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

Jianchao Gao
Department of Electrical Engineering and Computer Science
University of Michigan
jianchao@umich.edu

Dike Zhou
Department of Electrical Engineering and Computer Science
University of Michigan
zhoudike@umich.edu

Ameya Rane
Department of Electrical Engineering and Computer Science
University of Michigan
arane@umich.edu

***Abstract*—Security attacks are the most common way of phishing in today's internet driven world. Secure communication between multiple cores in a multi-core system is of utmost importance, especially if secure data is being translated. Various methods have been proposed to make this communication secure, either by encrypting the data being translated, by encrypting the bus etc. This project implements the bus based encryption technique using Tiny Encryption Algorithm, a simple, yet secure commonly used encryption algorithm. The security of this module is tested using analysis specified in this report, with substantial enhancements outlined, thereby giving our work an upper-hand over existing similar work.**

Keywords—TEA, security, bus encryption, bus-based multicore

I. Introduction

Owing to the inherently diffused nature of communication systems today, there is a flow of data to and fro from the servers/nodes/transceivers etc. in a communication network. Understandably, there are lot of ports which can be tapped into, and snooped upon, to read or observe the data being sent across. Shared resources, like a communication channel, appear to be the hotspot for data translation. Compromising the security of such a channel (a bus in this case) can expose the entire system to unauthenticated nodes, thereby permitting unsecure data transfer. This is a very serious issue if important/classified data is being moved around, and has the potential to cripple the financial sector, military communication etc. Hence, secure communication between different nodes in a communication channel is of paramount importance, especially in today's information technology driven world.

System security can be enhanced by numerous means, with solutions ranging from architectural reinforcement, to software solutions. At the micro architectural level, we propose to solve the above mentioned problem plaguing inter-core communication by using a hardware encryption method to encrypt data which is transmitted via the bus. The security of this bus encryption scheme is dependent on the inherent security of the encryption algorithm being implemented. A simple, fast and secure encryption would be to use any block encryption algorithm, with the Tiny Encryption Algorithm (TEA) being the simplest option. [1]

Along with the inherent secure behavior of the encryption algorithm, additional security features are provided by the micro-architectural changes implemented in this project. Also, frequent refreshing of the keys used for encryption/decryption is one way of enhancing the security of the already secure system. A system almost entirely immune to timing attacks, brute force and some smart attack techniques is developed, with an average performance overhead of 13%, which is much lower than those designed for the same purpose, thereby giving substance to our claim of building a better system.

Fig. 1. Data Flow of Tiny Encryption Algorithm

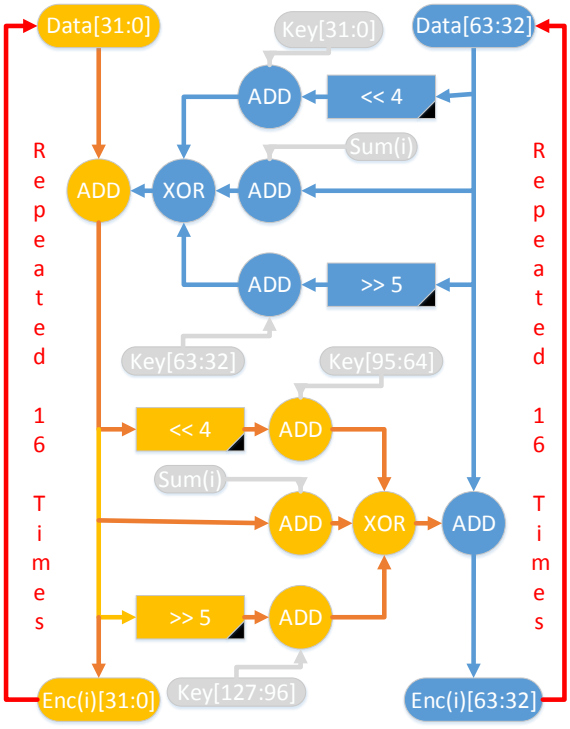


Fig. 2. Reference C code of Encryption Module

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

Fig. 3. Reference C code of Decryption Module

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

II. Background

Tiny Encryption Algorithm (TEA) [1] is the encryption method applied in the proposed design. This is a simple, yet secure algorithm, hence commonly used for simple implementation of secure communication systems. It is a symmetric encryption algorithm with characteristics of being very fast, simple (only shifting, adding, subtracting and XOR), and easy to understand. The data flow of one round of computing has been showed in Fig. 1. In order to perform an encryption effectively, a minimum of 16 rounds of computing need to be performed. To decrypt a message, an module with similar structure of encryption one can be used. The encryption and decryption modules' pseudo-codes are shown below in C in Fig. 2 and Fig. 3.

