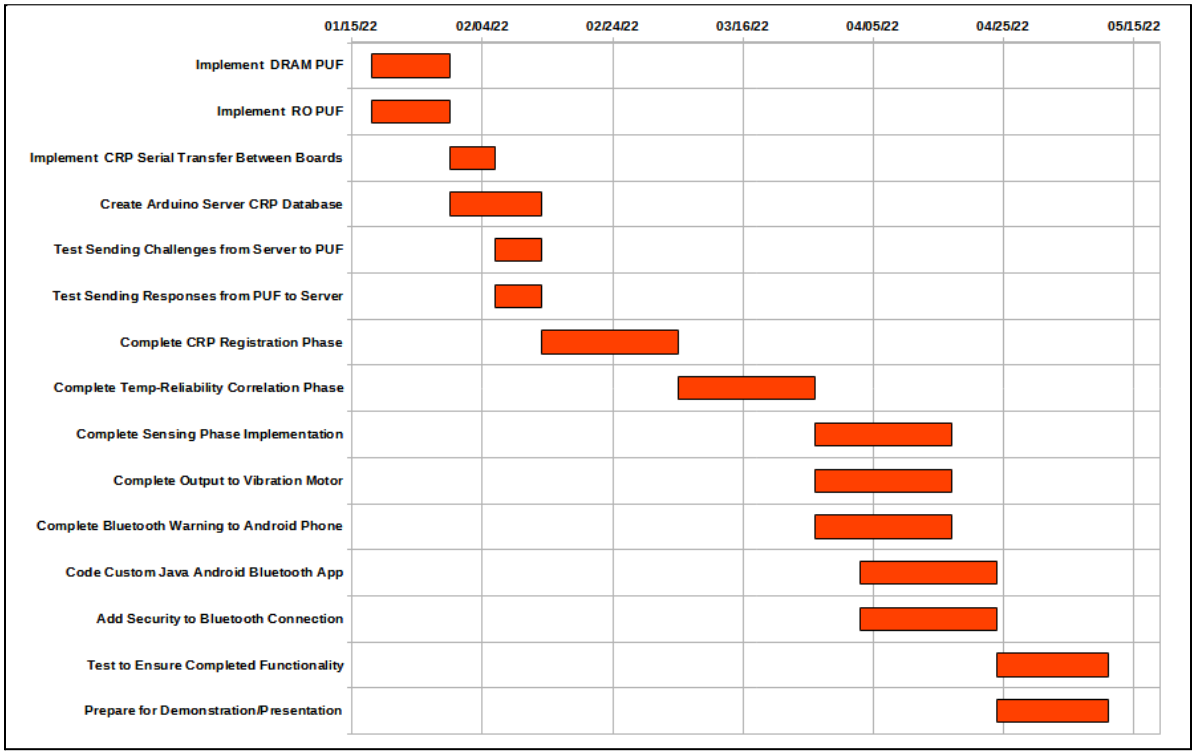


Figure 6.1: Project Implementation Gantt Chart

It is expected that the implementation timeline may differ from our anticipated timeline in **Figure 6.1** based on unforeseen challenges that may arise. In order to make time for these unexpected obstacles the timeline may have to be adjusted to some degree. Nonetheless it is the intention of the design team to adhere to the timeline as closely as circumstances allow.

7. IMPLEMENTATION

7.1 Serial CRP transfer between Arduino and DE2-115

The first step of the implementation was successfully connecting the DE2-115 and the Arduino Mega 2560 using a serial UART connection. For the DE2-115, this required adding the rs232 UART IP Core in Qsys. The Verilog HDL was then generated and the two wires were mapped to GPIO pins in the Verilog top module. This provided the DE2-115 with the necessary interface to perform UART communication. Two additional pins were added as well, an output and an input, using the same process but different GPIO pins. This was done to provide additional control signals in case they became necessary later on. The code for sending and receiving responses on the DE2-115 involved the use of built-in C functions like “getc”. When used in Nios II tools for Eclipse these functions access the serial buffer directly, making manual manipulation of the hardware registers unnecessary. The C language code for the DE2-115 UART communication is included in Appendix D at the end of this report.

Implementing UART on the Arduino Mega 2560 proved much simpler than the DE2-115. The 2560 has predesignated pins for use in UART, and default libraries that handle most of the logic. All that was necessary was to code properly using the provided C++ functions. This meant establishing a 19200 baud rate connection (matched by the baud rate set for the DE2-115 in Qsys). Once the baud rate serial connection was created, the UART output pin of the Arduino was attached to the UART input pin of the DE2-115, and the UART output pin of the DE2-115 was attached to the UART input pin of the Arduino. At this point code was written so that the Arduino would send 8 characters to the DE2-115, and the DE2-115 would respond by sending the same 8 characters back. The UART code for the Arduino is included in Appendices A, B, and C at the end of this report.

The settings for the serial connection were set to 8 data bits, 1 stop bit, with no parity. This was true for both the DE2-115 and the Mega 2560. The settings entered into Qsys for the UART RS-232 are shown in **Figure 7.1** below.

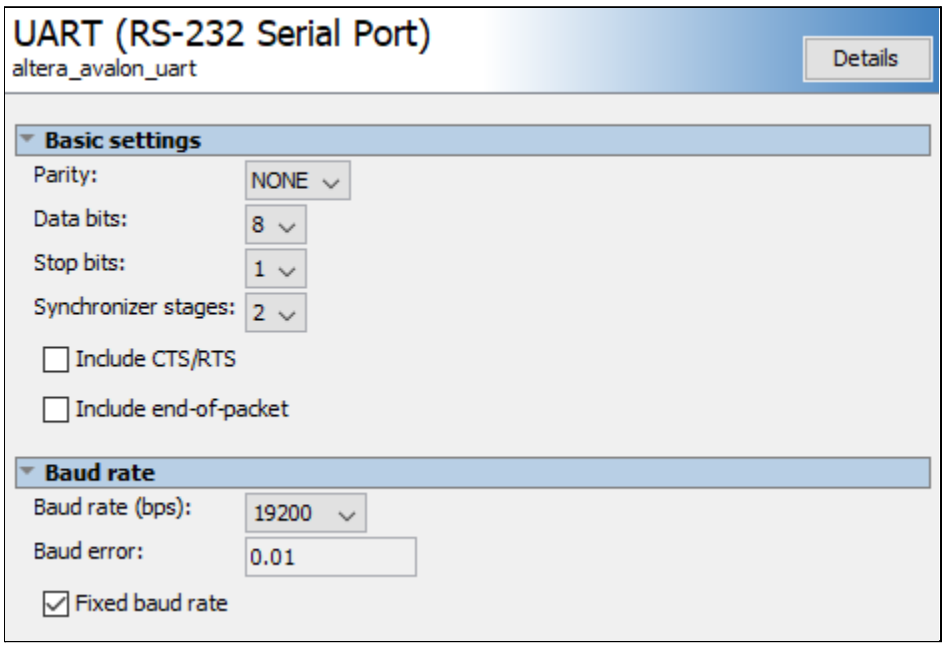


Figure 7.1: Qsys UART IP Core Settings

These same settings were the default for the Arduino, so no modifications were necessary.

7.2 Connecting DE2-115 to Serial SRAM using SPI

The second major milestone towards implementing a SRAM-PUF was connecting the DE2-115 to the 23K256 Serial SRAM chip. A number of steps were necessary to accomplish this. First the SPI(3-wire) IP Core was added to the Qsys design, the necessary connections were made, and the four signals were exported using the conduit field, as shown below in **Figure 7.2**.

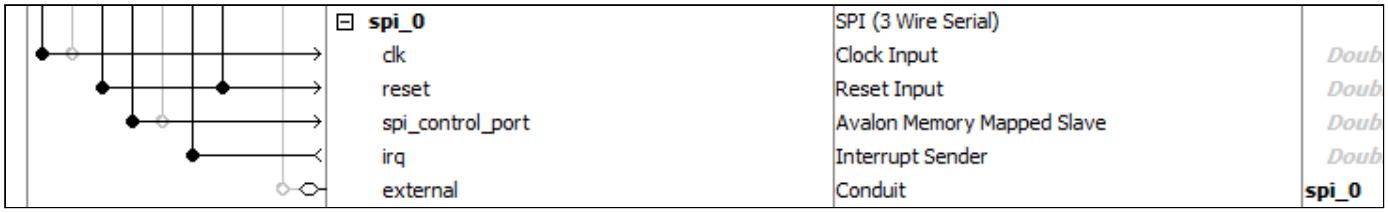


Figure 7.2: Qsys Configuration for SPI(3-Wire Serial)

The SPI component was configured to behave as a master, initiating transfers to the serial SRAM slave. The settings for the SPI component were set to match the 23K256 Serial SRAM specifications found in the 23K256 Datasheet [15]. The qsys settings are shown below in **Figure 7.3**.

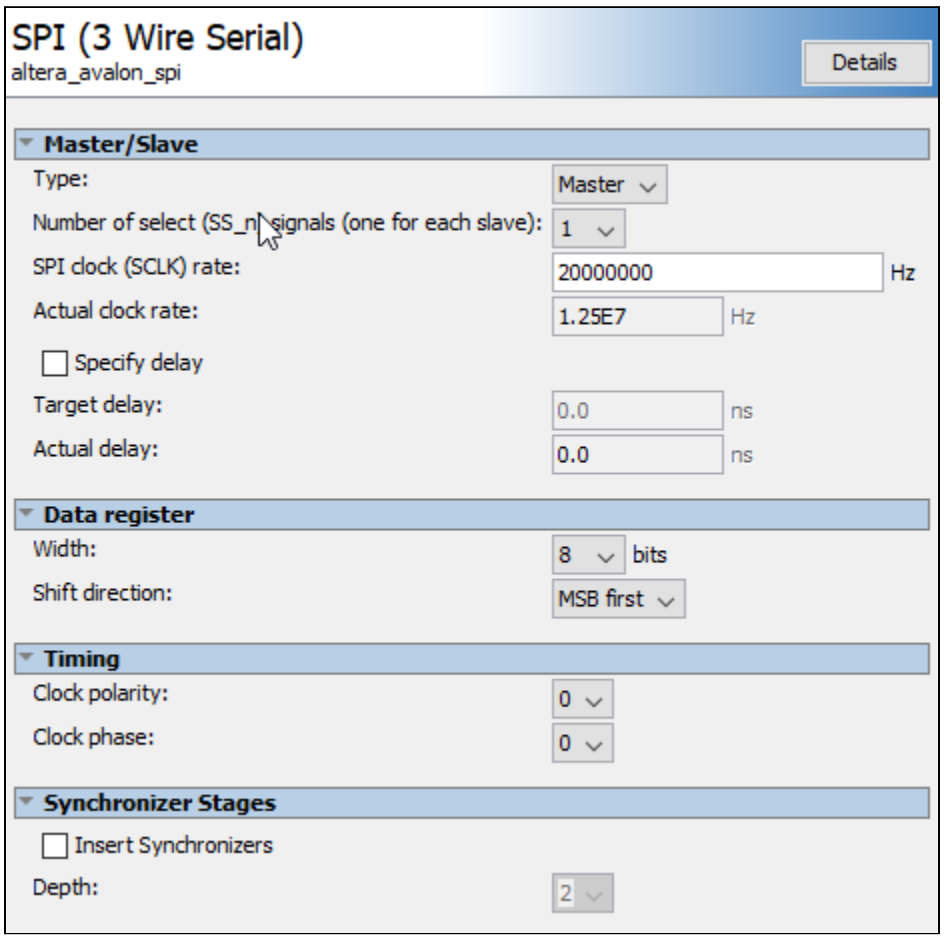


Figure 7.3: Qsys Settings for SPI(3-Wire Serial)

After generating the Verilog HDL files and recompiling the quartus project, it was possible to access the Serial SRAM in eclipse by using the Altera general SPI access function. However, this was not as simple as calling a single function. Sending and receiving instructions and data to and from the serial SRAM had to be done in a manner that followed the guidelines provided in the 23K256 datasheet.

For example, in order to read a byte from the SRAM, an 8-bit read instruction first had to be written to the SRAM. Then, without allowing the chip select signal to go high, the 16 bit address had to be sent in the form of two bytes, starting with the byte holding the most significant bits of the address.

In order to compare the signals being output to the SRAM with those described in the datasheet, the SignalTap II Logic Analyzer tool was utilized within Quartus. This allowed the signals being sent to the SRAM to be monitored, helping to quickly spot signals that didn't match those in the datasheet.

Figure 7.4 shows a datasheet-to-SignalTap comparison of a command being written to the SRAM. It's clear that the command is sent correctly because it matches the signals shown in the datasheet.

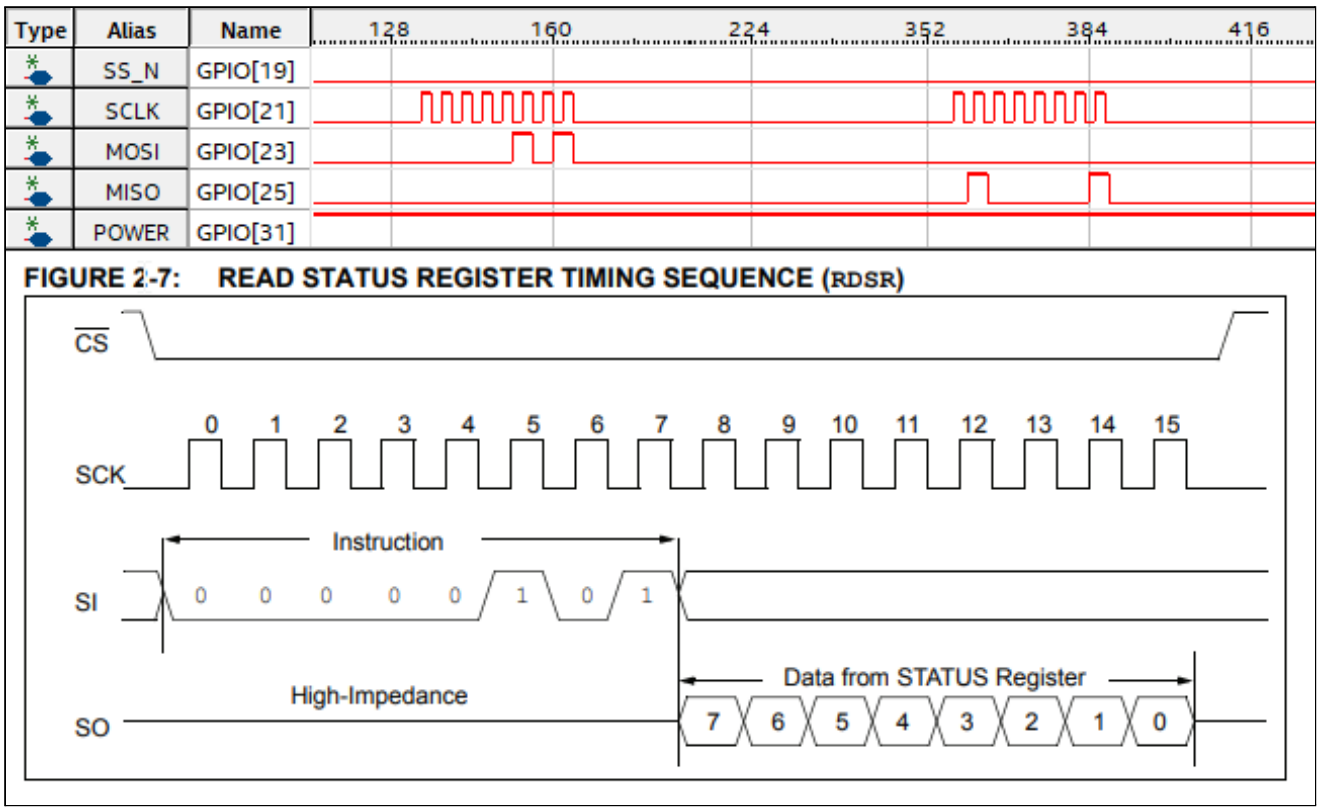


Figure 7.4: SignalTap vs 23K256 Datasheet[15]

In **Figure 7.4** the SignalTap signals are red, and the timing sequence for the command is shown below. A brief comparison reveals that they match up correctly, letting the designer know that he is sending this command exactly as the datasheet instructs. This is vital for debugging. Without a way to view the signals it would be impossibly tedious to determine any missteps when accessing the SRAM.

