A Faster-Third-Order Masking of Lookup Tables

The provided c code is to run the third-order look-up table (LUT) scheme. Our code supports the masked execution of AES and PRESENT block-ciphers using third-order LUT. This program is free software; you can redistribute it and/or modify it under the terms of the GPL-3.0 license.

It also provides source code for:

- 32 bit bitsliced implementation taken from https://github.com/annapurna-pvs/Higher-Order-LUT-PRG)
- Rivain-Prouff's implementation taken from https://github.com/coron/htable
- Coron's Increasing shares variant taken from https://github.com/coron/htable
- Coron's Increasing shares variant with LRV taken from https://github.com/coron/htable

Remark: Some methods from these files have been customised to meet the target architecture requirements.

Files like "aes.c", "aes.h", "aes_rp.c" and "share.c" are taken from a github repository having open access licence (https://github.com/coron/htable/tree/master/src).

The parameter "**nt**" (main.c) indicates the number of times to repeat the execution of a function. We have taken the average across 10 executions for the online phase of our scheme. While measuring the execution time of the offline phase, we manually ran the experiment 10 times and have taken the average. The reason being the offline phase of our scheme is computationally heavy.

Source Code Organisation:

The **main.c** is the starting point of the code.

There are four main sub-folders in the "FASTER-THIRD-ORDER-MASKING" folder. They are:

- Util: contains all the functions which is used across various schemes, as well as the RNGA function for random number generation
- PRESENT: contains the functions related to running the third-order scheme for PRESENT
- BITSLICE: contains the functions related to running the 32-bit bitsliced implementation
- AES: This folder contains the code related to running the customised third-order scheme for AES, Rivain-Prouff's RP implementation, Coron's higher-order increasing shares scheme and its LRV variant.

Remark: Since we are interested in only the third-order implementation, the number of shares has been hardcoded to 4 and it is recommended not to change it

To run the chosen scheme:

- The variable "**cipher"** (**main.c**) can be set to "AES_THIRD"/"PRESENT_THIRD" to run our customised scheme for AES/PRESENT in third order respectively.
- Set "**cipher**"(**main.c**) as "AES_HIGHER_ORDER_INCREASING_SHARES" and the variable "**type**"(**main.c**) as "BASIC" or "LRV" depending on your choice, to run corons generic higher order increasing shares scheme (without or with LRV)
- Set "**cipher**" as "BITSLICE" OR "AES_RP" to run the 32-bit masked bitsliced AES-128 or Rivian-Prouff's third-order instantiation respectively.

The code can either run on the target microcontroller or on a desktop.

Set the value of TRNG(in Utils/common.h) to "zero (0)" to run on a desktop (where the random seed is obtained using AES-CTR PRG) or to "one(1)" to use the device built-in RNGA.

This code will include the appropriate header files depending on the choice of TRNG parameter. S-CTR PRG) or to "one(1)" to use device built-in RNGA.

The Table below provides an overview of the ciphers the code can run as mentioned in the paper along with their configurations in the code.

SCHEME NAME (as in the paper)	**CIPHER-NAME**	**TYPE**
Our_Scheme (AES)	AES_THIRD	BASIC
Our_Scheme (PRESENT)	PRESENT_THIRD	BASIC
[CRZ18] (Increasing shares)	AES_HIGHER_ORDER_INCREASING_SHARES	BASIC
[CRZ18] (Increasing shares wit LRV)	AES_HIGHER_ORDER_INCREASING_SHARES	LRV
[GR17] (Bitslicing)	BITSLICE	BASIC
[RP10]	AES_RP	BASIC

To Run on Desktop:

Running the code on the desktop machine is seamless and can be run without editing the software code.