III. Proposed Architecture

The architecture of proposed design is shown in Fig. 4. The proposed design is an adaption of a multi-core system designed for EECS 470, consisting of 2 cores, and one memory module, integrated together using a simple bus with an added arbiter. Each of the cores has a LUT(lookup table) used for storing the keys, used for the encryption and decryption procedure. A random key generator module, called as "Cloud" is used to distribute the keys required for filling the LUTs. For each core, 2 set of keys will be stored for functional correctness. E.g. for core_0, one set is key_0_1, which is used to encrypt/decrypt messages between core 0 and core 1. The other set is key_0_m, which is used to encrypt/decrypt messages between core 0 and memory.

Each time a core requests to read contents of a memory cached in any other core, the data will be encrypted and then transmitted on the bus by the source core. In the process of encryption and decryption using TEA, only addition, subtraction and XOR are involved, which make it a simple and 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). Since TEA is quite simple to implement, it may not take very long to use brute force to hack into the bus and snoop on the data. So the aim here is to refresh the keys periodically to avoid brute force attacks. The generator will use a random number generator module to generate the random new keys required for the encryption/decryption. We can estimate the number of cycles required using the brute force method to hack into the system, and using this data, set the refresh period of the key-generator smaller than this time period. Basically, the strength of the encryption algorithm is reinforced to almost unbreakable levels using the key-refresh mechanism.

Fig. 5 shows the data flow of our design. Here we take the instance of this scenario: core_0 wants to load the value stored in memory with an address, and when the operation is done, it will respond with the requested value. In a conventional processor design, the flow will be as follows: (1) core puts the address on the bus together with request; (2) memory captures the request from the bus; (3) after the latency period, memory completes its loading and puts the data on the bus together with response signal; (4) core captures the data from the bus. Since the data on the bus can be fetch by any other nodes in this system, the system security is compromised if one core running any malicious software captures the data translating between other modules. (In this instance, we can assume core_1 can fetch the data translating between core_0 and memory, which should not be touched by core_1) In order to prevent information leaks to other core through bus, the proposed design will make sure all data on the bus are encrypted. Take the same scenario as before, the data flow of our proposed design will be: (1) Source core sends the address along with the request to encryption module, which will encrypt the address, together with the request. After this process is done, the cipher text is put on the bus (2) Memory sees the request signal and fetches the cipher text from bus. It will use the decryption module to reveal the

address in plaintext, and then send decrypted address to memory; (3) after a few cycles, the memory finishes loading and then send the data to encryption module together with response signal. This module will convert the data into cipher text and then put it along with the response signal on the bus; (4) The requesting core observes the response signal and will fetch the encrypted data. This data will be decrypted by decryption module and will be sent to the core. As a result, the communication on bus is encrypted and only the target module can decrypt it. So under this scenario, even though core_1 can fetch the data on the bus, it cannot decrypt it, as it does not have the corresponding key, thereby preserving the integrity of the system.

IV. DESIGN ANALISYS

The first half of this section is based on the implementation of encryption/decryption modules. In order to hide the latency due to encryption/decryption, these encryption/decryption modules are pipelined into multiple cycles. However, partitioning this module into different number of stages result into different impact on area and performance overhead. Generally, the more cycles to partition the module, the lesser the combinational delay required to run each cycle, but the more the total area. Also if the combinational delay is smaller than the latency of the core, it is negligible for further pipeline stages.

A. *Area & Performance Overhead*

The base core we used has an area of 12538496.545692 um2. Table I mentions the area of each individual module that have been used in this design, as well as the total area and delays. The area overhead is calculated as

$$Area\ Overhead = \left(\frac{Area_{design}}{Area_{base}}\right) - 1 \qquad (1)$$

The performance overhead in terms of delays is calculated as follows:

$$Performance\ Overhead = \left(\frac{CPI_{design}}{CPI_{base}}\right) - 1 \qquad (2)$$