Because it is necessary to repeatedly power cycle the SRAM in an SRAM-PUF implementation, it is important to be aware of the power-on state of the SRAM and its signals. Section 2.7 of the datasheet addresses this, and is shown below in **Figure 7.5**.

2.7 Power-On State

The 23X256 powers on in the following state:

- The device is in low-power Standby mode ($\overline{CS} = 1$)
- A high-to-low-level transition on $\overline{CS}$ is required to enter active state

Figure 7.5: Power-On State Information in 23K256 Datasheet[15]

This information was used in tandem with the Altera SPI general control function in order to repeatedly power cycle the SRAM while retaining proper functionality for subsequent data readings.

The single greatest obstacle in correctly controlling the SRAM was voltage control of the signals. More specifically, it was very difficult to determine the correct order of commands to send to the SRAM in order to ensure that all signals were set to low. It was determined that even if the SRAM power signal was low, if any other signal was high, the sram memory bits would retain their values indefinitely(from what we could reasonably observe). It was eventually determined through repeated attempts and use of the SignalTap Logic Analyzer that all signals could be set low by performing the following steps in their listed order:

1. Power up the SRAM
2. Set the slave select signal to 1, the value for the SRAM chip
3. Force slave select signal to be sent out
4. Power off SRAM

After performing these four steps in order, it was found that all signals to the SRAM were consistently set to low. This allowed the short retention times expected of a SRAM PUF implementation.

7.3 Performing bit masking to increase SRAM startup-value reliability

One of the necessary steps prior to implementing the SRAM-PUF was determining the most stable bits in the memory. The idea was to determine the bits that are least likely to flip between startups. While there are many methods of accomplishing this, the method we chose for our design was bitmasking due to its simplicity and feasibility with limited project resources. A simple algorithm was designed in order to repeatedly power cycle the SRAM and collect data on any changes that occurred between the current startup values and the previous ones. Repeating this process a large number of times resulted in an accurate bitmask that showed the locations of all bit-flips after a given number of power cycles.

By using the bitmask to determine the most reliable bits prior to implementing the SRAM PUF, a lot of the noise in the startup-values resulting from random bit flips could be eliminated. This was desirable because it would help ensure that an observed increase in the PUF noise/unreliability was the result of SRAM temperature changes and not unpredictable bit-flips in the memory.

The algorithm for determining the bitflips for a given number of power cycles, in this case 100, is shown below in **Figure 7.6**.

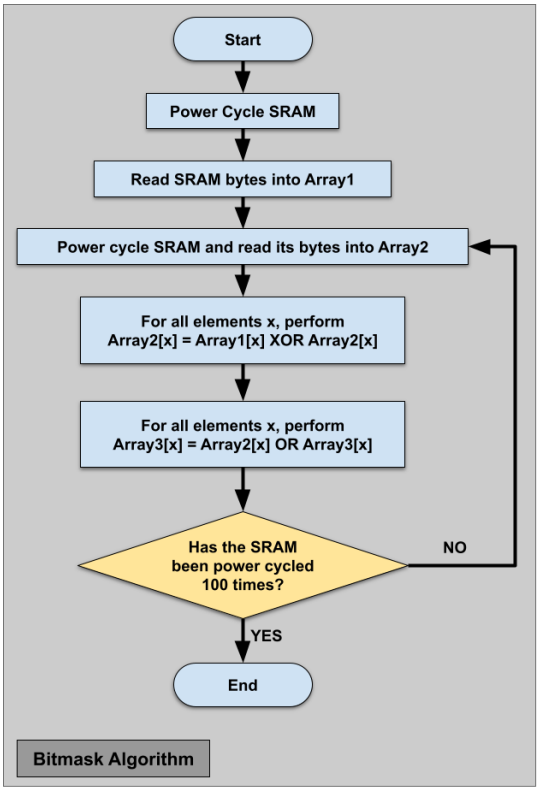


Figure 7.6: Simple Bitmasking Algorithm

The algorithm shown in **Figure 7.6** shows the steps taken in order to generate the bitmask. First, the SRAM is turned on and data is read from the memory and stored in an array. Second, the SRAM is power cycled and the new data is taken and stored in a second array. The next step is to XOR each pair of bits in these two arrays, producing a mask that shows the bitflips between the first and second SRAM reads. Lastly, a third array is OR'd with this mask in order to add the detected bitflips to the final output. This process repeats until a sufficient number of rounds have been completed. In general, more rounds will result in a more rigorous bitmask, since more bits will be detected as unreliable. An example of a short 10 byte bitmask generation using this algorithm is shown below in **Figure 7.7.**

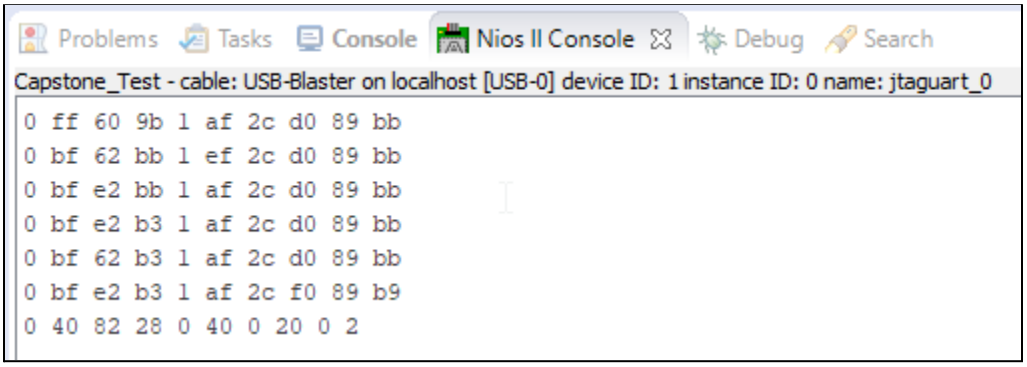


Figure 7.7: Simple Bitmasking Algorithm

The figure above shows a series of 10 byte reads from the SRAM memory. Between each read a power cycle is performed in order to acquire new start-up values, and the bitmasking algorithm gathers any differences between the reads, outputting them at the end of the 5 rounds. By aligning the columns of the data one can see that the final result accurately tracks the changes in the startup values, providing a mask that shows which bits are too unreliable for use in the SRAM-PUF.

At this point in the design of the PUF, a problem arised in regard to the time required to generate bitmask data. Generating accurate bitmask data took a fair amount of time since the SRAM had to be shut off for at least 100ms for each read. This may not seem like much time, but for 100 rounds of bit masking it amounts to 10 seconds, and for 1000 CRP pairs it amounts to about 2.8 hours. It was considered that a faster way to acquire the bitmask would be to create an exhaustive bitmask for the entire SRAM and store it along with program instructions on the DE2-115. This data could then be referenced at will without the need to generate it again during bitmask generation. A pre-generated bitmask was implemented for this reason following the same technique as before. The only difference was that the bitmasking would take place once and the mask would then be stored along with the program instructions on the DE2-115.

A few steps were performed in order to create and store the bitmask alongside the program instructions. First, the bitmask was generated and stored in a text document on an SD card using the DE2-115. Once the bitmask was properly stored on the SD card, the SD card data was extracted onto a computer. The text document data was then taken and converted into a c-language array in hexadecimal format, allowing it to be loaded to the DE2-115 along with the program instructions as a hard-coded array of bytes. A simple C++ program was coded in order to convert the unicode text document data to hexadecimal and format it in C language array format. The code for this program is included in **Appendix E** at the end of this report.

This change in the implementation required some debugging of the SD card interface on the DE2-115. After some difficulty implementing a correct write to the SD card, it was discovered that the problem was the formatting of the SD card, which could not be accomplished using the built in windows formatting tool. Instead, it was necessary to download and use standalone SD card format software from the SD Association website[23]. After using this program to format the SD card, the bitmask data written to the SD card properly resembled the expected bitmask values.

It is worth noting that other requirements also exist for any SD card used on the DE2-115. The card must be formatted in FAT16 using the aforementioned formatting program, and the SD card cannot be greater than 2 gigabytes in memory size. Also, the card type must be SD, not SDHC or any other card type. The design team was careful to utilize an SD card that met these requirements. One helpful

source of information used by the design team was the Altera SD Card Interface IP Core manual, which can be accessed from within a quartus 17.1 installation using a similar directory to that shown below.

C:/intelFPGA_lite/17.1/ip/altera/university_program/memory/altera_up_sd_card_avalon_interface/doc/SD_Card_Interface_for_SoPC_Builder.pdf

Once the SD card could be written to successfully a bitmask was generated for the entire SRAM memory. The time required to generate this bitmask depends on how accurate the bitmask is intended to be. To generate more accurate bitmasks, a larger number of SRAM power cycles are needed in order to detect a larger number of unreliable bits in the memory. For the purposes of this project the design team felt it was sufficient to generate the bitmask over 400 power cycles. The reasoning for this design decision lay in a simple calculation. Supposing that a bit had only a 0.5% chance of flipping each cycle, the chance of the bit not flipping would be 99.5%, or 0.995. Over 400 cycles this chance becomes 0.995^{400} which is equivalent to 0.13, or 13%. Thus, even with only a 0.5% chance to flip, after 400 power cycles there is a 87% chance that the bit will have flipped. For the purposes of this project the team felt this was adequate accuracy for the bitmask.

The time requirement for the stated 400 power cycle bitmask generation proved to be roughly 25 minutes. In order to achieve this speed it was necessary to disable all JTAG printing to the console. This meant commenting out all “printf” debugging usages in the bitmap generation code. The team found that repeated “printf” usage slowed down bitmap processing time per power cycle from 7 seconds to 38 seconds, or by a factor of around 5.43. Thus it was very beneficial to temporarily disable all debugging information printed to the console while performing the bitmask processing.

7.4 Voltage Variance Noise Mitigation

Prior to implementing the SRAM-PUF an attempt was made to decrease noise resulting from variations in voltage between power cycles of the PUF. A voltage regulator was placed between the PUF and the DE2-115 power output pin. A 100 byte bitmask was then generated five times with the voltage regulator, and an additional five times without. The results were then compared in order to determine if adding a voltage regulator helped to increase reliability. The results are shown below in **Figure 7.8.**

	Voltage Regulator					**No Voltage Regulator**				
Reliable Bit %	**89.5**	**89.25**	**89.375**	**89.375**	**89.375**	**90**	**91**	**90.625**	**90**	**90.125**
Average %	**89.275**					**90.35**				

Figure 7.8: Voltage Regulator Reliability Comparison

The results indicated that the regulator was irrelevant, perhaps even slightly detrimental. The cause may have been that the voltage output of the DE2-115 pins provided consistency that could not be improved by adding the voltage regulator. Assuming that this was the case, any measured differences in reliability would have been the result of random bit flips. After making the determination that the voltage regulator would offer no significant increase in reliability, it was decided that it would be left out of the final design.

7.5 Implementing SRAM-PUF Response Generation

One of the most important steps in the implementation was implementing the response generation of the PUF. Once the SPI SRAM was connected and the reliability bitmask data could be readily obtained, the next step was to code the PUF to take in challenges and generate responses. To accomplish this, a "Generate_Response" function was created in the C-language project code in Eclipse. This function can be found in **Appendix D**. The steps taken to generate a response were as follows:

1. Receive a challenge in the DE2-115 from the Arduino using serial UART
2. Derive a memory address in range of the 32768 byte SRAM from the challenge
3. Create a bitmask at this address for use eliminating unreliable bits from the function
4. XOR the SRAM of the address with the challenge to produce a response
5. OR this response with the corresponding mask bytes to set all unreliable bits to 1
6. Transmit the response data back to the Arduino for database storage/comparison

In step 1 a memory address was generated from the received challenge data. Some manipulations were first applied to the challenge data, splitting the bytes into groups of upper and lower bits. These bit groups were then concatenated into two large numbers. These large numbers were then XORed, producing a single large number. The modulo operation was then used to ensure that this single large number was not larger than the maximum size of the SRAM memory in bytes, 32,768. This resulted in a value that could be used as a memory address when pulling data from the PUF.

In step 2 a reliability bitmask was generated for the SRAM address derived in step 1.

In step 3 the SRAM bytes beginning at the derived address were XORed with the corresponding bytes in the received challenge. This step involved the SRAM data in the response generation, which in turn causes the response to rely on the physical characteristics of the SRAM chip.

In step 4 the response data resulting from step 3 was ORed with the corresponding reliability mask bytes, effectively setting any bits sensed as unreliable to 1. This ensured reliability of those unreliable bits, since they would now be 1 without fail.

Step 5 simply involved responding to the challenge from the Arduino by sending the generated response data to the Arduino using the serial UART connection established previously.

The generation of two CRP pairs can be seen on the Arduino serial monitor in **Figure 7.9** below.

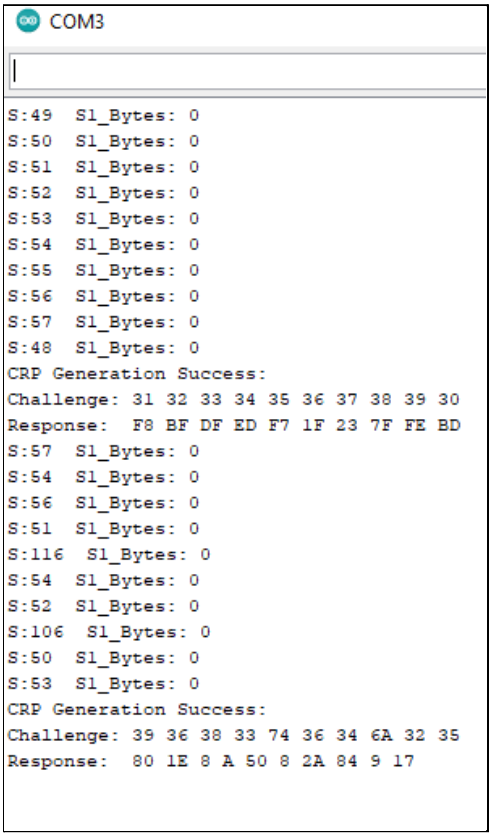


Figure 7.9: Challenge-Response Pair Generation

In the example shown in **Figure 7.9** two different challenges are sent to the DE2-115 through the UART connection, and both result in a unique response that was generated and transmitted by the DE2-115.

