Addis Ababa University
Addis Ababa Institute of Technology
School of Electrical and Computer Engineering

Tirur: A Low Overhead Hardware Extension for Securing Memory

A thesis submitted to the school of graduate studies of Addis Ababa university in partial fulfillment of the requirements for the degree of Master's of Science in Electrical and Computer Engineering (Computer Stream)

By :- Donayam Nega

Advisor :- Ato Fitsum Assamnew
Co-Advisor :- Dr. Salessawi Ferede

April 10, 2019

Chapter 4

Tirur: Design and Implementation

4.1 Introduction

The main design goal of Tirur is to provide an on-chip secure storage that can be used by application software with confidence. Primarily, the design must provide protection against hardware attacks like cold boot, bus snooping and memory replay. Application software can store their most sensitive data inside Tirur's internal storage and Tirur gives it protection against these attacks. Tirur's design assumes uncompromised OS kernel and only gives protection to hardware attacks.

The type of data that can be stored inside Tirur's internal storage is not specific. Application software can allocate memory inside it and store whatever data they presume is sensitive. Tirur with the help of OS will make sure this address space is only accessible by the application software to which it was allocated for.

Tirur is also extendable. Although Tirur's internal storage is limited in size, its page swapping mechanism gives it the flexibility to extend this storage by leveraging the external DRAM module while maintaining security.

In the following sections we will show the detailed design and sample implementations of Tirur.

4.2 Designs

Tirur is designed as a standalone module which can be easily integrated into any existing SoC. Figure 4.1 shows how Tirur fits in high level SoC design. Tirur is standalone as it does not require any support from the rest of the SoC components to perform its functions. It provides its own set of configuration registers and provides simple memory like interface

to communicate with the reset of the SoC. This interface allows it to be integrated as Memory Mapped IO peripheral device which intern simplifies its integration with existing SoCs.

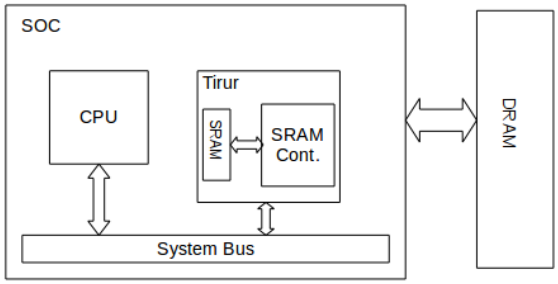


Figure 4.1: Tirur's High Level Design

As shown in Figure 4.1, Tirur is primarily composed of a dedicated data storage (SRAM) and controller. The storage can be used by application software to store their data while the controller controls the overall operation of Tirur and provides security for the data stored in the internal storage. In the following sections, we discuss each component of Tirur, their function and the design rational behind each component.

4.2.1 On-Chip Storage

One of the conditions that makes cold boot attack possible is the DRAMs location which is on relatively easily accessible location, on computer mother board, and these DRAMs are designed as removable modules and can easily be removed and installed in another computer which makes it even easier to execute cold boot attack.

In order to make this easy access more restricted we designed dedicated on-chip storage. Previous research works have tried to use the existing on-chip storage but this has limited the size of the available on-chip storage or compromised some functionality of the SoC. However, with this dedicated on-chip storage we can make larger size of memory available and also we don't have to compromise any previous functionality of the SoC. In addition, this on-chip storage can be used to store any kind of data, and with driver support, it can also be shared by many processes.

Another reason that makes cold boot attack possible is, because the processor does not have control over the initial data stored in the DRAM i.e. the processor does not clear the data inside the DRAM. Tirur's design includes controller which can control the initial data of Tirur's on-chip storage and also helps in keeping the data safe from other attacks.

4.2.2 Controller

Tirur's controller is composed of page swapper, encryption/decryption engine, integrity verification engine, and on-chip MAC storage. The following sub-sections provide the details of these components.

4.2.2.1 Page swapper

Area inside of SOC is limited and expensive. Hence, the max size of the on-chip memory will be limited. Since the size of DRAM is much much larger than the on-chip storage, we can page out the on-chip data to the DRAM when the on-chip storage becomes full. For this we require a paging mechanism.

The page swapper is responsible for moving data in and out of the on-chip memory. It is used to swap the data in the main on-chip memory (Section 4.2.1) or the IV and MAC storage (Section 4.2.2.4). Software can set the operation mode for this module to one of the following options.

- Page out/in vector file :- The on-chip IV and MAC storage has fixed size. When it gets full the system needs to flash it out to DRAM and make it available for the next operation. Similarly, when a memory page whose IV and MAC is not available inside this storage need to be accessed the system needs to swap out the IV and MAC storage. Such operations are accomplished by configuring this model to either page out/in vector file operation modes.

