Hardware Implementation of Secure Communication in a Bus-Based Multi-Core System Using Tiny Encryption Algorithm

Team Dja Dja

Problem Statement

There is a always a need of communicating between two individual cores which are a part of a multicore system. In the case of one core being hacked into, this compromised core can be used to snoop upon the communication between the other individual cores, owing to the bus topology used to tie the cores together. The idea of this project is to alleviate such a scenario by securing the flow of data from one core to another, by using an encryption scheme, thereby preventing unwanted snooping of data.

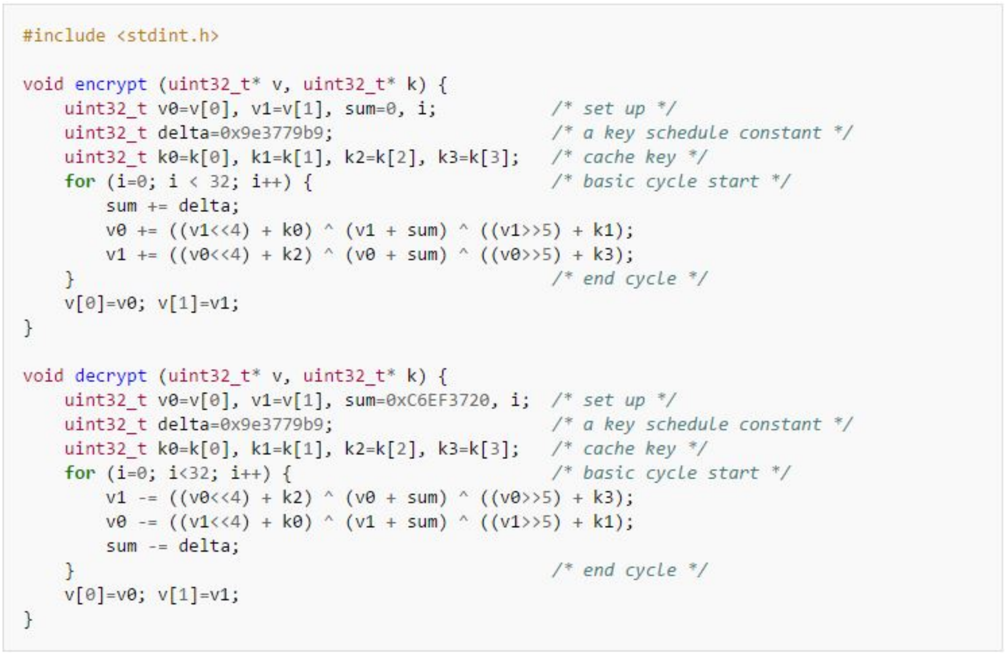
Need for Solution

The most straightforward issue arising from the problem is lack of security of the data. This issue holds paramount importance while sensitive data is being transmitted/received between the cores.

Also, one compromised core can be used to alter the behaviour of the system, in many ways. Trojan activity in one of the cores can also give rise to unwanted operations in the system. In-order to prevent this unusual source of activity, it is essential to secure the communication paths within the system.