7.6 Creating an Arduino SD-Card Memory Database to Store Challenges and Responses

Once the SRAM was fully implemented the next step was to create a database to store the CRP pairs. This was done by using the arduino and its ethernet shield to interface with a SD card. The correct responses had to be stored in non-volatile memory in order to check the correctness of incoming unverified responses from the PUF. To accomplish this the Arduino SD library was used along with its related functions. Setting up the SD card to be read on the Arduino Mega was more difficult than was expected. A number of errors were encountered until it was understood that the same SPI connections

used to access the SD card in the ethernet shield are also used to access the shield's ethernet port. This meant that in order for the chip select to be able to consistently target the SD card port for activation, the ethernet port chip select had to be forced into an inactive high state. In short, the chip selects for both the ethernet port and the SD card port had to be initialized as outputs and set to a default HIGH signal, rendering their selection inactive. This was accomplished using the code shown below in **Figure 7.10**.

```
pinMode(10, OUTPUT);//set ethernet chipselect pin to output mode
digitalWrite(10, HIGH);//set ethernet chipselect pin to high by default
pinMode(4, OUTPUT);//set chipselect pin to output mode
digitalWrite(4, HIGH);//set chipselect pin to high by default
```

Figure 7.10: SD Card Chip Select Setup Code

In order to lay a firm foundation for debugging, a considerable number of checks were run in the setup code as well. These checks ensured that the card was inserted, of the correct type, and initialized successfully. The feedback from these debugging checks provided key information that allowed problems in the SD card initialization to be fixed, resulting in consistent access to the SD card. This debugging code is included in **Appendix B** at the end of this report.

Once the SD card could be accessed properly, code was written to read challenges from a file on the SD Card, send the challenge to the DE2-115, receive a response from the DE2-115, and store the response in a second file on the SD card. It is worth noting that the SD library allowed opening two files simultaneously, making repeated opening and closing of files on the SD Card unnecessary. The results of the SD card debug checks are shown below in **Figure 7.11**.

```
Wiring is correct and a card is present.
Card type: SDHC

Files found on the card (name, date and size in bytes):
SYSTEM~1/      2021-12-31 09:27:08
  WPSETT~1.DAT  2021-12-31 09:27:08 12
  INDEXE~1      2022-03-06 11:43:20 76
LIST2.TXT      2000-01-01 01:00:00 11
LIST1.TXT      2022-03-06 11:48:20 10000
initialization done.
O==-- Database files opened successfully --==O
50 99 56 56 50 50 57 48 13 10
```

Figure 7.11: Arduino Debug Check Console Results

Some of this debug code was also used to check and ensure that the CRP's were being stored in the text file correctly. This is shown in **Figure 7.12** below.

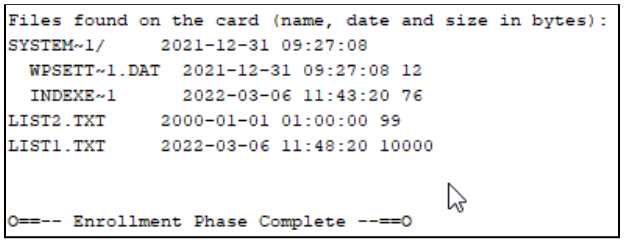


Figure 7.12: Checking Response Writes to File

In **Figure 7.12** the file contents of the SD card are listed in the Arduino console. The "LIST2.TXT" file is the text document used to store responses to challenges. The number of bytes stored in this file is shown to the right of the file name as 99. This is the anticipated result because 9 keys consisting of 10 bytes were stored, in addition to 9 newline characters. The sum of the memory stored in the file is therefore 9 x 10 + 9 characters, or 99 bytes. This shows that the responses were written to the file successfully. The design team also checked the micro SD card manually using a card reader, hex editor, and computer. The results are shown below in **Figure 7.13**, alongside the same response as it was printed in the console.

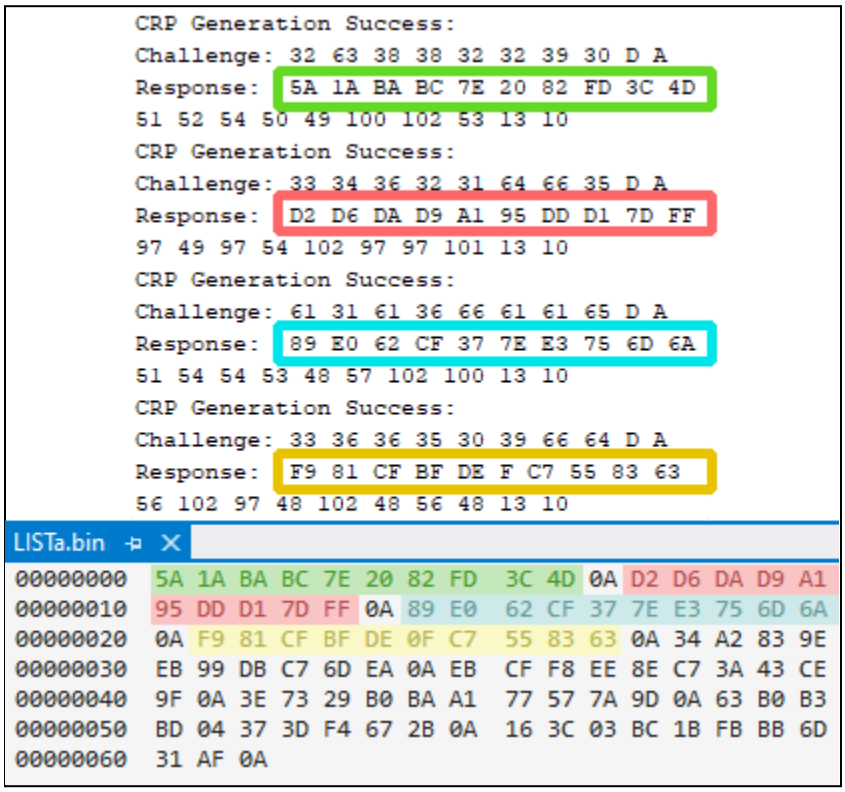


Figure 7.13: Manual Confirmation of Correct Response Storage in SD Card

The upper portion of **Figure 7.13** shows the responses that were received from the DE2-115, and the bottom half shows the SD card file contents opened in a hex editor. It is clear that the received

response matches the responses stored on the SD card perfectly, evidenced by the color matched responses in the figure. The design team also noted the repeated 0A data values, which represent the ASCII newline character. After confirming that responses were properly stored, the next step was to complete the enrollment phase by storing a sufficient number of responses at room temperature.

7.7 PUF Enrollment Phase at Room Temperature

The next stage of development was sending challenges, generating PUF responses at room temperature, and storing them in the database. This was successfully accomplished using the entirety of the enrollment phase Arduino code included in **Appendix A**. The number of responses to be stored was set to 900, and since 1000 challenges were stored manually in the challenge storage text file, there were more than enough challenges available to generate 900 responses. In addition to checking the data manually using an SD card reader, it could also be seen from printed debug information that the expected number of bytes had been stored as responses in the list3.txt file on the SD card. This debug information is demonstrated below in **Figure 7.14.**

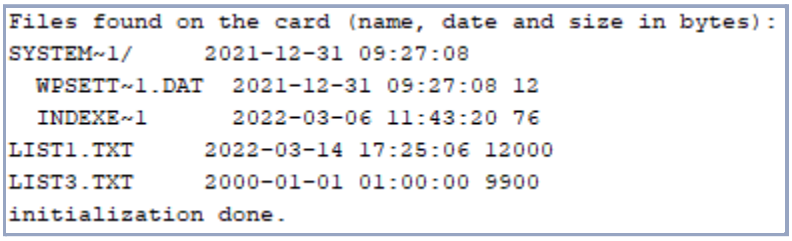


Figure 7.14: Enrollment Phase CRP Storage Confirmation

The information listed for the LIST1 and LIST3 text files in the above figure shows that LIST1 contains 12000 bytes and LIST3 contains 9900 bytes. When writing responses to LIST3 a single newline character was used for each response, and 10 characters were used for each response. Thus, (10+1)*900 = 9900 bytes. For LIST1, the challenge storage, 1000 challenges were placed in the file, and each challenge included newline and carriage return characters. Thus, (10+2)*1000 = 12000 bytes. To summarize, the memory usage in the files matched perfectly with what was expected, showing that the enrollment phase was performed successfully. After completing the enrollment phase, the next step was to implement the correlation phase, where PUF reliability metrics would be obtained at varying temperatures.

7.8 PUF Correlation Phase

The first step in implementing the Correlation phase was to repeatedly generate responses from the PUF and compare them with the responses stored in the database. Modifications were made to the enrollment phase code in **Appendix A** in order to produce the correlation phase code in **Appendix B**.

The major difference was that rather than storing the responses in the SD card database, the newly generated responses were instead compared with those already present in the database(as a result of enrollment). Once this modification was complete a simple test was run to ensure that the reliability ratio was being generated correctly. The results are shown below in **Figure 7.15**.

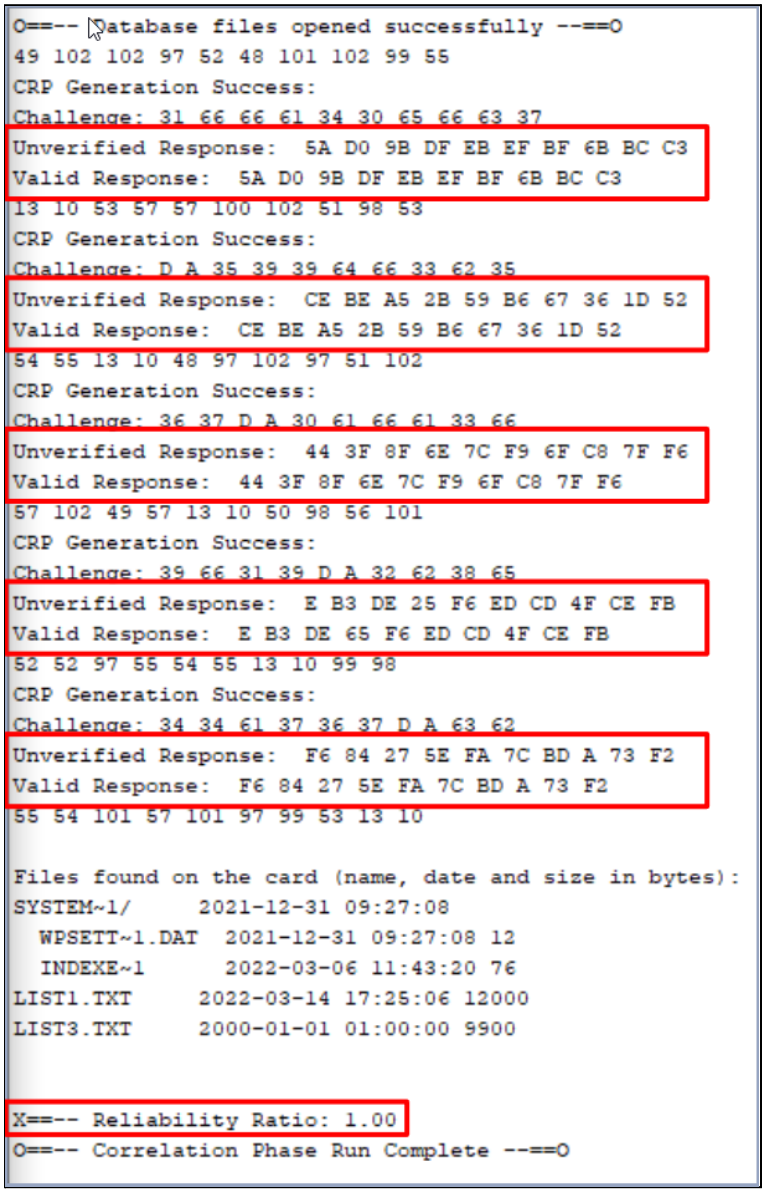


Figure 7.15: Correlation Phase Reliability Ratio Generation

For the simple correlation phase test shown in **Figure 7.15**, the generated responses are shown next to the valid responses. For this particular test of 5 CRPs, every response matched the one stored in the database, resulting in a reliability ratio of 1. This test was performed at room temperature, approximately the same temperature used to store the database responses, so it was expected that the

reliability ratio would be very high. Further tests were also performed using a much larger number of CRP's, and the results are shown below in **Figure 7.16**.

Reliability testing at 22.3℃ with a bitmask generated at 22.2℃ and database CRP's generated at 22.7℃		
Run #	# of CRP's Tested	Reliability Rate
1	900	896
2	900	894

Figure 7.16: Reliability Ratio Test Results at Room Temperature

The results of the tests confirmed the expectations of the design team. Given the closeness in temperature the reliability would be fairly high, which is exactly what was observed. The next step was to increment the SRAM chip temperature by 1℃ and reobtain the reliability rate. This process would then be continued until the temperature had surpassed 60,℃, the max recommended lithium ion battery temperature, by 10℃. Thus, the highest temperature that the SRAM chip was heated to was 70℃. The reliability ratio was calculated at 1℃ increments from 22℃ to 70℃.

One important step prior to actually performing the repeated reliability calculations at varying temperatures was making the response generation and collection as quick as possible. The reason it needs to be fast is that if reliability measurement takes too long the temperature may fluctuate during the measurement, resulting in inaccurate sensed temperatures. A number of steps were taken to ensure that the process was occurring as quickly as possible. The first was to lower the number of queried CRP's from 900 to 450, which saved time processing those additional 450 CRP's for each temperature measurement. Higher baud rates were also used to increase the speed of CRP data transmission, but problems arose for some of the computers used in development at higher baud rates (specifically windows 11 operating system computers), so it was decided to continue using 9600 for consistent functionality across computers.

Following the test above some usable correlation data was obtained by heating the SRAM chip using a heat gun and observing the corresponding reliability changes. This is shown below in **Figure 7.17**.