Fig. 4. Architecture of Proposed Design

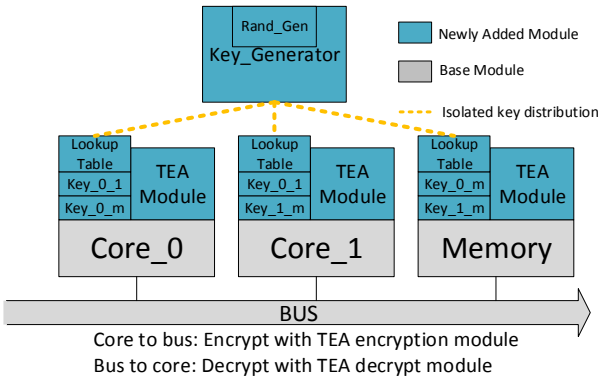


Fig. 5. Data Flow in Proposed Design

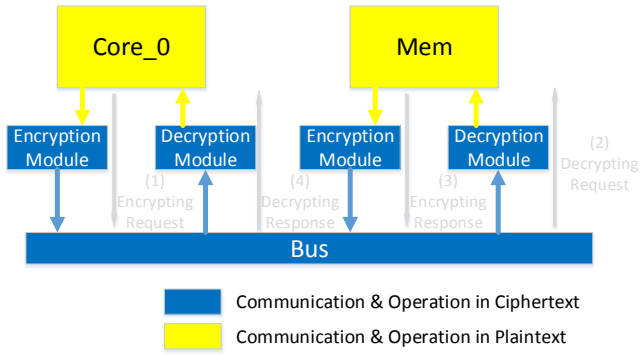


TABLE I. SUMMARY OF AREA AND PERFORMANCE OVERHEAD

Module	Total Area (um^2)	Latency (ns)	Area Overhead	Performance Overhead
Base Core	12538496.545692	6.0	N/A	N/A
Enc/Dec_ Mod 1 Cyc	9917197.113	7.0	81.01%	81.01% / 7.05 %
Enc/Dec_ Mod 2 Cyc	10128505.18	5.5	82.69%	0.8269/0.1393
Enc_Dec_ Mod 4 Cyc	11215335.24	4.0	91.36%	0.9136/0.2690
Enc/Dec_ Mod 8 Cyc	12137623.71	2.5	98.72%	0.9872/0.5172
Lookup Table	118469.0638	N/A	N/A	N/A
Key Generator	121309.122675	N/A	N/A	N/A

The area overhead of encryption/decryption modules are quite significant, as compared to other components in the design. The relationship between area overhead and cycles is shown in the below plot. However, it is worth noting that the baseline core adopted has only 0.5KB L1 Cache on each core. Since in any modern processor, the majority of area is occupied by L2 and L3 cache, and the cache size is usually in MB level, the area overhead will not be very significant as compared to other modules in the design.

Alternatively, we can use a sequential and blocking encryption/decryption module to do the same work. This sequential module can better utilize existing hardware and significantly reduce the area overhead. This comes at the cost of reduced throughput. For programs with low bus throughput, this can be a good and cheap alternative.

The base core we used has a clock cycle of 6ns. This table also shows the delays of encryption/decryption modules comparing with different computing cycles, in a precision of 0.5ns. From the above table it can be seen that, if the

encryption/decryption module takes more than 2 cycles, these added modules will not affect the baseline frequency of the core. For two-cycle encryption/decryption scenario, a mean CPI of 1.14 (normalized to baseline machine) was obtained for the EECS 470 test-cases. This value further drops to 1.07 using a single-cycle implementation.

Fig. 6, 7, and 8 are summaries of latency, area and performance overhead comparing with different number of cycles implemented for encryption/decryption modules.

B. *Experiment Setup & Results*

Our baseline reference is a R10K style Out-of-Order, two core system with snooping bus and MESI cache coherence protocol. A clock period of 6ns is achieved using this system. More detailed configuration can be found in [8].

After analyzing the overhead for different number of cycles, we decided to use modules with 2-cycle delay, as the clock period is not significantly affected by this value. The benchmark suite we used were adopted from EECS470 final project. We ran the same benchmark suite both on our modified machine and baseline model and then compare the performance.

As shown in Fig. 9, among the 21 programs, we have achieved a best case performance overhead of 0.9% and a geometric mean of 13.9%, which we believe is acceptable in most systems considering the security improvement. We will cover security analysis in section V Security Analysis.