- Page out/in memory :- This mode is similar to the above mode except that the memory that is going to be swapped is a block in the main on-chip storage.

Swapping out the on-chip data to DRAM will expose it to all the attacks we are trying to prevent. So, when data is paged out its integrity and confidentiality must not be compromised. For this we need encryption/decryption and integrity checking mechanism.

4.2.2.2 Encryption and Decryption module

The main purpose of this engine is to maintain data confidentiality. When data leaves the on-chip storage it will be encrypted and when it is paged in it will be decrypted. This will protect the data from cold boot and bus snooping attacks. For the encryption and decryption operations, we need secrete encryption and decryption key or session key. This key must be kept confidential and it should be accessible only within Tirur's boundary. In order to accomplish this, the session key is generated randomly, inside of Tirur and

using an on-chip random generator, at boot time and kept on-chip throughout the keys life time.

The presence of the encryption/decryption module can protect the swapped out data against cold boot and bus snooping attack. However, attack like memory replay attack is still possible. To detect such problem we need an on chip integrity checker.

4.2.2.3 Integrity checker

This module is responsible for computing MAC of Tirur's on-chip data when it is moved in and out of the Tirur's on-chip storage. In this work, we have used a different and simpler integrity checking mechanism than the previous integrity checking mechanisms discussed in Section 2.2.1. This mechanism involves providing integrity checking engine with fixed on-chip MAC storage and allowing software to control and supervise the engines operation without compromising security.

Fixing the size of the on-chip MAC storage reduces the high on-chip MAC/nonce storage memory requirement of the integrity checking mechanism given in Sections 2.2.1.

When certain part of the on-chip data is paged out the MAC of this data is computed and stored in one of the on-chip MAC storage registers. Then, hardware puts the address of this register in another public accessible register so that driver software can read it at the end of page out operation. When paging back this same data, driver software is expected to provide the address for of this MAC storage register. Then, the integrity checker will compute the MAC for the incoming data and compare it with the MAC stored at these address. When miss match occurs between the computed MAC and the on-chip stored MAC, an error flag is set in publicly readable flag register. Driver software can check this flag and choose to discard the data. This process allows the design of the on-chip integrity checker to be simplified. It also gives the integrity checker flexibility on where to store the computed MAC's as it does not need a complex management of the on-chip MAC storage. The integrity checker just has to check for unused MAC storage register and store the newly computed MAC there. Hence, this reduces the complexity in integrity checking engine design faced by integrity tree based hardware.

The MAC computation algorithms chosen for Tirur's implementation are CBC, and GCM. With these algorithms the size of the input data is not fixed i.e. the algorithms take any length of data and provides a fixed length MAC value. Using such algorithms allows Tirur not to have limit on the size of on-chip memory pages allocated to an application software. I.e. unlike integrity tree memory does not need to be divided into equal "M" places allowing even more flexibility for driver software to make this decision.

This MAC computation algorithms also use dedicated key to compute MAC for the data. This key is generated during Tirur's initialization sequence. Due to the availability of this key, either the computed MAC or the IV values can be safely stored in an off-chip storage further reducing the on chip memory requirement. However, an on-chip storage for either the MAC or the IV is still required.

When the MAC/IV storage is full, the hardware will set a bit in one of its configuration registers indicating MAC/IV file full status. For further secure memory allocation requests the driver software need to page out part of the MAC/IV storage. The driver software is also responsible for keeping track of all the paged out on-chip data and the corresponding MAC/IV addresses.

4.2.2.4 Initial Vector and Mac Storage

Tirur requires on-chip MAC/IV storage to protect data against replay attack. When certain portion of the on-chip memory or the MAC/IV storage is flashed to DRAM, its MAC is computed and stored in this module. Although, and to the best of our knowledge, the IV's used during encryption can be stored safely in an off-chip storage, in the prototype implementation of Tirur both the IV and MAC are kept on chip to prevent any unseen future attacks.

4.3 Implementation

When Tirur first boots, it will be in an ideal state. Tirur's first task is to clear all the on-chip memory data i.e. set it to zero. Clearing this data will make sure that any remnant data won't be available on the on-chip memory. Hence, giving further protection to the on-chip data from attacks like cold-boot. Following this, Tirur initializes the session encryption/decryption keys, generates random number for the IV generator counter, generates hash key for GCM tag generation and finally enters ideal mode where it waits for further commands. Figure 4.2 shows this sequence in flow chart.

4.3.1 Tirur's Operation Modes

Tirur has 6 operation modes. these are Ideal Mode, Page Out Memory, Page Out IV, Page In Memory, Page In IV, Read Through and Write Through. Tirur is configured for its different operation modes through the driver firmware.