Temperature	Reliability Rate 1	Reliability Rate 2	Reliability Rate 3	Reliability Rate 4	Reliability Rate 5	Average Reliability
34	0.9533	0.9489	0.9556	N/A	N/A	0.9526
50	0.9333	0.9156	0.8711	0.92	0.8844	0.90488
60	0.89164	0.8933	0.8822	0.8956	0.8911	0.890768

Figure 7.17: Useable Temperature vs Reliability Data Obtained

Using this data a graph was then created to demonstrate that the obtained data matches the expectations based on the inverse relationship between SRAM PUF reliability and temperature. This graph is shown below in **Figure 7.18.**

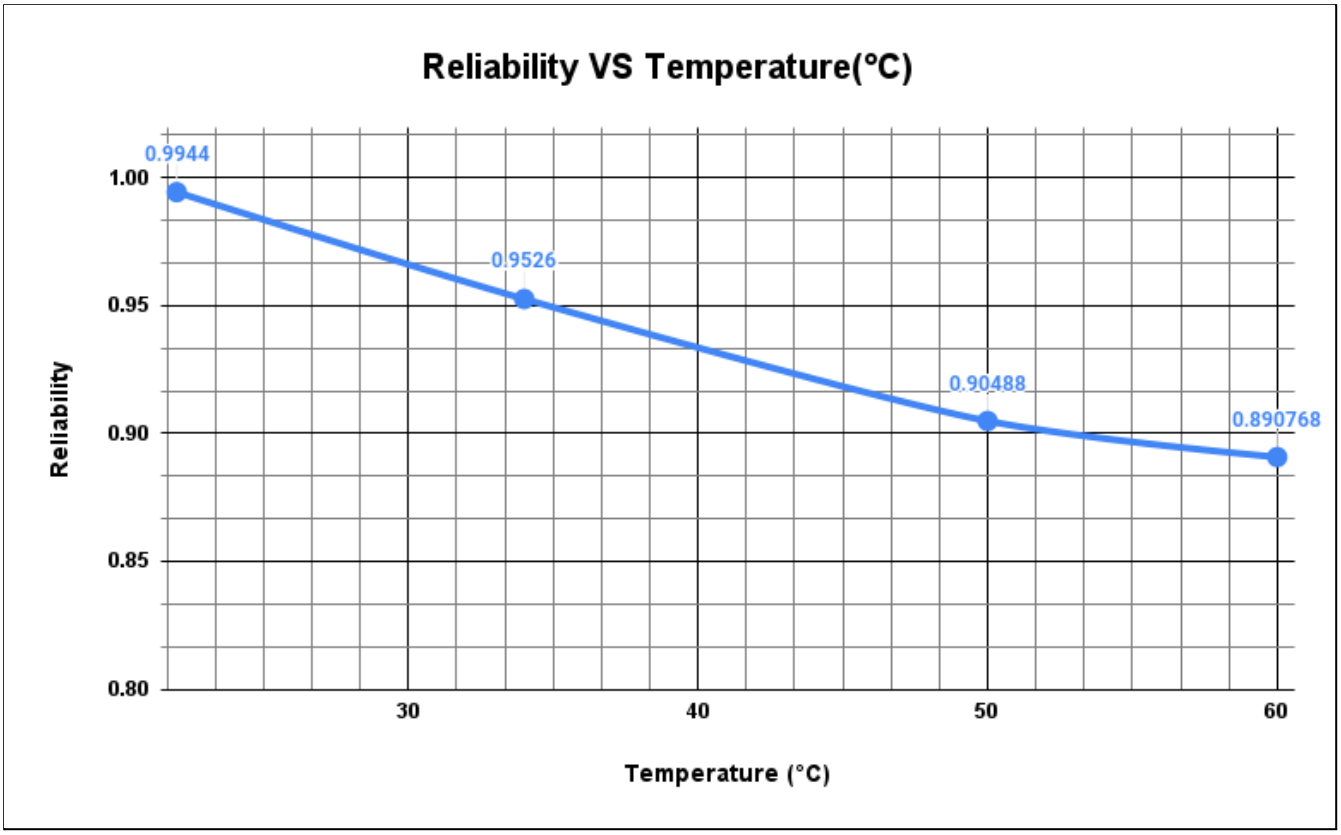


Figure 7.18: Useable Temperature vs Reliability Data Obtained

The graph in **Figure 7.18** demonstrates how the reliability of the SRAM PUF decreased as the SRAM chip was heated using the heat gun. Thus the reliability ratios corresponding to the threshold temperatures of 50℃ and 60℃ were 0.90488 and 0.890768 respectively. Having obtained these values it was now possible to sense temperature by repeatedly polling the SRAM for responses, checking them against the correct responses stored in the database, and comparing the resulting reliability ratio with our determined cutoff values.

7.9 PUF Sensing Phase

In order to implement temperature sensing the correlation phase code in **Appendix B** was modified, resulting in the sensing phase code in **Appendix C**. The single biggest difference in the code was that for temperature sensing we expect to continuously obtain reliability ratios until we reach threshold temperatures. Thus the code had to be modified to facilitate continuous reliability generations.

The code therefore works in an endless loop. Each time all 450 CRP's are checked a reliability ratio is calculated and compared with our threshold reliabilities. If the calculated reliability is below either of the threshold reliabilities, which were 0.90488 and 0.890768, then the code branches into the associated safety feature functionality.

7.10 Output to Vibration Motors

From the Arduino board, pins 4 and 5 were connected to a HIGH and LOW on the L298N Motor Drive Controller. The Motor Drive Controller was also powered from a 5V pin on the Arduino board. The vibration motor was then connected to one of the channels on the controller. In order for the vibration motor to be initially set to off and only be activated when a warning was triggered, adjustments were made within the code. Rather than having a HIGH and LOW in the code, both were set to HIGH.

7.11 Output to Digital Speaker

As part of the threshold temperature safety features a Digital Speaker was used to generate a tone by changing the input between low and high signals with a certain frequency. The positive terminal was connected to a pin on the Arduino board and negative terminal to ground.

7.12 Android App Bluetooth Warning

On the Arduino a bluetooth adapter was connected so that when the temperature threshold is triggered it will send a message to an app stating a warning that temperature levels are rising. The way this was connected to the Arduino was by using the TX3 and RX3 for communication and the 5V and GND for power for the connections from the Arduino to the bluetooth itself. Then on a wireless phone the app was opened "Serial Open Terminal" to be able to connect the bluetooth signal for communication. Within the code of the Arduino a new serial communication was added so that the message is sent via bluetooth. The output for the bluetooth was set up with pinMode 2, so now the way it was added to the code that once reliability ratio has trigger it then the message that is sent to the app is "Warning - Vehicle Battery Overheating" with the following message if the last threshold is triggered "Please Pull Over!"

7.13 HC-05 Bluetooth Module Secure Connection

One of the initial goals of the project was to secure the Bluetooth connection between the driver's android phone and the temperature monitoring system. The initial plan was for this security to be implemented at the application layer, but due to time constraints the team instead opted to use the HC-05 Bluetooth Module's built-in encryption and authentication. A number of steps were required to configure the security settings for the HC-05 Bluetooth Module. First, the module had to be turned on

in AT command mode. This was accomplished by first connecting all the HC-05 signals except for VCC and Key. The team then applied a high signal to the KEY(alternatively labeled ENABLE, or EN) pin while simultaneously pressing the small button above the KEY pin with a pen. While this was still being done a high signal was sent to the VCC pin, resulting in the HC-05 module starting in AT command mode. The Arduino MEGA 2560 was then coded to communicate with the HC-05 serially, and the "AT" command was sent to the HC-05 from the serial monitor console in the Arduino IDE. The result was that the HC-05 responded with "OK", indicating that it was successfully operating in AT command mode. Once it was certain that the HC-05 was in AT command mode, the "AT+SENM=3, 2" command was sent to the HC-05. This command set the security modes of the HC-05 to Link safety mode, enabled authentication, and set HCI encryption to both point-to-point and broadcast. The password for connecting was left to the default of 1234, and the bluetooth module security was tested by connecting securely using an android phone. The password had to be entered correctly and the bluetooth connection initialized securely before any information could be received on the android phone. These steps resulted in the HC-05 being set to Security Mode 3. According to [23], "Security Mode 3 is the link level-enforced security mode, in which a Bluetooth device initiates security procedures before the physical link is fully established. Bluetooth devices operating in Security Mode 3 mandate authentication and encryption for all connections to and from the device. Therefore, even service discovery cannot be performed until after authentication, encryption, and authorization have been performed." This is the highest security level available for Bluetooth 2.0 devices like the HC-05.

8. CONCLUSION

At the time that the design team concluded their work on the project nearly all of the milestones determined in the first semester design were completed. The design team successfully used the SRAM PUF to sense temperature thresholds by monitoring changes in the reliability of the PUF response. Safety features were also triggered at the threshold temperature, such as motors for vibrating the steering wheel and a warning message sent to the driver's android phone using Bluetooth. The PUF response reliability was increased significantly using bitmasking, making authentication much more consistent, and the bluetooth connection to the android phone was secured to the extent it could be given time constraints.

Despite the considerable amount of work completed, there are still numerous ways that the SRAM PUF temperature monitoring system could be improved. A method for sorting between temperature dependent CRP's and temperature independent CRP's would be useful, since it would allow the CRP's that are reliable across varying temperatures to be used for authentication and those that vary in reliability across changing temperatures to be used for temperature sensing. This would serve to increase the reliability of the SRAM PUF authentication. Bluetooth security implemented at the application layer may also provide more robust security between the android phone and the temperature

monitoring system. Techniques could also be applied to protect the SRAM chip itself against extreme low temperature data-retention based attacks.

Overall the design team was able to accomplish the majority of the design goals, and the expected result that the system would be more resistant to spoofing and physical attacks while also providing safety features typical of temperature sensors was successfully achieved.

9. CONTRIBUTIONS

The following is a list of contributions made by the team members working on this project.

Trevor Ridley:

- Created/presented slides 3, 6, and 7 in mid-semester presentation
- Edited enrollment Flow Chart Based on 11/5/2021 Feedback
- Wrote report abstract
- Designed Flow Chart
- Researched and wrote parts of literature review
- Worked on theoretical background
- Worked on general format of report
- Worked on reference formatting
- Updated block diagram with signal bit widths
- Edited Sensing & Correlation Flow Charts Based on 11/8/2021 Feedback
- Tested UART serial comms and DHT11 temperature sensor on Arduino MEGA 2560
- Tested UART serial comms and Voltage Regulator on DE2-115
- Implemented UART serial trading of CRPs between DE2-115 and Arduino
- Implemented SPI between the DE2-115 and the 23K256 Serial SRAM
- Handled the laser thermometer during correlation phase data acquisition

Julius Andrade:

- Created/presented slides 2 and 4 in mid-semester presentation
- Researched and wrote parts of literature review
- Edited Sensing & Correlation Flow Charts Based on 11/5/2021 Feedback
- Tested running server on Arduino
- Tested setting up simple webpage in HTML w/ Ethernet Shield 2
- Worked on theoretical background
- Worked on general format of report
- Worked on reference formatting
- Handled the heat gun during correlation phase data acquisition

Cristian Luna:

- Created/presented slides 5 and 8 in mid-semester presentation
- Researched and wrote parts of literature review
- Worked on theoretical background
- Worked on general format of report
- Worked on reference formatting
- Tested DHT11 temperature sensor on Arduino MEGA 2560
- Implemented vibration motor using Arduino MEGA 2560 & Controller
- Tested vibration motor w/ Arduino MEGA 2560 & w/ Motor Drive Controller
- Handled Arduino and DE2-115 software during correlation phase data acquisition

10. REFERENCES

[1] H. Ma, Y. Gao, O. Kavehei and D. C. Ranasinghe, "A PUF sensor: Securing physical measurements," *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2017, pp. 648-653, doi: 10.1109/PERCOMW.2017.7917639.

[2] Y. Cao, Y. Guo, B. Liu, W. Ge, M. Zhu and C. Chang, "A Fully Digital Physical Unclonable Function Based Temperature Sensor for Secure Remote Sensing," *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, 2018, pp. 1-8, doi: 10.1109/ICCCN.2018.8487347.

[3] S. Chen, B. Li, and Y. Cao, “Intrinsic Physical Unclonable Function (PUF) Sensors in Commodity Devices,” *Sensors*, vol. 19, no. 11, p. 2428, May 2019.

[4] C. Labrado, H. Thapliyal, S. Prowell, and T. Kuruganti, “Use of Thermistor Temperature Sensors for Cyber-Physical System Security,” *Sensors*, vol. 19, no. 18, p. 3905, 2019.

[5] Y. Gao, H. Ma, D. Abbott and S. F. Al-Sarawi, "PUF Sensor: Exploiting PUF Unreliability for Secure Wireless Sensing," in *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 64, no. 9, pp. 2532-2543, Sept. 2017, doi: 10.1109/TCSI.2017.2695228.

[6] S. Gujja, “Temperature variation effects on asynchronous PUF design using FPGAs,” *Ohiolink.edu*. [Online]. Available: https://etd.ohiolink.edu/apexprod/rws_etd/send_file/send?accession=toledo1417716527&disposition=inline. [Accessed: 17-Oct-2021].

[7] S. Sutar, A. Raha, and V. Raghunathan, “D-PUF: An intrinsically reconfigurable DRAM PUF for device authentication in embedded systems,” in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems - CASES ’16*, 2016.

[8] Y. Takeuchi *et al.*, “SRAM failure injection to a vehicle ECU and its behavior evaluation,” *Kobe-u.ac.jp*. [Online]. Available: https://www28.cs.kobe-u.ac.jp/wp-content/uploads/2017/03/1303_takeuchi_RIIF-2.pdf. [Accessed: 17-Oct-2021].

[9] J. Yang, Z. Duan, M. Wang, J. Mahmood, Y. Xiao, and Y. Yang, “An authentication mechanism for autonomous vehicle ECU utilizing a novel slice-based PUF design,” *Journal of New Media*, vol. 2, no. 4, pp. 157–165, 2020.

[10] U. Chatterjee, S. Chatterjee, D. Mukhopadhyay, and R. S. Chakraborty, “Machine Learning assisted PUF calibration for trustworthy proof of sensor data in IoT,” *ACM Trans. Des. Automat. Electron. Syst.*, vol. 25, no. 4, pp. 1–21, 2020.

[11] “Automotive Car Server/Communication Gateway supporting new ...” [Online]. Available: https://www.renesas.com/us/en/document/whp/automotive-car-servercommunication-gateway-supporting-new-ee-architecture?language=en. [Accessed: 13-Oct-2021].

[12] S. Ma et al., “Temperature effect and thermal impact in lithium-ion batteries: A review,” Progress in Natural Science: Materials International, vol. 28, no. 6. Elsevier BV, pp. 653–666, Dec. 2018. doi: 10.1016/j.pnsc.2018.11.002.

[13] M. El-Said, “ECU Counterfeit Mitigation Using Holistic Approach in Modern Automotive Echo System,” 2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC). IEEE, Jan. 2020. doi: 10.1109/ccnc46108.2020.9045510.

[14] M. Cortez, A. Dargar, S. Hamdioui, and G.-J. Schrijen, “Modeling SRAM start-up behavior for Physical Unclonable Functions,” 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). IEEE, Oct. 2012. doi: 10.1109/dft.2012.6378190.

[15] Microchip, “256K SPI Bus Low-Power Serial SRAM” 23A256/23K256 datasheet, 2011. Available: http://ww1.microchip.com/downloads/en/devicedoc/22100f.pdf

[16] F. Farha, H. Ning, K. Ali, L. Chen, and C. Nugent, “SRAM-PUF-Based Entities Authentication Scheme for Resource-Constrained IoT Devices,” IEEE Internet of Things Journal, vol. 8, no. 7.

Institute of Electrical and Electronics Engineers (IEEE), pp. 5904–5913, Apr. 01, 2021. doi: 10.1109/jiot.2020.3032518.

[17] M. Liu, C. Zhou, Q. Tang, K. K. Parhi, and C. H. Kim, "A data remanence based approach to generate 100% stable keys from an SRAM physical unclonable function," 2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED). IEEE, Jul. 2017. doi: 10.1109/islped.2017.8009192.

[18] A. Babaei and G. Schiele, "Physical Unclonable Functions in the Internet of Things: State of the Art and Open Challenges," Sensors, vol. 19, no. 14. MDPI AG, p. 3208, Jul. 21, 2019. doi: 10.3390/s19143208.

[19] W. Wang, U. Guin, and A. Singh, "Aging-Resilient SRAM-based True Random Number Generator for Lightweight Devices," Journal of Electronic Testing, vol. 36, no. 3. Springer Science and Business Media LLC, pp. 301–311, May 31, 2020. doi: 10.1007/s10836-020-05881-6.

[20] "SD Memory Card Formatter for Windows Download: SD association," *SD Association | The SD Association*, 17-Mar-2021. [Online]. Available: https://www.sdcard.org/downloads/formatter/sd-memory-card-formatter-for-windows-download/. [Accessed: 09-Mar-2022].