1. In the Makefile, set OS=1/0 depending on whether you are running on a windows or Linux system.

```
Help          Makefile - Faster-Third-Order-Masking-of-Lookup-Tables - Visual Studio Code
C main.c M ●   C common.h M   C present_htable_PRG.c M   C bitslice.c M   C shar
M Makefile
1   OS = 1
2   output: main.o aes.o aes_htable_PRG.o aes_Shares_prg.o present.o present_s
3           gcc main.o aes.o aes_htable_PRG.o aes_Shares_prg.o present.o prese
4
5   common.o: Util/common.c
6           gcc -c -O1 -pedantic -Wall -Wextra -DTRNG=0 Util/common.c
7
8   main.o: main.c
9           gcc -c -O1 -pedantic -Wall -Wextra -DTRNG=0 -DOS=$(OS) main.c
10
11  aes.o: AES/aes.c
12          gcc -c -O1 -pedantic -Wall -Wextra -DTRNG=0 AES/aes.c
13
14  aes_htable_PRG.o: AES/aes_htable_PRG.c
15          gcc -c -O1 -pedantic -Wall -Wextra -DTRNG=0 AES/aes_htable_PRG.c
```

2. Run the command “make” on the command line to generate the output files
3. Run the command “./output h” to helps to understand how to run the code in the command-line.

```
Lookup-Tables> ./output h
CHOOSE THE CIPHER-VALUE AND THE TYPE-VALUE
        CIPHER-VALUE                                TYPE-VALUE
        AES_THIRD:1                             |BASIC:1        |
        PRESENT_THIRD:2                         |BASIC:1        |
        BITSLICE:3                              |BASIC:1        |
        AES_HIGHER_ORDER_INCREASING_SHARES:4    |BASIC:1 , LRV:2|
        AES_RP:5                                |BASIC:1        |

COMMAND TO RUN THE CODE: ./output [CIPHER-VALUE] [TYPE-VALUE]

COMMAND TO RUN HELP: ./output h
```

4. Choose the **Cipher-Valu**e of the scheme to run, along with the **Type-Value**.

5. Run the command **./output [CIPHER-VALUE] [TYPE-VALUE]**

   For e.g .: If you want to run the scheme AES_RP of Type BASIC, you should run the command: ./output 5 1

```
Lookup-Tables> ./output 5 1
Successful execution of AES using RP
(Milli seconds) Overall timings:0.365300

Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
output.exe                   26796 Console                    1      3,260 K
```

6. The output from the code is:
   a) Indicative flag of execution status: success/failure
   b) On successful execution, the offline and the online execution times indicate the pre-processing time and online time, respectively and the memory consumed.

NOTE: The timings and memory consumed when running on a desktop may not reflective of the values shown in the paper, as those values are based on running on a resource-constrained microcontroller.

To Run on the microcontroller

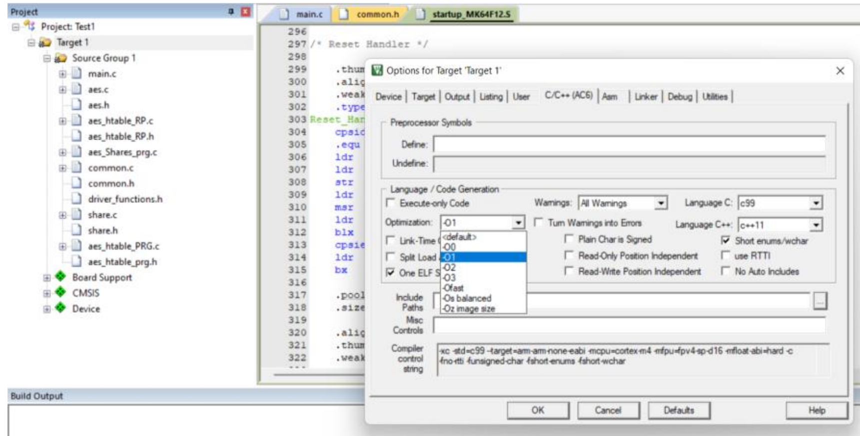
The code is written in C language and hence can be run in **ANY** embedded software development IDE like Keil MDK-ARM by Keil[], SEGGER Embedded Studio, eclipse, Kinetis Design Studio etc.