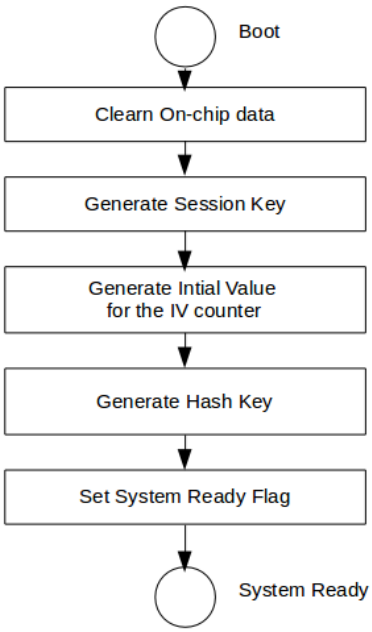


Figure 4.2: Tirur's Boot Flow Chart

4.3.1.1 Ideal Mode

In this mode, Tirur's different circuity's are disabled and it is waiting for commands. From this point, based on the current required operation driver firmware can set its operation to one of the following modes.

4.3.1.2 Page Out Memory/IV

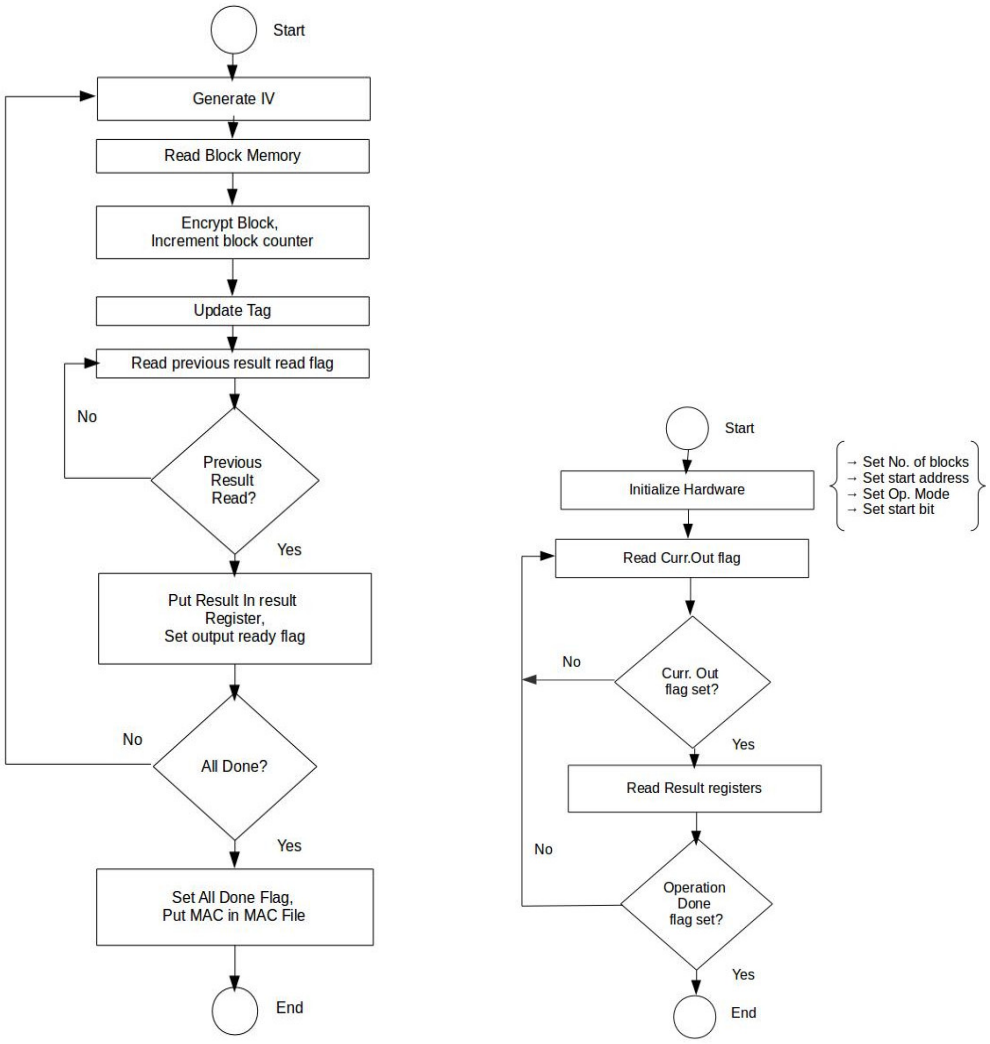
In this mode, Tirur is configured to swap out the on-chip memory or the MAC/IV storage to the DRAM. To use this mode driver firmware needs to first properly configure Tirur and set it to this operation mode. Driver firmware need to set the number of blocks and the start address of the memory block to page out and finally set the start bit to start the operation.