Fig. 6. Encryption Module Latency VS. Cycles of Implementation

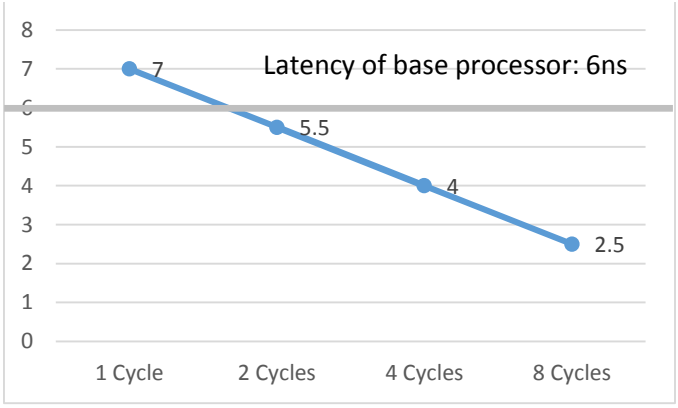


Fig. 7. Area Overhead VS. Cycles of Implementation

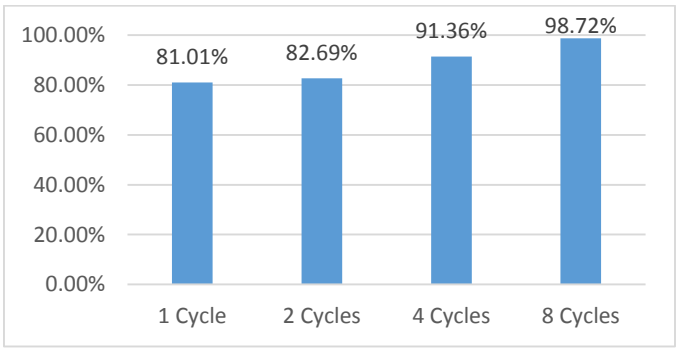
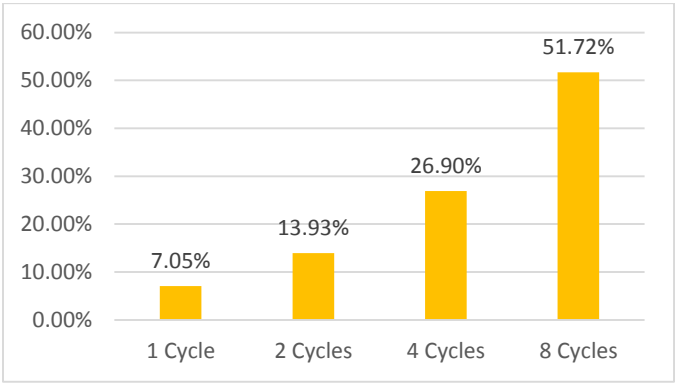


Fig. 8. Performance Overhead VS. Cycles of Implementation



C. *Discussion*

To better understand and explain the performance overhead, we further examine the results obtained from the benchmark suite. The overhead is mainly caused by extra memory access latency in order to encrypt/decrypt the data. A store instruction would not expose any extra latency due to the nature of unblocking memory access and Load-Store-Queue. However, a load request, either issued by I-Cache or D-Cache, has to wait for its data to be successfully fetched. During this time, if no instruction can be executed, the entire pipeline can be stalled and therefore cause a performance penalty. These load requests can be further divided into two groups: 1) load requests from D-Cache, and 2) load requests from I-Cache caused by mispredicted branch instructions. Load requests from I-Cache during normal execution situation would usually not cause noticeable performance degrade since they are usually issued by the prefetcher. To examine the two different troublesome load requests, we introduce D-Cache miss per instruction (DMPI) and branch misprediction per instruction (BMPI) as a quantitative representation for these abstract terms. As a load miss in I-Cache and in D-Cache would cause roughly the same number of stalls, we can simply add them up and get a new metric. We then compare the performance overhead and (BMPI+DMPI) of each program. As shown in Fig. 10, it turns out that the performance overhead and (BMPI+DMPI) have a Pearson product-moment correlation coefficient of 0.804, which means (BMPI+DMPI) itself can be a good indicator of performance overhead.