[21] C. Cakir, M. Bhargava, and K. Mai, "6T SRAM and 3T DRAM data retention and remanence characterization in 65nm bulk CMOS," Proceedings of the IEEE 2012 Custom Integrated Circuits Conference. IEEE, Sep. 2012. doi: 10.1109/cicc.2012.6330672.

[22] "What is bluetooth and how do I use it?," *Samsung uk*, 18-Dec-2020. [Online]. Available: https://www.samsung.com/uk/support/mobile-devices/what-is-bluetooth/. [Accessed: 08-May-2022].

[23] J. Padgette et al., "Guide to Bluetooth Security," National Institute of Standards and Technology, Jan. 2022. doi: 10.6028/nist.sp.800-121r2-upd1.

11. APPENDIX A

```
////////////////////////////////////////////////////////////////////////////////////////////////
//Arduino Mega 2560 PUF Temp Sensor Project Code ----- ENROLLMENT PHASE
//
//This Code is for the ENROLLMENT PHASE, where the CRP's are stored in the SD Card database.
//The code assumes that a "list1.txt" file is on the SD Card in the Ethernet Shield 5100 slot.
//The code also assumes that a "list2.exe" file is on this same SD Card.
//The list1 file should hold 1000 unique challenges.  Corresponding responses are written to list2.
//
//Note that the baud rate for the serial connection must match that of the DE2-115 UART IP Core
//If serial connection is not working this would be a good thing to check
//
////////////////////////////////////////////////////////////////////////////////////////////////
//Include Statements
////////////////////////////////////////////////////////////////////////////////////////////////
#include <SPI.h>
#include <SD.h>
//Declarations
bool WaitingOnResponse = false;
File file1;
File file2;
const int chipSelect = 4;//Set correct chipselect
Sd2Card card;
SdVolume volume;
SdFile root;
unsigned char challenge[10];//in the future we have to get the challenge from a server database
unsigned char response[10];
unsigned char incomingByte = 0;
int chalcount = 0;
int x = 0;
int CRP_Gens = 0;
////////////////////////////////////////////////////////////////////////////////////////////////
int CRP_Gen_Num = 900;//HERE WE SET THE NUMBER OF CRP's TO BE STORED IN THE
DATABASE(responses,really)
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
  pinMode(52, OUTPUT);
  pinMode(53, INPUT);
  pinMode(10, OUTPUT);//set ethernet chipselect pin to output mode
  digitalWrite(10, HIGH);//set ethernet chipselect pin to high by default
  pinMode(4, OUTPUT);//set chipselect pin to output mode
```

```
digitalWrite(4, HIGH);//set chipselect pin to high by default
while (Serial1.available() > 0) { //Clear the junk in the FIFO
  char t = Serial1.read();
}
////////////////////////////////////////////////////////////////////////////////////////////////
//////////////////////////////SD Card setup/debug, Proceed to line 103 to skip//////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
if (!card.init(SPI_HALF_SPEED, chipSelect)) {
  Serial.println("initialization failed. Things to check:");
  Serial.println("* is a card inserted?");
  Serial.println("* is your wiring correct?");
  Serial.println("* did you change the chipSelect pin to match your shield or module?");
  while (1);
} else {
  Serial.println("Wiring is correct and a card is present.");
}
// print the type of card
Serial.print("Card type: ");
switch (card.type()) {
  case SD_CARD_TYPE_SD1:
    Serial.println("SD1");
    break;
  case SD_CARD_TYPE_SD2:
    Serial.println("SD2");
    break;
  case SD_CARD_TYPE_SDHC:
    Serial.println("SDHC");
    break;
  default:
    Serial.println("Unknown");
}
if (!volume.init(card)) {
  Serial.println("Could not find FAT16/FAT32 partition.\nMake sure you've formatted the card");
  return;
}
Serial.println("\nFiles found on the card (name, date and size in bytes): ");
root.openRoot(volume);
root.ls(LS_R | LS_DATE | LS_SIZE);
if (!SD.begin(4)) {
  Serial.println("initialization failed!");
  while (1);
}
Serial.println("initialization done.");
/////////////////////////////////////////////Open both text files/////////////////////////////////////
SD.remove("list2.txt");//erase any leftove data inside the text doc
```

```
  file1 = SD.open("list1.txt", FILE_READ);//Open challenge list for reading
  file2 = SD.open("list2.txt", FILE_WRITE);//Open response list for writing

  if (!file1) {
    // if the file didn't open, print an error:
    Serial.println("error opening list1.txt");
    while (1);
  }
  if (!file2) {
    // if the file didn't open, print an error:
    Serial.println("error opening list2.txt");
    while (1);
  }
  Serial.println("O==-- Database files opened successfully --==O");
}
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
void loop() {
  ////////////////////////////////////////////////////////////////////////////////////////////
  //Get PUF response from DE2-115 Board and store it in list2.txt on the SD Card
  ////////////////////////////////////////////////////////////////////////////////////////////
  if (Serial1.available() >= 10)//Check whether there is 10 bytes of data in FIFO buffer (one response)
  {
    Serial.print('\n');
    Serial.print("CRP Generation Success:");
    Serial.print('\n');
    for (int i2 = 0; i2 < 10; i2++)//Loop to get all bytes in the recieve buffer, and store them in list2.txt
    {
      incomingByte = Serial1.read();//Get the byte from the Serial1 reciever buffer
      response[i2] = incomingByte;//Put the bytes in response array for debug printing
      file2.write(incomingByte);//Write each byte into file2
    }
    file2.print("\n");//break into a new line so the next response stored is on a new line
    Serial.print("Challenge: ");//Print out the Challenge sent to the DE2-115
    for (int z = 0; z < 10; z++)
    {
      Serial.print(challenge[z], HEX);
      Serial.print(' ');
    }
    Serial.print('\n');

    Serial.print("Response:  ");//Print out the Response from the DE2-115
    for (int i = 0; i < 10; i++)
    {
```

```
      Serial.print(response[i], HEX);
      Serial.print(' ');
    }
    Serial.print('\n');
    while (Serial1.available() > 0) //clear the junk in the FIFO
    {
      char t = Serial1.read();
    }
    WaitingOnResponse = false;//Now that a response was recieved we can go back to reading and
sending a new challenge
  }
  ////////////////////////////////////////////////////////////////////////////////////////////////
  //Get PUF challenge from list1.txt on the SD Card, and transmit it to the DE2-115
  ////////////////////////////////////////////////////////////////////////////////////////////////
  if (CRP_Gens <= CRP_Gen_Num && WaitingOnResponse == false)//check if we have generated the
desired # of CRPs, if not continue enrollment phase
  {
    for (int i2 = 0; i2 < 10; i2++)//Loop to get all bytes from the SD Card data line
    {
      //print the byte read from the file in serial monitor
      x = file1.read();
      //delay(5);
      Serial.print(x);
      Serial.print(' ');

      //save the challenge sent to the DE2-115
      char v = x;
      challenge[chalcount] = v;
      chalcount++;
      if (chalcount == 10)
      {
        chalcount = 0;
      }
      //delay(5);
      // Send the byte acquired from the challenge database file
      Serial1.write(x);

      //wait 10 ms for the write
      //delay(20);
    }
    CRP_Gens++;//increment the number of CRP's generated and stored in the database
    WaitingOnResponse = true;//Set wait to true, forcing it to wait for a response before sending
subsequent challenges
  }
  else if (CRP_Gens > CRP_Gen_Num)
```

```
  {
    file2.close();

    Serial.println("\n\nFiles found on the card (name, date and size in bytes): ");
    root.openRoot(volume);
    root.ls(LS_R | LS_DATE | LS_SIZE);

    Serial.print("\n\n");
    Serial.print("O==-- Enrollment Phase Complete --==O");
    while(1){
      delay(1000);
    }
  }
}
```

12. APPENDIX B

```
////////////////////////////////////////////////////////////////////////////////////////////////
//Arduino Mega 2560 PUF Temp Sensor Project Code ----- CORRELATION PHASE
//
//This Code is for the CORRELATION PHASE, where we characterize the PUF throughout the temperature range.
//This means repeatedly generating responses from the SRAM and checking them with the responses in the database.
//This is done while manually increasing the SRAM temperature using a heat gun.  The expectation is that
//as the SRAM temperature rises the PUF response will become increasingly unreliable.  This unreliability is
//quantified by a simple ratio of correct responses to incorrect ones, providing a metric that will be associated
//with varying temperatures.  Once this data is accumulated it must be added to its own database, allowing
//the Arduino to look up a unreliability ratio and promptly find the corresponding temperature.
//
//For a single run, the heat gun is used to heat the SRAM by a set degree increment(1C for example),
//then a sufficiently large number of responses are generated.  Each response is checked against the
//stored response in the database.  If they are identical, the PUF reliability variable is incremented.
//If they are not identical, the reliability variable is not incremented.  Once all the CRP's have been
//compared, the reliability ratio is stored alongside the current temperature for reference later during
//the SENSING PHASE.  The expectation is that the change in reliability will be enough to provide a
//gradient that corresponds to the temperature with a fair amount of accuracy.
//
////////////////////////////////////////////////////////////////////////////////////////////////
//Include Statements
////////////////////////////////////////////////////////////////////////////////////////////////
#include <SPI.h>
#include <SD.h>
//Declarations
bool WaitingOnResponse = false;
File file1;
File file2;
unsigned long myTime;
const int chipSelect = 4;//Set correct chipselect
Sd2Card card;
SdVolume volume;
SdFile root;
unsigned char challenge[10];//in the future we have to get the challenge from a server database
unsigned char response_unverified[10];
unsigned char response_valid[10];
unsigned char incomingByte = 0;
```

```
int chalcount = 0;
int x = 0;
int CRP_Gens = 0;
double Reliability_Ratio = 0;
double CRP_Valid = 0;
double CRP_Invalid = 0;
////////////////////////////////////////////////////////////////////////////////////////////////
double CRP_Gen_Num = 10;//HERE WE SET THE NUMBER OF CRP's TO BE TESTED AGAINST THE DATABASE
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
  pinMode(52, OUTPUT);
  pinMode(53, INPUT);
  pinMode(10, OUTPUT);//set ethernet chipselect pin to output mode
  digitalWrite(10, HIGH);//set ethernet chipselect pin to high by default
  pinMode(4, OUTPUT);//set chipselect pin to output mode
  digitalWrite(4, HIGH);//set chipselect pin to high by default
  while (Serial1.available() > 0) { //Clear the junk in the FIFO
    char t = Serial1.read();
  }
  ////////////////////////////////////////////////////////////////////////////////////////////////
  ////////////////////////////////SD Card setup/debug, Proceed to line 103 to skip//////////////////////////
  ////////////////////////////////////////////////////////////////////////////////////////////////
  if (!card.init(SPI_HALF_SPEED, chipSelect)) {
    Serial.println("initialization failed. Things to check:");
    Serial.println("* is a card inserted?");
    Serial.println("* is your wiring correct?");
    Serial.println("* did you change the chipSelect pin to match your shield or module?");
    while (1);
  } else {
    Serial.println("Wiring is correct and a card is present.");
  }
  // print the type of card
  Serial.print("Card type: ");
  switch (card.type()) {
    case SD_CARD_TYPE_SD1:
      Serial.println("SD1");
      break;
    case SD_CARD_TYPE_SD2:
      Serial.println("SD2");
      break;
    case SD_CARD_TYPE_SDHC:
```

```
      Serial.println("SDHC");
      break;
    default:
      Serial.println("Unknown");
  }
  if (!volume.init(card)) {
    Serial.println("Could not find FAT16/FAT32 partition.\nMake sure you've formatted the card");
    return;
  }
  Serial.println("\nFiles found on the card (name, date and size in bytes): ");
  root.openRoot(volume);
  root.ls(LS_R | LS_DATE | LS_SIZE);
  if (!SD.begin(4)) {
    Serial.println("initialization failed!");
    while (1);
  }
  Serial.println("initialization done.");
  ////////////////////////////////////////////Open both text files/////////////////////////////////////
  SD.remove("list2.txt");//erase any leftove data inside the text doc

  file1 = SD.open("list1.txt", FILE_READ);//Open challenge list for reading
  file2 = SD.open("list3.txt", FILE_READ);//Open response list for reading

  if (!file1) {
    // if the file didn't open, print an error:
    Serial.println("error opening list1.txt");
    while (1);
  }
  if (!file2) {
    // if the file didn't open, print an error:
    Serial.println("error opening list2.txt");
    while (1);
  }
  Serial.println("O==-- Database files opened successfully --==O");
  myTime = millis();
}
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////
void loop() {
  ////////////////////////////////////////////////////////////////////////////////////////////////
  //Get PUF response from DE2-115 Board and check it with the corresponding list2.txt response
  ////////////////////////////////////////////////////////////////////////////////////////////////
  if (Serial1.available() >= 10)//Check whether there is 10 bytes of data in FIFO buffer (one response)
  {
    //Serial.print('\n');
```

```
    //Serial.print("CRP Generation Success:");
    //Serial.print('\n');
    for (int i2 = 0; i2 < 10; i2++)//Loop to get the unverified response from the DE2-115 and the
correct/valid response from the SD Card
    {
     incomingByte = Serial1.read();//Get the byte from the Serial1 reciever buffer
     response_unverified[i2] = incomingByte;//Put the bytes in response array for checking
     x = file2.read();//Get the byte from the SD card database
     response_valid[i2] = x;//Put the bytes in response array for checking
    }
    x = file2.read();//Read one extra byte to skip endline character
    for (int i2 = 0; i2 < 10; i2++)//Loop to compare the two responses
    {
     if (response_valid[i2] == response_unverified[i2])
     {
        if(i2 == 9)
        {
           CRP_Valid += 1;
        }
     }
     else
     {

     }
    }
    //This debug info printing to console can be commented out when speed is needed
    /*Serial.print("Challenge: ");//Print out the Challenge sent to the DE2-115
    for (int z = 0; z < 10; z++)
    {
     Serial.print(challenge[z], HEX);
     Serial.print(' ');
    }
    Serial.print('\n');

    Serial.print("Unverified Response:  ");//Print out the Response from the DE2-115
    for (int i = 0; i < 10; i++)
    {
     Serial.print(response_unverified[i], HEX);
     Serial.print(' ');
    }
    Serial.print('\n');

    Serial.print("Valid Response:       ");//Print out the correct Response from the stored SD Card data
    for (int i = 0; i < 10; i++)
    {
```

```
    Serial.print(response_valid[i], HEX);
    Serial.print(' ');
   }
   Serial.print('\n');
   while (Serial1.available() > 0) //clear the junk in the FIFO
   {
    char t = Serial1.read();
   }
   */
   WaitingOnResponse = false;//Now that a response was recieved we can go back to reading and sending a new challenge
  }
  ////////////////////////////////////////////////////////////////////////////////////////////////////
  //Get PUF challenge from list1.txt on the SD Card, and transmit it to the DE2-115
  ////////////////////////////////////////////////////////////////////////////////////////////////////
  if (CRP_Gens < CRP_Gen_Num && WaitingOnResponse == false)//check if we have generated the desired # of CRPs, if not continue
  {
   for (int i2 = 0; i2 < 10; i2++)//Loop to get all bytes from the SD Card data line
   {
    //print the byte read from the file in serial monitor
    x = file1.read();
    //delay(5);
    //Serial.print(x);
    //Serial.print(' ');

    /*//save the challenge sent to the DE2-115
    char v = x;
    challenge[chalcount] = v;
    chalcount++;
    if (chalcount == 10)
    {
     chalcount = 0;
    }
    */
    // Send the byte acquired from the challenge database file
    Serial1.write(x);

    //wait 10 ms for the write
    //delay(20);
   }
   CRP_Gens++;//increment the number of CRP's generated and stored in the database
   WaitingOnResponse = true;//Set wait to true, forcing it to wait for a response before sending subsequent challenges
  }
```

```
  else if (CRP_Gens >= CRP_Gen_Num)
  {
    unsigned long mytime2 = millis();
    file2.close();
    file1.close();
    Serial.println("\n\nFiles found on the card (name, date and size in bytes): ");
    root.openRoot(volume);
    root.ls(LS_R | LS_DATE | LS_SIZE);

    Serial.print("\n\n");
    Reliability_Ratio = (CRP_Valid/CRP_Gen_Num);//Here we set the reliability ratio, which can also
be considered the percentage of correct responses
    Serial.print("O==-- Responses Generated: ");
    Serial.print(CRP_Gens, 1);
    Serial.print("\n");
    Serial.print("O==-- Number Correct: ");
    Serial.print(CRP_Valid, 1);
    Serial.print("\n");
    Serial.print("O==-- Reliability Ratio: ");
    Serial.print(Reliability_Ratio, 4);
    Serial.print("\n");
    Serial.print("O==-- Processing Time: ");
    Serial.print((mytime2-myTime), DEC);Serial.print(" milliseconds");
    Serial.print("\n");
    Serial.print("O==-- Correlation Phase Run Complete --==O");
    while (1) {
      delay(1000);
    }
  }
}
```