During this operation, Tirur generates new IV as seed for the encryption operation; reads a block of memory; and encrypt it. Then, it updates the current MAC value to reflect the newly encrypted block. Finally, Tirur puts this encrypted block in the output result registers and sets the current output ready flag bit.

Tirur internally keeps track of the number of blocks paged out and pointer to the current memory address that is being paged out. When all the initially configured number of blocks have been paged out, Tirur sets the operation done bit and puts the computed MAC in the MAC/IV register file, copy the address of this MAC value to last used MAC

address register and finally sets itself to ideal mode.

In our current implementations, during each operation software must pull the current output ready flag and read the results register when the flag is raised. When all the page out operation is complete, software needs to read the value of the last used MAC address register and save it for the later use as this value will be required during the page in operation. Figure 4.3 gives the flow chart for the page out memory/IV operation.



(a) Hardware Flow Chart (b) Software Flow Chart

Figure 4.3: Page Out Memory/IV Flow Chart

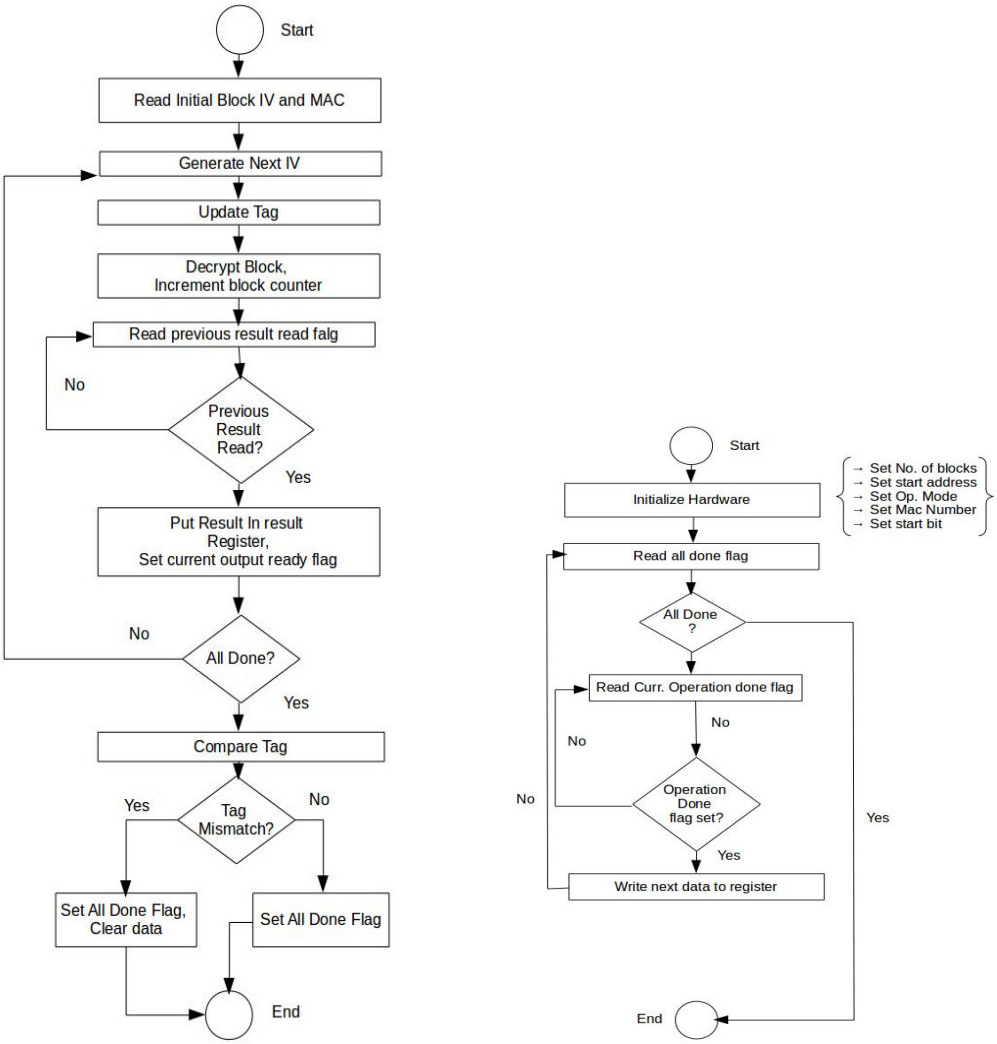
4.3.1.3 Page In Memory/IV

This mode is used when driver software needs to page back memory blocks. Driver firmware first sets the memory address to put the data in, the number of blocks to page

in, the MAC number to use for verification, provides the first block to page in, and sets the start operation bit.

Tirur then reads the IV used from the MAC/IV storage, updates the tag, decrypt the block and writes it to memory. After the initially specified number of blocks have been paged in, Tirur will compare the final computed tag with the on-chip stored tag. If mismatch occurs it will set tag verification error flag and clear all the currently paged in data.