Idea Abstraction

We propose to solve this problem by using an hardware encryption method to encrypt data which is transmitted via the bus. Each time a core requests to read contents the of memory from other any other core, the data will be encrypted and then transmitted on the bus.
TEA (Tiny Encryption Algorithm) is a simple and fast symmetric encryption method. In the process of encryption and decryption, only addition, subtraction and XOR are involved, which make it a good hardware encryption method. In all, 16 rounds of encryptions will be executed (we can run multiple rounds in one clock cycle if core clock is slow enough compared to the computation latency). The algorithm has been show below in detail.



```c
#include <stdint.h>

void encrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;           /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}

void decrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i;  /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i<32; i++) {                         /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

Fig 1: Simple implementation of TEA algorithm

Since TEA is quite simple to implement, it may not take very long to use brutal force to hack into the bus and snoop on the data. So the target here is to renew the keys periodically to avoid hacking using brute force. We will use a centralized key generator & distributor to implement this. The generator will use random numbers to generate random new keys.We can estimate the number of cycles required using the brute force method to hack into the system, using this data, set the refresh period of the key-generator smaller than this time period.

The overall architecture has been shown below.

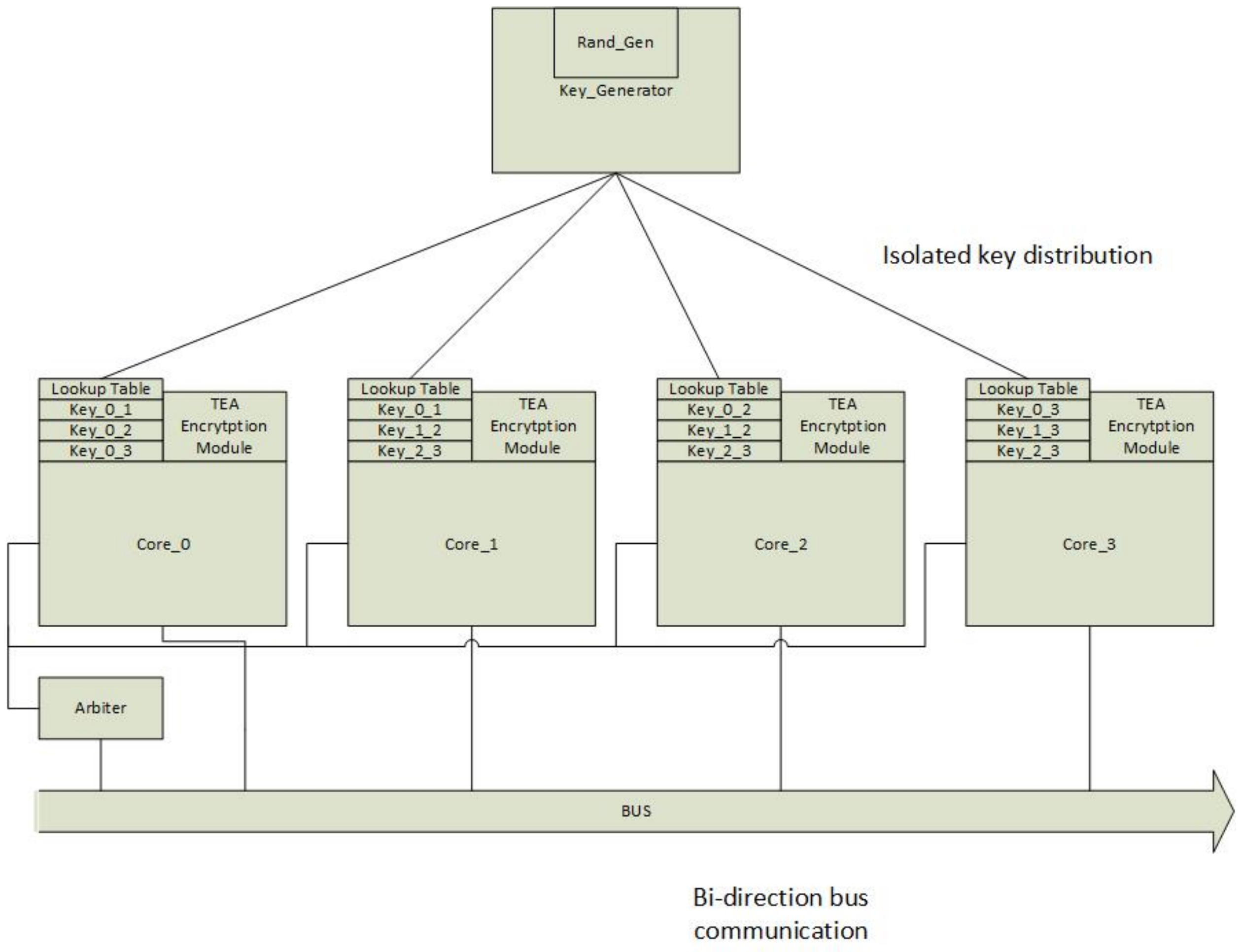


Fig 2: Architecture of the proposed system

Project Development

First, we aim to implement the encryption/decryption as well as key generation/distribution modules ie. the architecture shown in Fig 2, in Verilog. This is the most important part of the project.

Once the blocks are ready, we aim to integrate the modules with existing open-source cores (from OpenCores, e.g. OR1200) and bus (also from OpenCores, e.g. Wishbone; or a custom made bus); and use VCS to run the required simulations.

Evaluation

First, we will compare our algorithm with several popular encryption algorithms in terms of security and implementation complexity (statistically). We expect to see our algorithm achieving relatively higher level of security but taking far less time (lesser number of cycles) to compute the results.

In the next step we will develop test programs (or using existing multithreaded benchmarks) to compute the time overhead because of the integrated encryption logic. We expect to see a small percentage of increase in execution time (If we use C++ to build a functional model, we might need to inject bus traffic based on traces. Traces can either acquired from Doowon, or generated by ourselves)

We plan to run the same program (or inject the same traffic pattern), and then compare the execution time with and without encryption.

Timeline

10/23 checkpoint 1	Hardware module finished Timing estimation finished System integration started
11/13 checkpoint 2	System Integration finished Debug phase Completed
12/04 checkpoint 3	Data gathering phase Ideas and implementations checked with professor and GSI
12/10 checkpoint 4	Everything done