13. APPENDIX C

```
////////////////////////////////////////////////////////////////////////////////////////////////
//Arduino Mega 2560 PUF Temp Sensor Project Code ----- SENSING PHASE
//
//This Code is for the SENSING PHASE, where we actively sense temperature by repeatedly generating responses
//and comparing them with those stored in the database at room temperature, resulting in a reliability rate
//that can then be checked against the cutoff reliability rates determined in the correlation phase.
//
//The cutoff and warning temperatures were 50C and 60C, corresponding to reliabilities of 0.90488 and 0.890768.
//
//The difficulty is that there seems to be a significant amount of random bit flips, which increases as the
//temperature increases.  Thus the accuracy of the sensing seems to be within an error rate of around 1-3%.
//
//This aside, the way this code works is by repeatedly polling the reliability, then checking the result against
//the above thresh-hold values for reliability.
//
////////////////////////////////////////////////////////////////////////////////////////////////
//Include Statements
////////////////////////////////////////////////////////////////////////////////////////////////
#include <SPI.h>
#include <SD.h>
//Declarations
bool WaitingOnResponse = false;
File file1;
File file2;
unsigned long myTime;
const int chipSelect = 4;//Set correct chipselect
Sd2Card card;
SdVolume volume;
SdFile root;
unsigned char challenge[10];//in the future we have to get the challenge from a server database
unsigned char response_unverified[10];
unsigned char response_valid[10];
unsigned char incomingByte = 0;
int chalcount = 0;
int x = 0;
int CRP_Gens = 0;
int Rel_Stack = 0;
double Rel_Val = 0;
```

```
double Reliability_Ratio = 0;
double CRP_Valid = 0;
double CRP_Invalid = 0;
int vibmotor = 5;
int vibmotor2 = 4;
////////////////////////////////////////////////////////////////////////////////////////////////
double CRP_Gen_Num = 450;//HERE WE SET THE NUMBER OF CRP's TO BE TESTED
AGAINST THE DATABASE
////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
void setup() {
  Serial.begin(9600);
  Serial1.begin(9600);
  Serial3.begin(9600);
  digitalWrite(vibmotor, HIGH);
  digitalWrite(vibmotor2, HIGH);
  pinMode(2, OUTPUT);
  pinMode(52, OUTPUT);
  pinMode(53, INPUT);
  pinMode(10, OUTPUT);//set ethernet chipselect pin to output mode
  digitalWrite(10, HIGH);//set ethernet chipselect pin to high by default
  pinMode(4, OUTPUT);//set chipselect pin to output mode
  digitalWrite(4, HIGH);//set chipselect pin to high by default
  while (Serial1.available() > 0) { //Clear the junk in the FIFO
    char t = Serial1.read();
  }

  pinMode(vibmotor, OUTPUT);
  pinMode(vibmotor2, OUTPUT);

  ////////////////////////////////////////////////////////////////////////////////////////////////
  ////////////////////////////////SD Card setup/debug, Proceed to line 103 to skip//////////////////////
  ////////////////////////////////////////////////////////////////////////////////////////////////
  if (!card.init(SPI_HALF_SPEED, chipSelect)) {
    Serial.println("initialization failed. Things to check:");
    Serial.println("* is a card inserted?");
    Serial.println("* is your wiring correct?");
    Serial.println("* did you change the chipSelect pin to match your shield or module?");
    while (1);
  } else {
    Serial.println("Wiring is correct and a card is present.");
  }
  // print the type of card
  Serial.print("Card type: ");
  switch (card.type()) {
```

```
    case SD_CARD_TYPE_SD1:
      Serial.println("SD1");
      break;
    case SD_CARD_TYPE_SD2:
      Serial.println("SD2");
      break;
    case SD_CARD_TYPE_SDHC:
      Serial.println("SDHC");
      break;
    default:
      Serial.println("Unknown");
  }
  if (!volume.init(card)) {
    Serial.println("Could not find FAT16/FAT32 partition.\nMake sure you've formatted the card");
    return;
  }
  //Serial.println("\nFiles found on the card (name, date and size in bytes): ");
  //root.openRoot(volume);
  //root.ls(LS_R | LS_DATE | LS_SIZE);
  if (!SD.begin(4)) {
    Serial.println("initialization failed!");
    while (1);
  }
  Serial.println("initialization done.");
  ////////////////////////////////////////////Open both text files/////////////////////////////////////
  SD.remove("list2.txt");//erase any leftove data inside the text doc

  file1 = SD.open("list1.txt", FILE_READ);//Open challenge list for reading
  file2 = SD.open("list3.txt", FILE_READ);//Open response list for reading

  if (!file1) {
    // if the file didn't open, print an error:
    Serial.println("error opening list1.txt");
    while (1);
  }
  if (!file2) {
    // if the file didn't open, print an error:
    Serial.println("error opening list2.txt");
    while (1);
  }
  Serial.println("O==-- Database files opened successfully --==O");
  myTime = millis();
}
////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
void loop() {
  ////////////////////////////////////////////////////////////////////////////////////////////////
  //Get PUF response from DE2-115 Board and check it with the corresponding list2.txt response
  ////////////////////////////////////////////////////////////////////////////////////////////////
  if (Serial1.available() >= 10)//Check whether there is 10 bytes of data in FIFO buffer (one response)
  {
    //Serial.print('\n');
    //Serial.print("CRP Generation Success:");
    //Serial.print('\n');
    for (int i2 = 0; i2 < 10; i2++)//Loop to get the unverified response from the DE2-115 and the
correct/valid response from the SD Card
    {
      incomingByte = Serial1.read();//Get the byte from the Serial1 reciever buffer
      response_unverified[i2] = incomingByte;//Put the bytes in response array for checking
      x = file2.read();//Get the byte from the SD card database
      response_valid[i2] = x;//Put the bytes in response array for checking
    }
    x = file2.read();//Read one extra byte to skip endline character
    for (int i2 = 0; i2 < 10; i2++)//Loop to compare the two responses
    {
      if (response_valid[i2] == response_unverified[i2])
      {
        if (i2 == 9)
        {
          CRP_Valid += 1;
        }
      }
      else
      {

      }
    }
    //This debug info printing to console can be commented out when speed is needed
    /*Serial.print("Challenge: ");//Print out the Challenge sent to the DE2-115
      for (int z = 0; z < 10; z++)
      {
      Serial.print(challenge[z], HEX);
      Serial.print(' ');
      }
      Serial.print('\n');

      Serial.print("Unverified Response:  ");//Print out the Response from the DE2-115
      for (int i = 0; i < 10; i++)
      {
      Serial.print(response_unverified[i], HEX);
```

```
    Serial.print(' ');
    }
    Serial.print('\n');

    Serial.print("Valid Response:       ");//Print out the correct Response from the stored SD Card data
    for (int i = 0; i < 10; i++)
    {
    Serial.print(response_valid[i], HEX);
    Serial.print(' ');
    }
    Serial.print('\n');
    while (Serial1.available() > 0) //clear the junk in the FIFO
    {
    char t = Serial1.read();
    }
   */
   WaitingOnResponse = false;//Now that a response was recieved we can go back to reading and
sending a new challenge
  }
  ////////////////////////////////////////////////////////////////////////////////////////////////
  //Get PUF challenge from list1.txt on the SD Card, and transmit it to the DE2-115
  ////////////////////////////////////////////////////////////////////////////////////////////////
  if (CRP_Gens < CRP_Gen_Num && WaitingOnResponse == false)//check if we have generated the
desired # of CRPs, if not continue
  {
   for (int i2 = 0; i2 < 10; i2++)//Loop to get all bytes from the SD Card data line
   {
    //print the byte read from the file in serial monitor
    x = file1.read();
    //delay(5);
    //Serial.print(x);
    //Serial.print(' ');

    /*//save the challenge sent to the DE2-115
      char v = x;
      challenge[chalcount] = v;
      chalcount++;
      if (chalcount == 10)
      {
      chalcount = 0;
      }
    */
    // Send the byte acquired from the challenge database file
    Serial1.write(x);
```

```
     //wait 10 ms for the write
     //delay(20);
    }
    CRP_Gens++;//increment the number of CRP's generated and stored in the database
    WaitingOnResponse = true;//Set wait to true, forcing it to wait for a response before sending subsequent challenges
  }
  else if (CRP_Gens >= CRP_Gen_Num)
  {
    Rel_Stack++;
    file2.close();
    file1.close();
    //Serial.println("\n\nFiles found on the card (name, date and size in bytes): ");
    //root.openRoot(volume);
    //root.ls(LS_R | LS_DATE | LS_SIZE);

    Serial.print("\n\n");
    Reliability_Ratio = (CRP_Valid / CRP_Gen_Num); //Here we set the reliability ratio, which can also be considered the percentage of correct responses
    Rel_Val += Reliability_Ratio;//Here we add the current reliability ratio to the reliability value, after 3 ratios are added we divide by 3 to get a more accurate sensed reliability
    Serial.print("O==-- Reliability Ratio: ");
    Serial.print(Reliability_Ratio, 6);
    Serial.print("\n");
    if (Rel_Stack == 3)
    {
      Rel_Stack = 0;//set number of sensed reliabilities back to 0
      Reliability_Ratio = Rel_Val / 3;//Get the average of the 3 sensed reliabilities
      Rel_Val = 0;//set reliability sum back to 0
      Serial.print("O==-- Average Reliability Ratio: ");
      Serial.print(Reliability_Ratio, 6);
      Serial.print("\n");
      if (Reliability_Ratio == 0.890768)
      {
        //Here we trigger shutoff
        Serial.print("O==-- SHUTOFF TRIGGERED!!!");
        Serial.print("\n");
      }
      else if (Reliability_Ratio <= 0.90488)// 0.90488
      {
        //Here we trigger message to android phone, warnings, etc
        Serial.print("O==-- WARNING TRIGGERED!!!");
        Serial.print("\n");
        int x1 = 0;
        Serial3.print("Warning - Vehicle Battery Overheating \n");
```

```
      Serial3.print("Please Pull Over! \n");
      while (x1 < 10000)
      {
       x1 = x1 + 1;
       digitalWrite(vibmotor, HIGH);
       digitalWrite(vibmotor2, LOW);
       digitalWrite(2, HIGH);
       delay(2);
       digitalWrite(2, LOW);
      }
     }
     else
     {
      Serial.print("O==-- Temperature Acceptable...");
      Serial.print("\n");
     }
     Serial.print("restart!");
    }
    else
    {
     delay(20);
    }
    CRP_Gens = 0;//Reset the crp_gens so we can continue on to get the next reliability value
    CRP_Valid = 0;//Reset number of valid CRPs
    file1 = SD.open("list1.txt", FILE_READ);//Reopen challenge list to read from start again
    file2 = SD.open("list3.txt", FILE_READ);//Reopen response list to read from start again
    while (Serial1.available() > 0) //clear any junk in the FIFO
    {
     char t = Serial1.read();
    }
    WaitingOnResponse = false;//Make sure this is false so when we restart checking reliability we start
by sending challenge
  }
}
```