In the current implementations of Tirur, driver software has to pull for the ready flag before writing in the next block of data to be paged in. Software also has to check for the error flag at the end of the page in operation and take appropriate action in case of an error. Figure 4.4 gives flow chart for the page In memory/IV operation.



(a) Page In hardware flow chart

(b) Page in software flow chart

Figure 4.4: Page In Memory/IV Flow Chart

4.3.1.4 Read/Write Thought

This mode is used when the system need to put plain text data inside the on-chip memory or when on-chip data is moved to CPU to perform computation on it. When computation is to be done on the on-chip data it will be copied to CPU registers. At this point a context switch operation by the OS could flush the data to the main memory. To prevent this, any computation on the on-chip data must be done inside of interrupt service routine (ISR) by disabling global interrupts so that the OS won't interrupt and flash sensitive data to the DRAM. The ISR can also clean CPU registers at the end of the routine.

Tirur's hardware implementations were done for a 32bit SoC architecture (Rocket

Core). In the follwing section we present the implementation details and choices we made on these implementations.

4.4 Hardware Implementation

The hardware implementation of Tirur was done using Chisel. After all the modules were implemented and tested using Chisel's testbench their corresponding Verilog code was generated using Chisel-3 Verilog generator. Then, the generated Verilog code was ported to the new Xilinx's HDL design, synthesis and analysis tool(Vivado Design Suite version 2017.2) and synthesized for FPGA target. The following sections present hardware design details of each sub-component of Tirur.

As shown in Figure 4.5 internally Tirur is composed of on-chip storage (SRAM), data page in/out controller, MAC/IV storage, configuration registers, a random number generator and counter.

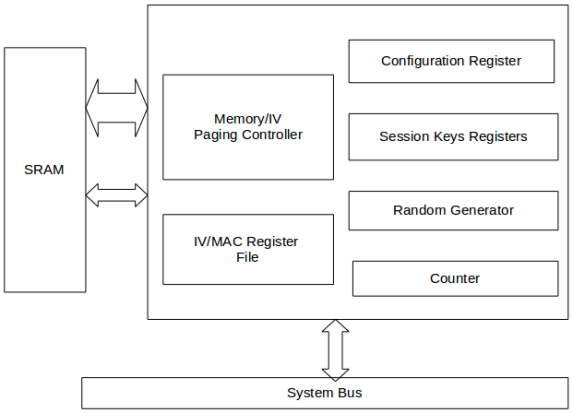


Figure 4.5: Tirur's Internal Details

4.4.1 On-Chip Storage

The On-Chip storage was implemented as dual port SRAM. Each memory cell is 32bit wide. The SRAM controller uses the 32bit bus when writting the initial plain text data and also when data is moved to and from the CPU. The 128bit bus is used during the paging operations. This dual port implementation was chosen to allow faster data read during memory paging operation.

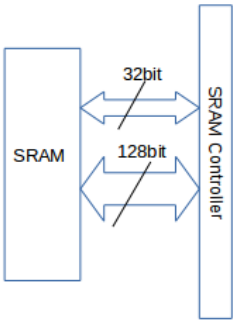


Figure 4.6: On-Chip Memory

4.4.2 Page Swapper

The page swapper as shown in Figure 4.7, consists of encryption/decryption engine, integrity checker, paging controller and write through circuitry.

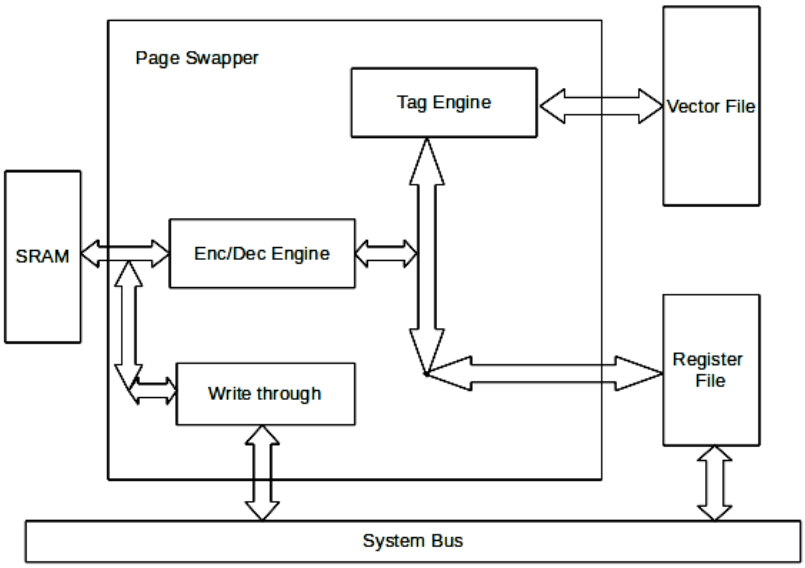


Figure 4.7: Page Swapper

Encryption/Decryption Engine

This sub-circuit implements the AES encryption algorithm. Inputs to this engine are read either from the register files or from the on-chip memory. Output results are also written to the on-chip memory or register files. In our prototype, we have implemented two versions of this engine.