Fig. 9. Performance Overhead vs Benchmarks

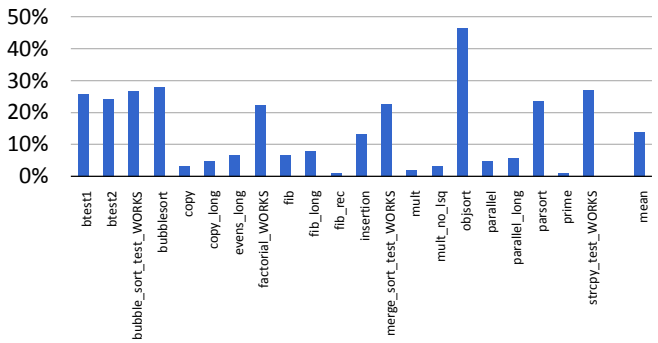
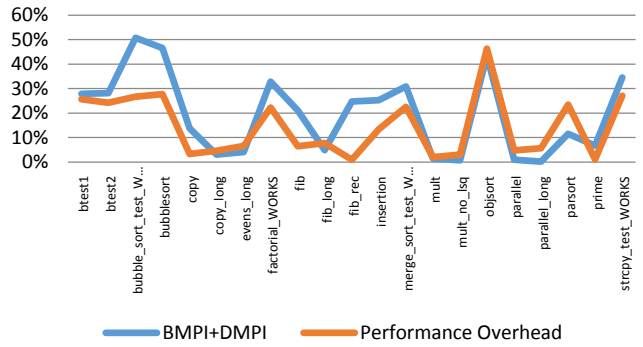


Fig. 10. Correlating BMPI+DMPI to Performance Overhead



As a result, programs with higher branch misprediction rate and higher L1 D-Cache miss rate are more likely to suffer the extra penalty of this new design. However, this result also provides us with a direction to optimize the performance with such encryption design. A better branch predictor will help to reduce the BMPI, and a better L1 D-Cache can help reduce the DMPI. In this way, the performance penalty because of having this encryption is likely to be further covered up.

V. Security Analysis

A. Attacking Using Brute Force

In-order to analyze the secure behavior of our system, various methods (attack the underlying cryptographic algorithm, attack the hardware implementation etc.) are used to try and hack into the system. For all these analysis, the keys are to be kept constant, i.e., they aren't refreshed after a fixed number of cycles. Brute force was one of the methods tried to hack into the secure multi-core system. As we are changing our key at a very frequent rate in the actual implementation, brute force does not make any sense, and is carried out just for analytical purposes, and finding a higher value for the time period of refreshing the keys. Since the key is 128 bits long, it takes maximum 10^{37} cycles for all the combination of keys to be tried out. A point worth noting here is that the 'equivalent keys' weakness[1] has been considered in the above analysis. Assuming each encryption/decryption process takes 2 cycles, and considering that all the encryptions/ decryptions are happening in parallel, it takes max 10^{38} computation cycles for the complete brute-force procedure to finish.

Assuming that the keys are being refreshed after a very relaxed interval of a million cycles, the probability that brute force will be able to attack the system decreases even further. In numbers, the probability that brute force can hack into our system is of the order of 10^{-32}, and grows linearly as the refresh interval is increased. Hence, we can conclude that our proposed technique is resilient to brute force attack.

B. Attacking Using "Impossible Differential Cryptanalysis"

From the above analysis, it is quite clear that using brute force for attacking our system is not a clever idea, owing to the extremely large number of cycles required for the iterations. So, a smarter method is required to actually test our system to its limits. Impossible Differential Cryptanalysis is one such method used to smartly hack into our system. [2] provides a detailed description of the method, and the tools used for analyzing the effect of the mentioned method on our multi-core system. Following the mathematical steps mentioned in [3], we can infer that that the number of cycles required to hack into our system is much lower than required using Brute Force attack method.

According to [3], 64 bits of the 128 bit will have to be determined by exhaustive searching, other than the 64 bits which are generated using Impossible Differential Cryptanalysis in a few hundred cycles, which is the time required for the warm up of the differential module. So, assuming the latency is masked behind the time required for exhaustive search, we can say that maximum 10^{18} cycles are required to hack into this system. This is much less than the values obtained using Brute force, still high enough to allow us to switch our keys at a very relaxed interval. Assuming that the keys are being refreshed after an interval of a million cycles, the probability that brute force will be able to attack the system decreases to the order of 10^{-15},with the value increasing linearly with increase in the refresh interval. This probability is low enough to validate the security of the implemented system.

C. Random Number Generator and Key Refreshing Frequency

In all encryption designs, the quality of keys determine how secure the scheme is. In order to provide numbers with perfect randomness, many True Random Number Generator (TRNG) are proposed based on natural physical phenomenon and are readily available. In our design, a TRNG is required to provide concrete keys to make our design secure. For simplicity, we just assume that there is a perfect TRNGs available in the same system. Our key generator module ("The Cloud") will exploit this hardware and continuously distribute keys to each pair of nodes on the bus after a fixed number of cycles.

It is quite intuitive that the security would be enhanced if the keys are refreshed at a frequent interval. However, this frequency cannot be increased to an infinitely high value. The throughputs of TRNGs can become a bottleneck for the updated frequency.