14. APPENDIX D

```
#include "unistd.h"
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <altera_avalon_spi.h>
#include "system.h"
#include "alt_types.h"
#include "sys\alt_irq.h"
#include "sys\alt_stdio.h"
#include <sys/alt_flash.h>
#include "altera_avalon_uart_regs.h"
#include "altera_avalon_spi_regs.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_uart.h"
#include "altera_avalon_uart_fd.h"
#include "altera_avalon_timer_regs.h"
#include "Altera_UP_SD_Card_Avalon_Interface.h"
#include <altera_up_sd_card_avalon_interface.h>
#include "Bitmask2.h"//include the custom-generated reliability bitmask for the SRAM chip
#define READ_BLOCK 0x0011//Define read and write blocks for SD_Card commands:
#define WRITE_BLOCK 0x0018
int SD_Base = ALTERA_UP_SD_CARD_AVALON_INTERFACE_0_BASE;//Use "SD_Base" for
easier name convention:
char c1 = 0;
int Chars_Received = 0;
int x1 = 0;
int CRP_Number = 1000; //Here we set the number of CRP pairs we will be using, this is currently not
used, we just return responses indefinitely
static alt_u8 rxdata[10]; //Declare a static char array of 7 chars, used for storing the received Challenge
static alt_u8 txdata[10]; //Declare a static char array of 7 chars, used for storing the generated Response
short int file_handle; //this will be the file handler used for accessing data on the SD card
alt_u8 readDat[1];
alt_u8 garb[1];
alt_u8 StartDat[32720];
alt_u8 ReferenceDat[32720];
alt_u8 TempDat[32720];
alt_u8 BitmaskOutALL[32720];
alt_u32 flags = 0;
alt_u8 com_read[] = {0x03};//Read
alt_u8 com_write[] = {0x02};//Write
alt_u8 com_r_sr[] = {0x05};//Read from Status
alt_u8 com_w_sr[] = {0x01};//Write to Status
alt_u8 seq_w[] = {0x41};//To set sequential access mode
```

```
alt_u8 L1[] = {0x00};//To set byte-wise access mode
alt_u16 count = 32710;//32700 bytes in SPI-SRAM
alt_u16 count2 = 10;//for debugging SPI-SRAM reads, seem to only get randomness when its at or
around 20???
alt_u16 count3 = 50;//for debugging SPI-SRAM reads
alt_u16 countZ = 0;//for debugging SPI-SRAM reads
alt_u16 bitMaskDepth = 10;//for setting the number of bytes we test during bitmask testing
//=========================================================================
=========================================================================
=====
//==============================GENERAL
FUNCTIONS============================================================GENER
AL FUNCTIONS==============================
//=========================================================================
=========================================================================
=====
void Time_Delay(int x)//simple processing based delay timer
{
        while (x>0)
        {x--;}
}
void View_ONCHIP_Contents(int start_offset)//Loop to write out ONCHIP memory contents
{
        int zeros = 0;
        int x1 = 0;
        while(1)
          {
                int datax = IORD_ALTERA_AVALON_PIO_DATA(ONCHIP_MEMORY2_0_BASE
+ start_offset);
                printf("%d ", datax);
                if (x1 == 0){
                if(datax == 0){zeros++;} else{printf(" ____Zeros: %d ", zeros);  x1 = 1;}
                }
                start_offset = start_offset + 4;
                Time_Delay(100);
          }
}
void View_SRAM_Contents(int start_offset)//Loop to write out onboard SRAM contents
{
        int indexa = 0;
        while(1)
          {
                int datax = IORD_ALTERA_AVALON_PIO_DATA(SRAM_0_BASE + start_offset);
                printf("%d--", indexa);
                printf("%d ", datax);
```

```
                start_offset = start_offset + 4;
                indexa++;
                Time_Delay(100);
        }
}
void Zero_Fill_SRAM(int start_offset, int span)//Loop to 0-fill onboard SRAM contents
{
        int current = 0;
        while(1)
                {
                        IOWR_ALTERA_AVALON_PIO_DATA(SRAM_0_BASE + start_offset, 0);
                        start_offset = start_offset + 4;
                        current++;
                        if (current == span){break;}
                        Time_Delay(100);
                }
}
void Prep_Timer()//Prep the timer for 1 second counting
{
        IOWR_ALTERA_AVALON_TIMER_PERIOD_0(TIMER_0_BASE, 0xF080);
        IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x02FA);//starting period
is 1 second
        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear the TO bit by
writing 0 to the status register
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
}
void Prep_Timer_fifth()//Prep the timer for 0.2 second counting
{
        IOWR_ALTERA_AVALON_TIMER_PERIOD_0(TIMER_0_BASE, 0x9680);
        IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x0098);//starting period
is 0.2 seconds
        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear the TO bit by
writing 0 to the status register
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
}
void Prep_Timer_milli()//Prep the timer for millisecond counting
{
        IOWR_ALTERA_AVALON_TIMER_PERIOD_0(TIMER_0_BASE, 0xc350);
        IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x0000);//starting period
is 0.2 seconds
        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear the TO bit by
writing 0 to the status register
```

```
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
}
void Prep_Timer_micro()//Prep the timer for millisecond counting
{
	IOWR_ALTERA_AVALON_TIMER_PERIOD_0(TIMER_0_BASE, 0x0032);
	IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x0000);//starting period
is 0.2 seconds
	IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear the TO bit by
writing 0 to the status register
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
}
bool TimeoutBitHigh()//used in timer implementation function
{
	int ClkReg = IORD_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE);//Read status
register
	int regTest = ClkReg & 0x01;//test the Timeout bit in the status register using bitwise and
operation
	if(regTest == 0x01)//if timeout bit is active
	{
		IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear the TO
bit
		return true;
	}
	else
	{
		return false;
	}
}
void Interval_Timer_Delay(int seconds)//call hardware interval timer, does not stop processing
{
	Prep_Timer();
	IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x02FA);//write to period
to reset counter
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
	while(1)
	{
		if(TimeoutBitHigh())
		{
			IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear
the TO bit by writing 0 to the status register
			seconds--;
			if(seconds == 0)
```

```
                        {break;}
                }
        }
}
void Interval_Timer_Delay_fifth(int seconds)//call hardware interval timer, does not stop processing
{
        Prep_Timer_fifth();
        IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x0098);//write to period
to reset counter
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
        while(1)
        {
                if(TimeoutBitHigh())
                {
                        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear
the TO bit by writing 0 to the status register
                        seconds--;
                        if(seconds == 0)
                        {break;}
                }
        }
}
void Interval_Timer_Delay_milli(int seconds)//call hardware interval timer, does not stop processing
{
        Prep_Timer_milli();
        IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x0000);//write to period
to reset counter
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1
and start bit to 1
        while(1)
        {
                if(TimeoutBitHigh())
                {
                        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear
the TO bit by writing 0 to the status register
                        seconds--;
                        if(seconds == 0)
                        {break;}
                }
        }
}
void Interval_Timer_Delay_micro(int seconds)//call hardware interval timer, does not stop processing
{
        Prep_Timer_micro();
```

```
	IOWR_ALTERA_AVALON_TIMER_PERIOD_1(TIMER_0_BASE, 0x0000);//write to period to reset counter
	IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_0_BASE, 6);//set CONT bit to 1 and start bit to 1
	while(1)
	{
		if(TimeoutBitHigh())
		{
			IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_0_BASE, 0x0);//clear the TO bit by writing 0 to the status register
			seconds--;
			if(seconds == 0)
			{break;}
		}
	}
}
void View_SDRAM_Contents(int start_offset, int length)//Loop to write out onboard SDRAM contents
{
	while(length != 0)
	  {
		length--;
		int datax = IORD_ALTERA_AVALON_PIO_DATA(SDRAM_0_BASE + start_offset);
		printf("%d ", datax);
		start_offset = start_offset + 4;
	  }
}
void Zero_Fill_SDRAM(int start_offset, int span)//Loop to 0-fill onboard SDRAM contents
{
	int current = 0;
	int start_offset2 = start_offset;
	while(current != span)
		{
			current++;
			IOWR_ALTERA_AVALON_PIO_DATA(SDRAM_0_BASE + start_offset2, 0x7FFFFFFF);
			start_offset2 += 4;
		}
}
//=============================================================================================================
//==============================SD-CARD FUNCTIONS==========================================SD-CARD FUNCTIONS==============================
```

```
//=============================================================================
=============================================================================
=====
void SD_CARD_INIT()
{
	alt_up_sd_card_dev *sd_card_dev =
alt_up_sd_card_open_dev("/dev/Altera_UP_SD_Card_Avalon_Interface_0");
	while(sd_card_dev == 0)
	{

	}
}

void SD_CARD_FAT_CHECK()
{
	if(alt_up_sd_card_is_FAT16())
	{
		printf("Fat16 Confirmed \n");
	}
	else
	{
		printf("Fat16 CONFIRMATION FAILURE \n");
	}
}
//=============================================================================
=============================================================================
=====
//==========================SPI SRAM
IMPLEMENTATION==========================================================SPI
SRAM IMPLEMENTATION====================
//=============================================================================
=============================================================================
=====
//===================================Below are the SRAM serial
commands=============================================================
=================
//alt_u8 com_read[] = {0x03};//Read
//alt_u8 com_write[] = {0x02};//Write
//alt_u8 com_r_sr[] = {0x05};//Read from Status
//alt_u8 com_s_sr[] = {0x01};//Write to Status
//alt_u8 seq_w[] = {0x41};//To set sequential access mode
//alt_u8 L1[] = {0x00};//To set byte-wise access mode
//alt_u16 count = 32700;//32700 bytes in SPI-SRAM
//flags or ALT_AVALON_SPI_COMMAND_MERGE - flags lets chipselect go high after command,
merge leaves it low so multiple commands can take place in series
```

```
//INFO  - Recall that the SRAM has 32,768 bytes.  Divide by 60 to get 546 lines.
//=============================================================================
=============================================================================
=====
//=========================================================SPI SRAM
Functions=============================================================================
======
void POWER_SPI_SRAM()//Turns on power to SPI SRAM
{
	IOWR_ALTERA_AVALON_PIO_DATA(SPI_POWER_BASE, 1);
	Time_Delay(5000);//Wait a bit
	//		Base	Slave	w_length	w_data	r_length	r_data	flags
	alt_avalon_spi_command(0xa0,	0,	1,	com_r_sr,	1,	readDat,	flags);//Trigger
chipselect to switch SPI-SRAM to active mode
	//See section 2.7 of the the 23K256 Data Sheet for why the above command is necessary
}
void DEPOWER_SPI_SRAM()//Turns off power to SPI SRAM
{
	IOWR_ALTERA_AVALON_PIO_DATA(SPI_POWER_BASE, 0);
}
alt_u8 getUpperByte(alt_u16 d)//Returns upper byte of a word
{
	alt_u8 hi = (d & 0xff00) >>8;
	return hi;
}
alt_u8 getLowerByte(alt_u16 d)//Returns lower byte of a word
{
	alt_u8 lo = (d & 0x00ff);
	return lo;
}
alt_u8 getLSBs(alt_u8 d)//Returns LSBs of a byte
{
	alt_u8 lo = (d & 0x0f);
	return lo;
}
alt_u8 getMSBs(alt_u8 d)//Returns MSBs of a byte
{
	alt_u8 hi = (d & 0xf0) >>4;
	return hi;
}
void SPI_Write_Zeros(alt_u16 span)//Writes 0's to SRAM from address 0 to address "span", useful for
checking data retention time effects
{
	alt_u16 span2 = 0;
	alt_u8 UB[] = {getUpperByte(span2)};
```

```
	alt_u8 LB[] = {getLowerByte(span2)};
	Time_Delay(1000);//Wait a bit
	//		Base Slave w_length w_data r_length r_data flags
	alt_avalon_spi_command(0xa0, 0, 1, com_write, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Send Read Instruction
	alt_avalon_spi_command(0xa0, 0, 1, UB, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Send first 8 bits of address
	alt_avalon_spi_command(0xa0, 0, 1, LB, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Send last  8 bits of address

	while(span2 < (span - 1))
	{	//		Base Slave w_length w_data r_length r_data flags
		alt_avalon_spi_command(0xa0, 0, 1, LB, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Repeatedly write data
		Time_Delay(100);//Wait a bit
		span2++;
	}
	alt_avalon_spi_command(0xa0, 0, 1, LB, 0, readDat, flags);//Write last
data
	Time_Delay(100);//Wait a bit
}
void SPI_Write_Ones(alt_u16 span)//Writes 1's to SRAM from address 0 to address "span", useful for
checking data retention time effects
{
	alt_u16 span2 = 0;
	alt_u8 span3[] = {0x01};
	alt_u8 UB[] = {getUpperByte(span2)};
	alt_u8 LB[] = {getLowerByte(span2)};
	Time_Delay(1000);//Wait a bit
	//		Base Slave w_length w_data r_length r_data flags
	alt_avalon_spi_command(0xa0, 0, 1, com_write, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Send Read Instruction
	alt_avalon_spi_command(0xa0, 0, 1, UB, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Send first 8 bits of address
	alt_avalon_spi_command(0xa0, 0, 1, LB, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Send last  8 bits of address

	while(span2 < (span - 1))
	{	//		Base Slave w_length w_data r_length r_data flags
		alt_avalon_spi_command(0xa0, 0, 1, span3, 0, readDat,
ALT_AVALON_SPI_COMMAND_MERGE);//Repeatedly write data
		Time_Delay(100);//Wait a bit
		span2++;
	}
```

```
	alt_avalon_spi_command(0xa0, 0, 1, span3, 0, readDat, flags);//Write last data
	Time_Delay(100);//Wait a bit
}
void SPI_Read_Store_X(alt_u16 span, alt_u16 addr, alt_u8 array1[])//Read and store span bytes of SRAM data starting from address addr
{
	alt_u16 addr2 = addr;
	alt_u16 span2 = 0;
	alt_u8 UB[] = {getUpperByte(addr2)};
	alt_u8 LB[] = {getLowerByte(addr2)};
	//Time_Delay(1000);//Wait a bit
	//		Base Slave w_length w_data r_length r_data flags
	alt_avalon_spi_command(0xa0, 0, 1, com_read, 0, readDat, ALT_AVALON_SPI_COMMAND_MERGE);//Send Read Instruction
	alt_avalon_spi_command(0xa0, 0, 1, UB, 0, readDat, ALT_AVALON_SPI_COMMAND_MERGE);//Send first 8 bits of address
	alt_avalon_spi_command(0xa0, 0, 1, LB, 0, readDat, ALT_AVALON_SPI_COMMAND_MERGE);//Send last 8 bits of address

	while(span2 < (span - 1))
	{	//		Base Slave w_length w_data r_length r_data flags
		alt_avalon_spi_command(0xa0, 0, 0, LB, 1, readDat, ALT_AVALON_SPI_COMMAND_MERGE);//Repeatedly read data
		//Time_Delay(100);//Wait a bit
		alt_u8 zz = readDat[0];
		//Time_Delay(100);//Wait a bit
		array1[addr2] = zz;//store a byte of SRAM data in memory
		//Time_Delay(100);//Wait a bit
		addr2++;
		span2++;
	}
	alt_avalon_spi_command(0xa0, 0, 0, LB, 1, readDat, flags);//Read last data, end sequential read with high SS_n signal
	//Time_Delay(100);//Wait a bit
	alt_u8 zz = readDat[0];
	//Time_Delay(100);//Wait a bit
	array1[addr2] = zz;//store a byte of SRAM data in memory
}
void SPI_Set_Sequential_Read()//Set SPI-SRAM to sequential mode, allowing continuous reading of bytes
{
	//		Base Slave w_length w_data r_length r_data flags
	alt_avalon_spi_command(0xa0, 0, 1, com_w_sr, 0, readDat, ALT_AVALON_SPI_COMMAND_MERGE);//INSTRUCTION - Write Status Register
```

```
	alt_avalon_spi_command(0xa0, 0, 1, seq_w, 0, readDat, flags);//Data to Status Register(sequential mode)
	alt_avalon_spi_command(0xa0, 0, 1, com_r_sr, 1, readDat, flags);//INSTRUCTION - Read status register
	Time_Delay(100);//Wait a bit
	//alt_u8 zz = readDat[0];
	//printf("STATUS: %x \n" , zz);//printout status register of SRAM(tells us what mode SRAM is configured to)
	//if(zz!=0x41){printf("STATUS SET FAILED\n");}
}
void SPI_SRAM_RESET2READ_CYCLE(alt_u16 in1, alt_u16 addr, alt_u8 array1[])//(1)Turns on SPI-SRAM, (2)Sets it to sequential mode, (3)Reads SPI from 0 to "in1", (4)Turns off SPI-SRAM
{
	POWER_SPI_SRAM();//Power up SPI-SRAM
	IOWR_ALTERA_AVALON_SPI_SLAVE_SEL(SPI_0_BASE, 1);//Switch slave select to the chip
	IOWR_ALTERA_AVALON_SPI_CONTROL(SPI_0_BASE, ALTERA_AVALON_SPI_CONTROL_SSO_MSK);//force ss_n outputs to go out
	DEPOWER_SPI_SRAM();//Power down SPI-SRAM
	Interval_Timer_Delay_milli(120);//============================DATA RETENTION WAIT TIME(ms)==============================
	POWER_SPI_SRAM();//Power up SPI-SRAM
	//printf("\n");
	//Time_Delay(300);
	SPI_Set_Sequential_Read();//Set SPI-SRAM to sequential mode, allowing continuous reading of bytes
	//Time_Delay(300);
	SPI_Read_Store_X(in1, addr, array1);//Read and store in1 bytes of SRAM data starting from address addr
	//Time_Delay(300);
	//SPI_Write_Zeros(in1);
	DEPOWER_SPI_SRAM();//Power down SPI-SRAM
}
void SPI_SRAM_READ_CYCLE(alt_u16 in1, alt_u16 addr, alt_u8 array1[])//
{
	//Function Details:
	//This function reads from the SRAM at a given location without a retention time delay.  This is for use in correlation/sensing phases.
	//The "SPI_SRAM_RESET2READ_CYCLE" function is used to generate bitmasks, since it requires avoiding data retention in order to get a reliable bitmask.
	//So for generating the bitmask the SRAM is turned off for 120ms or so each run.  This time delay is not needed in correlation/sensing phases.
	//We can simply turn on the SRAM, use it to generate the needed response, then shut it down.
	POWER_SPI_SRAM();//Power up SPI-SRAM
```

```
	IOWR_ALTERA_AVALON_SPI_SLAVE_SEL(SPI_0_BASE, 1);//Switch slave select to the
chip
	IOWR_ALTERA_AVALON_SPI_CONTROL(SPI_0_BASE,
ALTERA_AVALON_SPI_CONTROL_SSO_MSK);//force ss_n outputs to go out
	DEPOWER_SPI_SRAM();//Power down SPI-SRAM
	Interval_Timer_Delay_micro(10);//==============================DATA RETENTION
WAIT TIME(ms)==============================
	POWER_SPI_SRAM();//Power up SPI-SRAM
	SPI_Set_Sequential_Read();//Set SPI-SRAM to sequential mode, allowing continuous reading
of bytes
	SPI_Read_Store_X(in1, addr, array1);//Read and store in1 bytes of SRAM data starting from
address addr
	DEPOWER_SPI_SRAM();//Power down SPI-SRAM
}
void Print_Data_Array(alt_u16 length, alt_u16 addr, alt_u8 array[])
{
	int z = 0;
	while(length > 0)
	{
		printf("%x ", array[addr]);
		addr++;
		length--;
		z++;
		if (z>100)
		{
			printf("\n");
			z = 0;
		}
	}
	printf("\n");
}
void Gather_Bitmask_Data_ALL(alt_u16 span, int Rounds)//span in memory, number of runs
{
	SPI_SRAM_RESET2READ_CYCLE(span, 0, StartDat);//==============STEP 1 READ
IN START DATA=================
	//Print_Data_Array(span, 0, StartDat);
	alt_u16 i = 0;
	Time_Delay(100);
	while(Rounds > 0)//Loop to repeatedly AND referenceData and TempData to keep track of 1's
becoming 0's
	{
		SPI_SRAM_RESET2READ_CYCLE(span, 0,
ReferenceDat);//==============STEP 2 READ IN REF DATA================
		Print_Data_Array(span, 0, ReferenceDat);
		i = 0;
```

```
		while(i < span)//this loop performs bitwise OR on each byte in ReferenceDat
		{
			TempDat[i]=ReferenceDat[i]^StartDat[i];//==============STEP 3 OR START^REF=================
			BitmaskOutALL[i]=TempDat[i]|BitmaskOutALL[i];//==============STEP 4 OR RESULT WITH OUTPUT ARRAY=================
			i++;
		}
		Rounds--;
	}
	printf("\n");
	printf("Generated Bitmask: ");
	Print_Data_Array(20, 0, BitmaskOutALL);
}
void Gather_Bitmask_Data_Limited(alt_u16 span, int Rounds, alt_u16 addr)//span in memory, number of runs, start address
{
	SPI_SRAM_RESET2READ_CYCLE(span, addr, StartDat);//==============STEP 1 READ IN START DATA=================
	Print_Data_Array(span, addr, StartDat);
	alt_u16 i = 0;
	Time_Delay(100);
	while(Rounds > 0)//Loop to repeatedly AND referenceData and TempData to keep track of 1's becoming 0's
	{
		SPI_SRAM_RESET2READ_CYCLE(span, addr, ReferenceDat);//==============STEP 2 READ IN REF DATA=================
		Print_Data_Array(span, addr, ReferenceDat);
		i = 0;
		alt_u16 addrz = addr;
		while(i < span)//this loop performs bitwise OR on each byte in ReferenceDat
		{

TempDat[addrz]=ReferenceDat[addrz]^StartDat[addrz];//==============STEP 3 OR START^REF=================

BitmaskOutALL[addrz]=TempDat[addrz]|BitmaskOutALL[addrz];//==============STEP 4 OR RESULT WITH OUTPUT ARRAY=================
			addrz++;
			i++;
		}
		Rounds--;
	}
	//printf("\n");
	printf("Generated Bitmask: ");
```

```
	Print_Data_Array(20, 0, BitmaskOutALL);
}
unsigned int Generate_Address(alt_u8 challenge[10])
{
	alt_u8 UB0 = getMSBs(challenge[0]);//switch between getting MSBs and LSBs of each byte in
the challenge
	alt_u8 LB1 = getLSBs(challenge[1]);
	alt_u8 UB2 = getMSBs(challenge[2]);
	alt_u8 LB3 = getLSBs(challenge[3]);
	alt_u8 UB4 = getMSBs(challenge[4]);
	alt_u8 LB5 = getLSBs(challenge[5]);
	alt_u8 UB6 = getMSBs(challenge[6]);
	alt_u8 LB7 = getLSBs(challenge[7]);
	alt_u8 UB8 = getMSBs(challenge[8]);
	alt_u8 LB9 = getLSBs(challenge[9]);
	LB7 = LB7 ^ LB9;
	UB6 = UB6 ^ UB8;

	alt_u8 LB0 = getLSBs(challenge[0]);//switch between getting MSBs and LSBs of each byte in
the challenge
	alt_u8 UB1 = getMSBs(challenge[1]);
	alt_u8 LB2 = getLSBs(challenge[2]);
	alt_u8 UB3 = getMSBs(challenge[3]);
	alt_u8 LB4 = getLSBs(challenge[4]);
	alt_u8 UB5 = getMSBs(challenge[5]);
	alt_u8 LB6 = getLSBs(challenge[6]);
	alt_u8 UB7 = getMSBs(challenge[7]);
	alt_u8 LB8 = getLSBs(challenge[8]);
	alt_u8 UB9 = getMSBs(challenge[9]);
	LB6 = LB6 ^ UB9;
	UB7 = UB7 ^ LB8;
	unsigned int X1 =
(UB0<<28)|(LB1<<24)|(UB2<<20)|(LB3<<16)|(UB4<<12)|(LB5<<8)|(UB6<<4)|(LB7);
	unsigned int X2 =
(LB0<<28)|(UB1<<24)|(LB2<<20)|(UB3<<16)|(LB4<<12)|(UB5<<8)|(LB6<<4)|(UB7);
	unsigned int X3 = X1^X2;
	X3 = X3 % 32700;
	//printf("\n");
	//printf("Generated Address: %x \n", X3);
	return X3;

}
void Generate_Response(alt_u8 challenge[10])
{
unsigned int addr = Generate_Address(challenge);//Generate an address from the challenge
```

```
SPI_SRAM_READ_CYCLE(10, addr, StartDat);//store SRAM read for the generated address in
StartDat
//Gather_Bitmask_Data_Limited(10, 100, addr);//Gather bit reliability data of this address, store
SRAM read in StartDat
//Print_Data_Array_start(10, addr);
int inc = 0;
unsigned int addr1 = addr;
while(inc < 10)//Set txdata to the data read from the SRAM XOR'd with challenge data,
                        //Then or the result with the corresponding bit mask to set unreliable bits to 1
{
        txdata[inc] = StartDat[addr1]^challenge[inc];//Response = SRAM_data XOR Challenge_data
        txdata[inc] = txdata[inc]|CONSTANT_186B_BITMASK2[addr1];//Response = Response OR
Bitmask_data
        addr1++;
        inc++;
}
}
void Oldresponse()//A simple response function to send via UART the same data recieved via UART
(for debug purposes)
{
        Chars_Received = 0;
        while (Chars_Received < 10)//Loop to move all 10 characters into the txdata array
        {
                txdata[Chars_Received] = rxdata[Chars_Received];
                Chars_Received++;
        }
}
void Store_Bitmask_To_SD_Card(alt_u16 span, int Rounds)//Generates a bitmask over "Rounds" # of
power cycles, more cycles = more rigor
{
        //NOTE: When using this function it is prudent to comment out any printf commands.
        //      The reason is that the built-in c input-output commands (getc, printf etc) are
        //              actually redirected to target the JTAG serial connection.  Thus, repeated use of
        //              such functions repeatedly sends information through the JTAG USB connection
to the pc.
        //              This seems to slow all other processing by a factor of around 5.4.  Further testing
        //              is needed to fully understand what is happening here, but for all practical
purposes
        //              it is enough to say that printf should be used as little as possible when speed and
        //              time efficiency is a concern.
        //
        //=======================- SD CARD CHECK
-============================//Checks if card is present
        short int *Aux_Status_Reg = (short int*)(SD_Base + 564);
        short int status;
```

```
	do{//LOOP TO CHECK IF CARD HAS BEEN INSERTED
		status = (short int)IORD_16DIRECT(Aux_Status_Reg, 0);
	}
	while((status & 0x02)==0);
	printf("-Card Inserted- \n");
	//=======================- Open SD Card Interface -=====================//
	SD_CARD_INIT();
	//=======================- Check if SD is FAT16 -=======================//
	SD_CARD_FAT_CHECK();
	//=======================- Get file handle for text.txt -===============//
	file_handle = alt_up_sd_card_fopen("text.txt", false);//creates file "text.txt" if it does not exist
	printf("\nfile_handle: %d \n", file_handle);
	//=======================- Put bitmask data in BitmaskOutALL array -====//
	Gather_Bitmask_Data_ALL(span, Rounds);
	//=======================- Put BitmaskOutALL contents on SD Card -======//
	int j = 0;
	while (j < span)
	{
		alt_up_sd_card_write(file_handle, BitmaskOutALL[j]);

		//A wait for processing may be necessary here, only testing will tell
		j++;
	}
	alt_up_sd_card_fclose(file_handle);
	printf("O==-- Bitmask Storage in SD Card Completed --==O");
}
int main()
{

	//Gather_Bitmask_Data_ALL(20, 20);
	//Store_Bitmask_To_SD_Card(count, 400);
	//while(1);

	//Gather_Bitmask_Data_Limited(count2, 1, 0);
	//alt_u8 challengez[10] = {0xa3,0x53,0x06,0x32,0x77,0xac,0xbd,0x06,0x32,0x77};
	//Generate_Address(challengez);
	//challengez[6] = 0xad;
	//Generate_Address(challengez);
	//challengez[4] = 0xac;
	//Generate_Address(challengez);
	//challengez[2] = 0x11;
	//Generate_Address(challengez);
	//while(1){}
```

```
//=============================================================================
=============================================================================
====
//==================================CRP KEY
TRADING=========================================================CRP KEY
TRADING============================
//=============================================================================
=============================================================================
====
        FILE* fp;
        fp = fopen ("/dev/uart_0", "r+"); //Open file for reading and writing
        /*while(1)//Wait for first junk toss
          {
                if (fp)
                {
                        //printf("\n");
                        rxdata[0] = getc(fp);
                        break;
                        //rxdata[Chars_Received] = getc(fp);//READ THE RXDATA REGISTER
                        //Chars_Received++;//INCREMENT # OF CHARS RECEIVED
                }
          }*/
        while(1)//CRP_Number > 0
          {
          ////////////////////////////////////////////////////////////////////////////////////////////////////////
          //RECEIVING CHALLENGE
          ////////////////////////////////////////////////////////////////////////////////////////////////////////
          //printf("R: ");
          Chars_Received = 0;
          while (Chars_Received < 10)//Loop to receive all 10 characters/bytes
          {
                if (fp)
                {
                        //printf("c+");
                        rxdata[Chars_Received] = getc(fp);//READ THE RXDATA REGISTER
                        Chars_Received++;//INCREMENT # OF CHARS RECEIVED
                }
          }
          //printf("done \n");
          //At this point one entire 80 bit(10 char) challenge has been received.  Thus, it is printed out to
the console.
          //for (int i1 = 0; i1 < 10; i1++)
          //{
          //  printf("%c", rxdata[i1]);
          //  Time_Delay(30);
```

```
	//}
	////////////////////////////////////////////////////////////////////////////////////////////////////////
	//RESPONSE GENERATION TAKES PLACE HERE(SRAM PUF IMPLEMENTATION)
	////////////////////////////////////////////////////////////////////////////////////////////////////////
	Generate_Response(rxdata);
	//Oldresponse();
	////////////////////////////////////////////////////////////////////////////////////////////////////////
	//SENDING RESPONSE
	////////////////////////////////////////////////////////////////////////////////////////////////////////
	//Chars_Received = 0;
	//printf("S: ");
	//while (Chars_Received < 10)
	//{
		//printf("%x ", txdata[Chars_Received]);

		//  c1 = txdata[Chars_Received];
		//  IOWR_ALTERA_AVALON_UART_TXDATA(UART_0_BASE, c1);
		//  Chars_Received++;
		//  Time_Delay(1000);
	//}
	//printf("\n");
	fwrite(txdata, 1, 10, fp);//response sent
	Chars_Received = 0;
	CRP_Number--;
	//Time_Delay(12000);
}
}
```