The first is loop unrolled implementation where the AES engine is implemented in

such a way as to perform the encryption/decryption in just one clock. The second version performs similar operation in 11 clock cycles. For stream ciphering both CTR mode and CBC mode of operations have been implemented.

Tag Engine

This engine implements Galois filed (g-128) multiplication for MAC computation. It is implemented to perform the multiplication in just one clock. The output of the encryption module is feed to this engine to compute the MAC during paging out operation and the input blocks are feed to it during the page in operation. When a started paging out operation completes, the computed MAC is stored in the vector file and when a paging in operation completes the newly computed MAC is stored in temporary register and compared with the corresponding MAC value stored during page in operation.

Write Through

This circuit writes the incoming data or reads from the on-chip memory directly bypassing the encryption/decryption engine. This mode is activated based on the chosen operating mode.

4.4.3 Random Generator

The random generator is implemented as 16bit random generator using linear feedback shift register circuit. Tirur requires 64bit random number for initial vector and 128 bit random number for session key. These numbers are generated from this random generator module in 4 and 8 clock cycles respectively.

4.4.4 Configuration Registers

These are a set of registers through which Tirur can be configured and its different statuses can be read from. These registers contain flags like module ready, start & stop, error, current operation done and etc. Tirur's different operating modes are also configured through these registers. In addition data to be paged in/out of the on-chip storage is written/read to/from these registers.

On the current prototype implementation Tirur has 10 configuration registers. Summary of the available registers is given in the Table 4.1

Address	Name	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	Bit 5	Bit 6	....	Bit 31
0	TConf0	Start	MD0	MD1	MD2	-	-	-	-	-
1	TConf1	Memory pagging start address								
2	TConf2	Number of pages								
3	Vect	Register File Number								
4	ERRS	VFS	TVEF	-	-	-	-	-	-	-
5	Res0	Page out result /page in value 0								
6	Res1	Page out result /page in value 1								
7	Res2	Page out result /page in value 2								
8	Res3	Page out result /page in value 3								
9	Flags	-	MRF	CRF	-	ODF	DIR	DOR	-	-

Table 4.1: Register Files

TConf0 Register

Bit0 Start: Tirur Operation start/stop bit

0 = Tirur is in off mode and no operation is under progress

1 = Tirur is on and working according to the configuration

Bit3-1 MD3:MD0: Tirur operation mode select bit

000 = Ideal mode/ no operation

001 = Page out Memory

010 = Page out vector/mac file

011 = Page in memory

100 = Page in vector/mac file

101 = write/read through mode

Bit32-4 Unimplemented

TCon1 Register

Bit31-0 Memory page start address

- The start address of on-chip memory to page in/out the data

TCon2 Register

Bit31-0 The number of memory pages to move

Vect Register

Bit31-0 The address for initial vector and mac storage used page out/in the current memory blocks

ERRS Register

Bit0 VFS:- Vector file status,

0 =Vector file is not full

1 =Vector file is full requires paging out the vector file

Bit 1 TVE:- Tag verification error

0 =Tag is correct

1 =Error tag verification failed

Res0 Register

Bit31-0 The first lower 32 bit data to page in/out

Res1 Register

Bit31-0 The second lower 32 bit data to page in/out

Res2 Register

Bit31-0 The third lower 32 bit data to page in/out

Res3 Register

Bit31-0	The Higher 32 bit data to page in/out Read or write to this register will start the next operation automatically

Flags Register

Bit0	Unimplemented	
Bit1	MRF: Module ready flag	
	0	=Module is not ready, initializing
	1	=Module is ready
Bit2	CRF: Current output ready flag	
	0	=The current round output is not ready, result registers are not valid
	1	=The current round output is ready, result registers are valid
Bit3	Unimplemented	
Bit4	ODF:- Operation done flag	
	0	=The last started operation is not completed yet
	1	=The last started operation is completed
Bit5	DIR :- Data Input Ready (Read only)	
Bit6	DOR : Data Output Ready (Read only)	

4.4.4.1 MAC/IV Storage Module

This module is a set of fixed size on-chip register file which is responsible for storing initial vectors, used during encryption, and computed MACs. Every register slot is appended with single bit that indicates whether it is used or not. This register file is manged automatically by the hardware. The data contained in this file is never read as plain text outside of the module. When this storage is full, driver software can page it out during which it is encrypted. For this storage a minimum of two blocks are swapped at a time.