Remark: We used KEIL IDE and adjusted the settings as mentioned below to compile the code and obtaining results. The similar settings can be found on the respective IDE chosen.

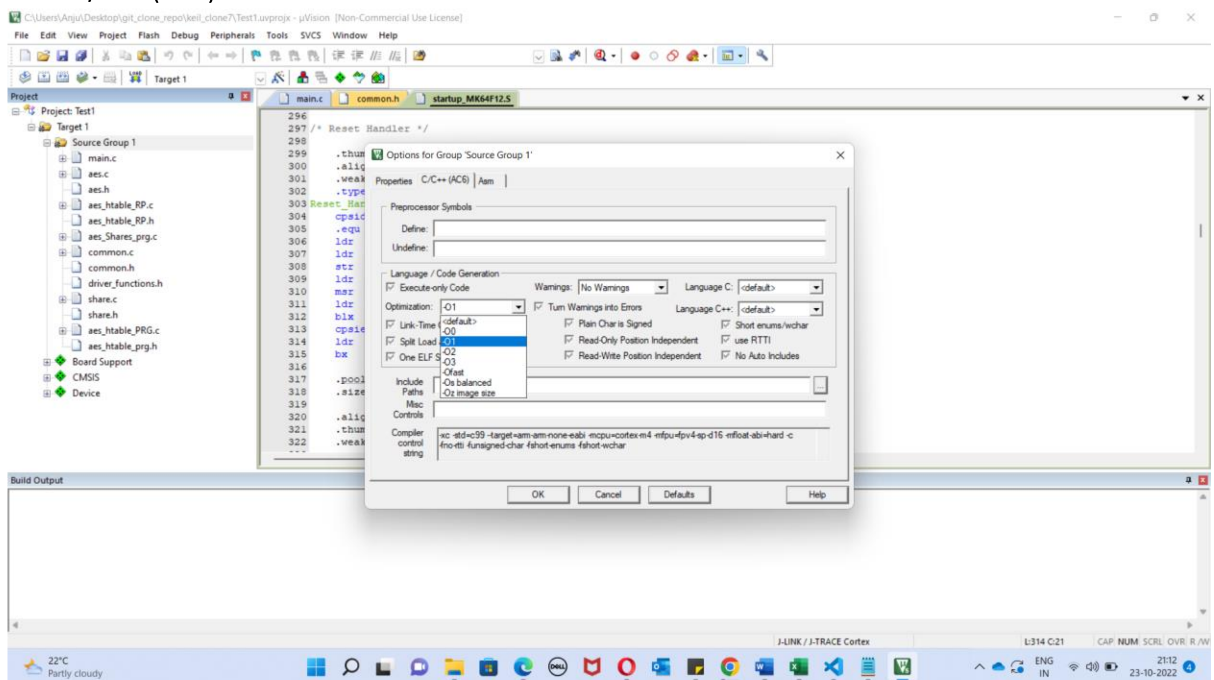
The target device used is NXP-FRDM-k64F development platform. The microcontroller used in the development platform is MK64FN1M0VLL12, based on ARM Cortex-M4 processor having a 256 KB RAM, 1 MB flash memory and a clock frequency of 120 MHz. For debugging the code, we used “JLINK cortex” from SEGGER.

