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

Team Dja Dja

Proposed Solution to Unsecure inter-core communication:

We propose to solve the problem of unsecure inter-core communication by using a hardware encryption method to encrypt data which is transmitted via the bus. Each time a core requests to read contents of the 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:

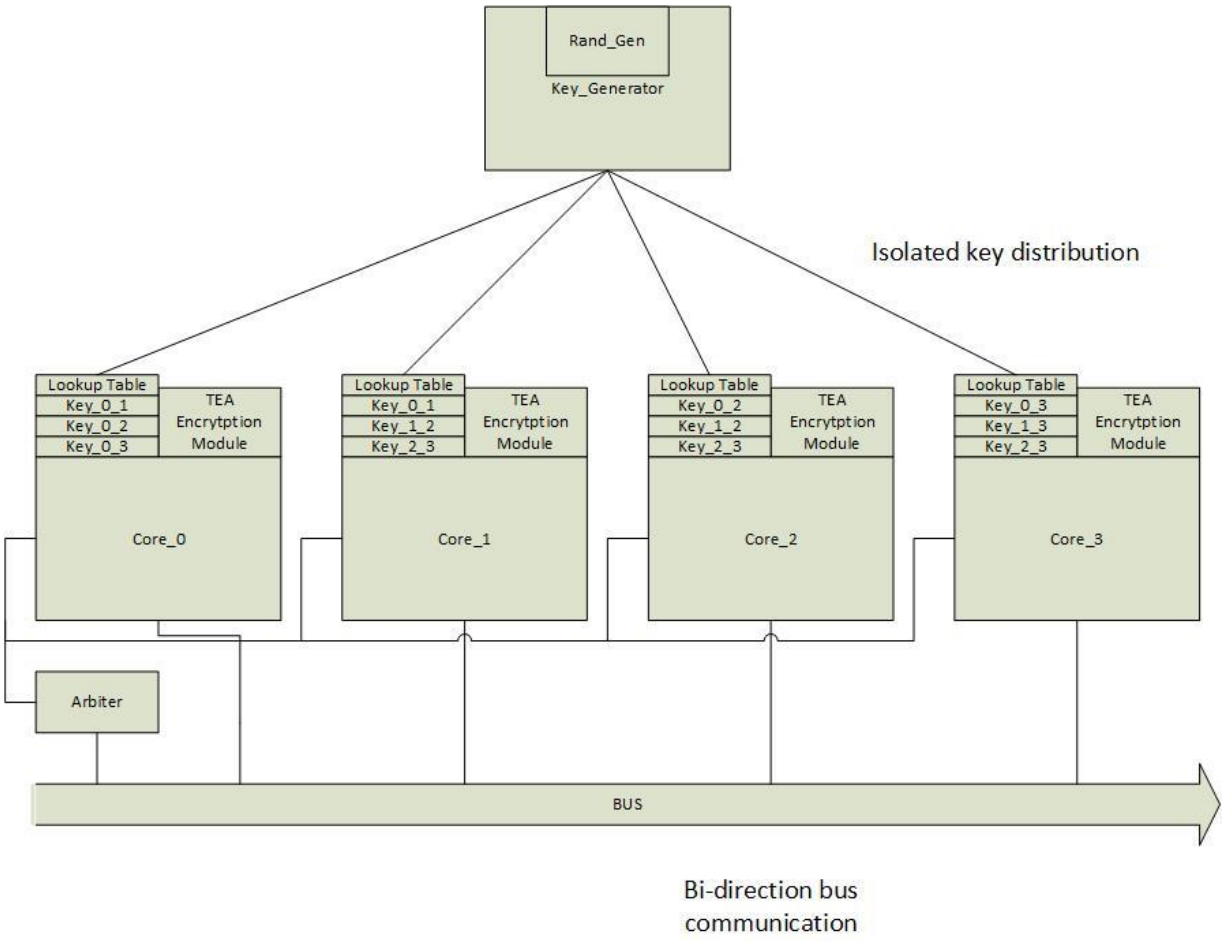

Fig 2: Architecture of the proposed system

Progress:

Key generator:

The key generator module will generate random keys and distribute to all cores once in a specified time interval. A local timer is used to decide the distribution of these keys. For 4 cores scenario, (4 choose 2) = 6 keys will be generated each time. We use $random() function provided by Verilog instead of an explicit random number generator hardware for simplicity. 16 cycles "pseudo latency" was introduced when generating one random number in order to mimic hardware functionality. A simple selection FSM is used to decide which pair of cores the newly generated key will get assigned to. This module was designed in Verilog and its functionality was tested using VCS.

Encryption & Decryption Module:

The original algorithm suggests that at-least 16 rounds of computation for the encryption/decryption process. In this design, 2 rounds of computation were done per one cycle, and totally 8 cycles to finish an encryption/decryption cycle. A pipeline structure was introduced here to maximize the throughput of the key-generator system.

Besides the computation module, there is a lookup table to hold the keys generated from key generator. This table holds two sets of keys; one, the present set, and the other one, the previous key set, just in case the message was encrypted by the old key. Every time a new set of keys reach this queue, the previously used keys are moved to old-key table, the ones currently in the old-key table are kick out, and write the new keys are then written into the new-key table.

This module was modeled in Verilog and the functionality was tested using VCS.

Cores And Wishbone bus model:

The 32-bit open-source core OpenRISC 1200 is used in this project. The RTL for this core was obtained from opencores.org. A salient feature of this core is the inbuilt interface for a 32-bit Wishbone bridge, somewhat simplifying the tedious upcoming task of integration.

The open-source Wishbone bus was selected for this application owing to its simplicity and ease of implementation. The Wishbone-bus builder was obtained from opencores.org, understood, and modified to generate a 32-bit 'shared-bus configuration' as per our requirement. The RTL generated by the builder can be easily integrated with the obtained core.

Challenges:

The main aim here is to figure out an optimal time-interval to update and distribute keys to cores, as a lot of tradeoffs are involved here. Also, in order to achieve the highest possible level of security with the algorithm, the absolutely random nature of the key generator should be exploited, which is quite challenging to develop.

The open-source Wishbone-bus builder script generates a VHDL version of the bus. Trying to simulate the same using VCS was time-consuming owing to limited experience with the same. The idea of converting the VHDL to Verilog was not successful owing to fundamental limitations in the generator script.

Also, the next step, ie integrating the modules together is expected to cumbersome owing to the auto-generated, as well as custom blocks used in the design.