This is because one of the slots is required to store MAC for the current blocks (blocks from the MAC/IV storage) being processed and the other slot will be available for use. Similar to memory page in operation, when MAC/IV storage is page in, the corresponding IV and MAC data are read and their address is automatically marked as free for next use.

This module also includes a simple controller which is responsible for searching and returning the address of any available unused slot; and updating the status of each slot. Figure 4.8 shows the block diagram the MAC/IV storage module.

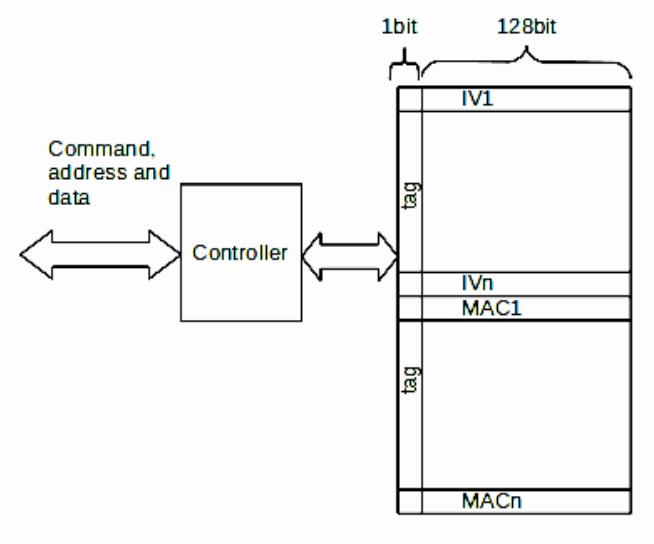


Figure 4.8: MAC/IV storage

In the current prototype implementation MAC/IV storage has 7 commands

1. Ideal (0) :- With this command the module is set to ideal mode and performs no operation

2. ReadVector (1) :- With this command the module reads value of initial vector at the specified address and put the result in the data bus

3. WriteVectorAndMAC (2) :- With this command the module writes the provided initial vector and MAC value to the specified address

4. ReadMac (3) :- With this command the module reads value of MAC at the specified address and put the result in the data bus

5. FreeIVVectorSlot (4) :- With this command the module clear the tag bit at the specified address indicating that the memory slot is free for next use.

6. FreeAll (5) :- With this command the module clears all the memory slots making them available for next use.

7. GetFreeAddress (6) :- With this command the module searches for free memory slot and returns its address.

4.4.5 Integration with the Rocket Core

Tirur was integrated with the rocket core as Memory mapped input output(MMIO) device. Figure 4.9 gives the SoC Tirur was integrated with. The SoC contains 32bit rocket-core (RV32MAC); separate instruction and data cache; and two interrupt controllers, the core local Interrupt (CLINT) which generates per heart software interrupts and timer interrupts and Platform level interrupt controller(PLIC) which supports 127 global interrupts with 7 priority levels.

Tirur was connected to the 32bit TileLink-UL peripheral port. In our prototype implementation Tirur has base address of 0x52000000 and spans size of 0x10000. We used this address range as it was free address range and does not conflict with other existing peripheral devices.

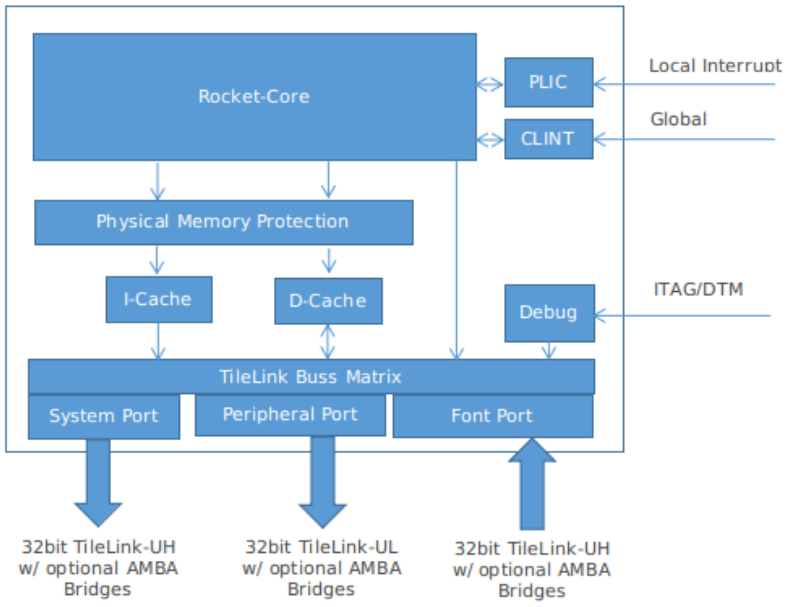


Figure 4.9: Rocket Chip

4.5 Driver Software

To introduce Tirur to the OS, a driver software is required. The driver will be responsible to control and coordinate Tirur's operation. The driver software will have the following functions.