According to Intel [7], the on-chip TRNG can have a throughput of 70 ~ 200 MB/s. We just take 100 MB/s as the throughput and the clock of TRNG use the same clock as the rest of the two-core system, to ease our analysis here. Three nodes (2 cores, 1 memory) exist in our system so that at least 3 keys should be generated before sending new keys to each node. After simple math, we find that our key generator has to wait at least 80 cycles to start a new round of updating keys.

In reality, the TRNG may use a different clock, and this clock may be slower than the rest of the system, so that it may take longer than we have here to generate enough keys.

VI. Related Work

Design in [4] proposes a similar bus-encryption mechanism using OS scratch-pad space for the encryption/decryption, with a performance slowdown of 40%, much more than the 14% average obtained using this design. Paper [5] describes an

overview of the existing bus encryption techniques, mentioning that an average performance overhead of 10% is considered as reasonable for such designs. Similar results are obtained for our design, albeit with a larger area overhead, making it quite reasonable for comparisons with state of art designs. Also, [6] mentions a novel technique for encrypting the data on the bus, with a performance overhead of 4%, but uses a 32-bit cipher text, thereby casting doubts on the security of the entire system in general.

VII. Conclusions and Future Work

This project has proposed a design dedicated for secure communication on a bus based multi-core system, by introducing Tiny Encryption Algorithm(TEA), a simple symmetric encryption method to encrypt all data communications on the bus. In order to avoid the encryption module from becoming the critical path, the module has been pipelined into 2 stages. Under this implementation, the proposed design has achieved a performance overhead of 13.9% and area overhead of 82.7%. However, since the base system contains only 1KB size of cache in total, and cache with size of MB level usually takes the majority of area consuming in modern CPU design, the relative area overhead will be much more smaller on a modern CPU die.

Due to constraints on time, there are some ideas that we came up with, but could not implement. One is to improve the implementation of the algorithm to have smaller area impact on hardware. Encryption/decryption modules in one core can be combined to save area. Second, we can set a mechanism to encrypt data stored in memory to further secure message. A lookup table needs to set aside in order to re-encrypt data with new keys if the keys have been renewed. Third, an information leakage detection mechanism, along with a dynamic key refresh mechanism can be added to the design. This function can inform the key generator if the current key set have been hijacked, and the direct the key generator to refresh the keys to minimize information leakage.

VIII. Acknoledgement

We thank Prof. Valeria Bertacco for her guidance and advice throughout this project. Besides, we would like to specially thank GSI Doowon Lee for his invaluable help during the entire semester.

References

[1] DJ Wheeler; RM Needham. “TEA, a Tiny Encryption Algorithm”. B. Preenel, editor, Fast Software Encryption, Second International Workshop (LNCS 1008), 1995, pp.363-368

[2] Kelsey, John; Schneier, Bruce; Wagner, David. "Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES" (PDF). Lecture Notes in Computer Science, 1996

[3] Moon, Dukjae; Hwang Kyungdeok; Lee, Wonil; Lee, Sangjin; Lim, Jongin "Impossible Differential Cryptanalysis of reduced Bound TEA and XTEA". Lecture Notes in Computer Science, 2002, Vol 2365, pp 49-60

[4] Xi Chen; Dick, R.P.; Choudhary, A., "Operating System Controlled Processor-Memory Bus Encryption," Design, Automation and Test in Europe, 2008, pp.1154-1159, 10-14

[5] Elbaz, R.; Torres, L.; Sassatelli, G.; Guillemin, P.; Rigaud, J.B., "Hardware engines for bus encryption: a survey of existing techniques," Design, Automation and Test in Europe, 2005, Proceedings, pp.40-45 Vol. 3, 7-11

[6] Elbaz, R.; Torres, L.; Sassatelli, G.; Guillemin, P.; Bardouillet, M.; Martinez, A., "A parallelized way to provide data encryption and integrity checking on a processor-memory bus," in Design Automation Conference, 2006, 43rd ACM/IEEE , pp.506-509

[7] Intel, “Intel Digital Random Number Generator (DRNG) Software Implementation Guide”, 2014, https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf

[8] Guo, Xiaoming; He, Sijia; Wang, Mengtian; Xu, Bangqi, “EECS 470 Final Report,” 2015

[9] Elbaz, R.; Torres, L.; Sassatelli, G.; Guillemin, P.; Rigaud, J.B., "Hardware engines for bus encryption: a survey of existing techniques,", Design, Automation and Test in Europe, 2005, Proceedings , vol., no., pp.40-45 Vol. 3, 7-11