We compile our implementations using the −O1 flag. This flag can be set by following the below steps:

1. Right click on Target and then select “options for Target”
2. click C/C++ (AC6)
3. Change the optimization from default to O1, click ok



4. Right click on Source Group and then select options for “Source Group”
5. click C/C++ (AC6)



6. Change the optimization from default to O1, click ok

To run the code on target architecture

- First build the code by clicking Project->Build Target
- Once build is successful, click on debug ->Start/Stop debug session

Calculating the Memory Consumed

Once the build is successful, the output is given as below:

```
Program Size: Code=8292 RO-data=1436 RW-data=48 ZI-data=4420
```

Total RAM memory consumed by the program can be calculated as:

Total RAM= RW data + ZI-data

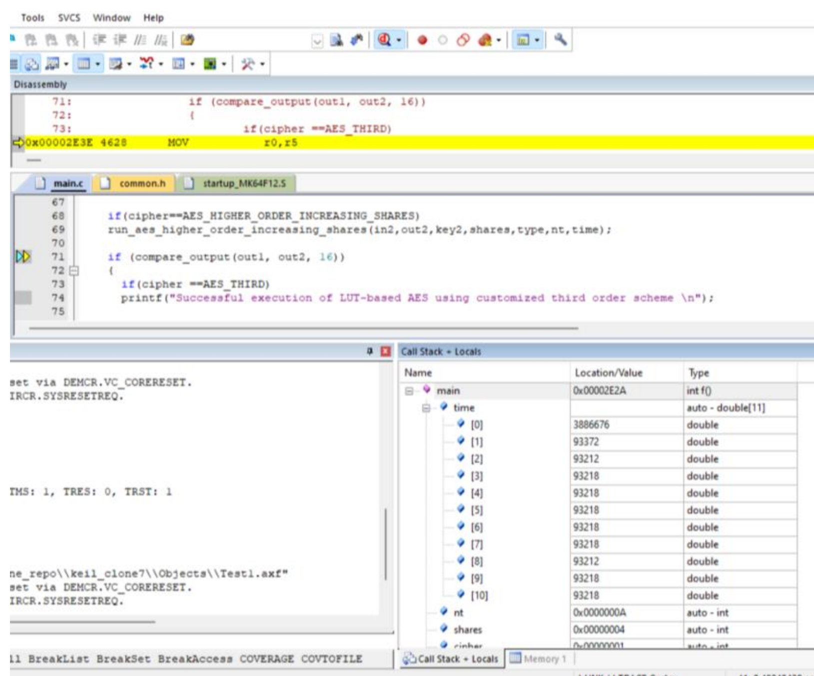
To calculate the clock cycles required:

We calculate the clock cycles required by the scheme using the 24-bit SysTick timer, which is present on nearly all Cortex-M processors.

For scheme AES_THIRD, PRESENT_THIRD, AES_HO_I_S:

- Set the array time[] as size nt+1
- where **time[0] will indicate the offline timings**
- time[i] from i=1 to nt indicates the online time required during each iteration of the function.
- The **average online time** can be calculated by adding time[i] from 1 to nt, and dividing the sum by nt.

Remark : The time[] array of size nt+1 was required, as otherwise the microcontroller was throwing hard fault error.



For Circuit based scheme (BITSLICE/AES_RP)

We added a break point for the function we want to measure the execution times and we read the value from the "states register" before and after the function execution. By subtracting the values before and after the function execution, we computed the timings. Divide the computed timing by "nt".

Register	Value
R2	0x320B6A19
R3	0x978511DC
R4	0x20001158
R5	0x0000000A
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x20000FF8
R13 (SP)	0x20001120
R14 (LR)	0x20000FF9
R15 (PC)	0x00002E8E
xPSR	0x61000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	PSP
States	86606 →
Sec	0.00866060
FPU	

(before executing function)

Registers

Register	Value
R2	0x00000000
R3	0xA0000000
R4	0x20001158
R5	0x20001148
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x040F0000
R13 (SP)	0x20001120
R14 (LR)	0x80905B33
R15 (PC)	0x00002EA6
xPSR	0x81000000
Banked	
System	
Internal	
Mode	Thread
Privilege	Privileged
Stack	PSP
States	10766858 →
Sec	1.07668580
FPU	

(after executing function)

Difference = 10766858-86606=10680252 =10.6M clock cycles

Div by nt (nt=10) = 1.06M clock cycles.