Control memory allocation and deallocation inside Tirur's address space

When application software requires a secure memory, allocation is done through this driver software. This software will keep track of the available free memory address spaces and allocate them appropriately.

Control on-chip memory paging operation

The driver firmware is responsible for controlling memory paging operation. It automatically flashes out the on-chip memory pages when it is full. It also pages in memory blocks when they are requested for read or write.

Control MAC/IV file paging operation

The on-chip MAC/IV file is also controlled by this driver software. When different memory pages are moved between the on-chip memory and the DRAM the appropriate MAC and IV values must be available on the on-chip MAC/IV storage memory. The driver firmware is responsible to keep track of the MAC and IV values and memory pages that are encrypted with them. The driver firmware also makes sure that the appropriate MAC and IV value is present in the on-chip memory when required.

Perform read and write operation on behalf of application software

All the read and write operation to the secure memory goes through the diver firmware. This allows the driver to control access to on-chip memory i.e. it can prevent applications from accessing memories outside their allocated ranges. This also allows the driver firmware to page in the appropriate memory pages and MAC and IV values at the right time. Figure 4.10 shows the communication between application software, driver software and hardware to perform memory allocation and read and write operation.

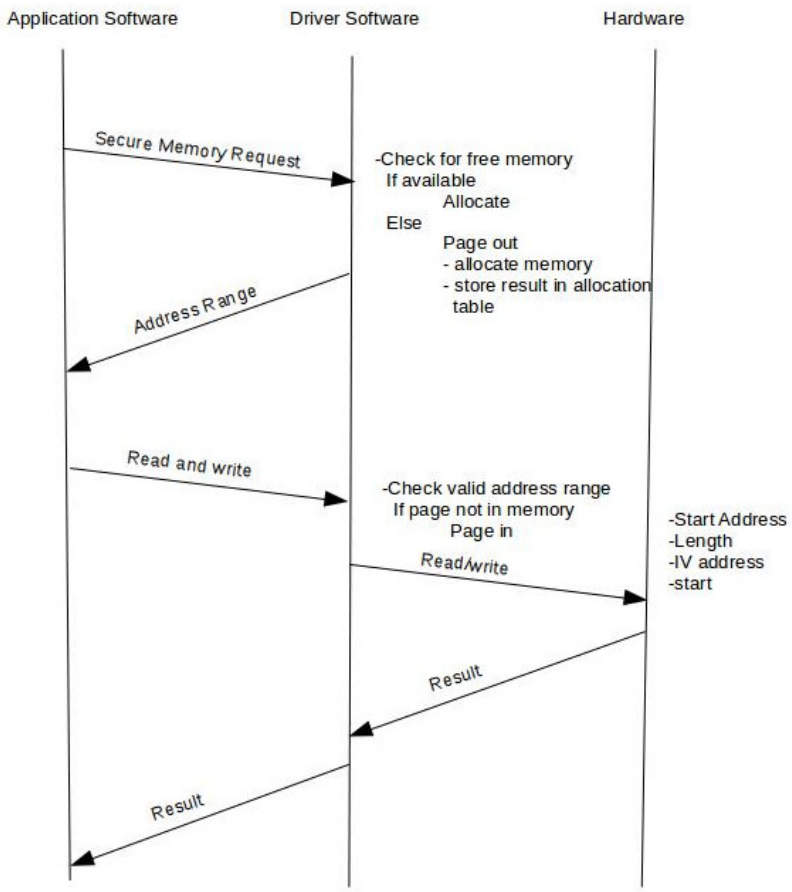


Figure 4.10: Data flow between application software, driver software and hardware

As can be seen from the flow chart, when application software needs a secure memory, it sends secure memory request to the driver software. Then the driver software will check for unused on-chip memory or if the on-chip memory is full it swaps out a process and allocates address range for the requesting process. Once the application software is allocated some address range, it can read and write data on this address range through the driver software.

4.6 Summary

In this chapter we have presented the design ideas and implementation details of Tirur. Tirur was designed to as a secure storage for application software. To this end, it has its own on-chip secure storage which can be used to store any application data. Besides this storage, Tirur includes controller which gives protection against unauthorized access of data in the storage. This controller consists of encryption/decryption engine, integrity

checker , MAC/IV storage and page swapper.

In order to better see the performance of Tirur, we have integrated Tirur with existing SoC (Rocket Chip) as MMIO. Tirur also requires a driver software in the OS side. This driver software is used to configure and properly operate Tirur. The driver software is also responsible to keep track of Tirur's secure pages located in DRAM.