13. APPENDIX E

```
//C++ Program to convert a BIN file to its hex equivalent in text characters
//This code can be used to convert bitmask hex data from an SD card to text
//The resulting text can then be used to hard code a C array containing bitmask data
#include <vector>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <ostream>
using namespace std;
vector<unsigned char>data;
int main()
{
	vector<unsigned char>data;
	string filename = "C:/Users/Trevor/Desktop/TEXT1.BIN";
	FILE* f = NULL;
	fopen_s(&f, filename.c_str(), "rb");
	while (!feof(f)){
		unsigned char c;
		fread(&c, 1, 1, f);
		data.push_back(c);
		//cout << std::hex << (int)c <<", ";
	}
	cout << data.size();
	fclose(f);
	f = NULL;
	fopen_s(&f, "C:/Users/Trevor/Desktop/output.BIN", "w");
	int i = 0;
	while (i<data.size()) {
		string s;
		stringstream ss;
		ss << std::hex << (int)data[i];
		ss >> s;
		s = "0x" + s;
		s += ",";
		fputs(s.c_str(), f);
		if (i % 32 == 0 && i > 0){
			string s2 = "\n";
			fputs(s2.c_str(), f);}
		i++;}
	fclose(f);
	return 0;
}
```

Summer Engineering Experience 2023

HIGH SCHOOL REGISTRATION
NOW OPEN

ENGINEERING, ARCHITECTURAL STUDIES AND CONSTRUCTION MANAGEMENT CAMP WITH HANDS ON ACTIVITIES THAT SHOWCASE MULTIPLE DISCIPLINES AND APPLICATIONS !

Details:

- 5-day camp for students entering grades: 9th, 10th, 11th & 12th in the Fall of 2023
- 9 AM- 3 PM from June 26th - 30th
- Registration Fee: $250

REGISTER HERE

FRESNO STATE
Lyles College of Engineering

2023 Explore Engineering

MIDDLE SCHOOL REGISTRATION
NOW OPEN

ENGINEERING, ARCHITECTURAL STUDIES AND CONSTRUCTION MANAGEMENT CAMP WITH HANDS ON ACTIVITIES THAT SHOWCASE MULTIPLE DISCIPLINES AND APPLICATIONS !

Details:

- 5-day camp for students entering grades: 6th, 7th & 8th in the Fall of 2023.
- 9 AM- 3 PM from June 19th - 23rd
- Registration Fee: $250

REGISTER HERE

FRESNO STATE
Lyles College of Engineering