Dynamic Encryption Key Design and Management for Embedded Systems with Insecure External Memory

Mei Hong[1] Hui Guo[1]

[1] University of New South Wales, Australia
{meihong,huig}@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-201206
March 2012

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

To effectively encrypt memory contents of an embedded processor, multiple keys which are dynamically changed are necessary. However, the resources required to store and manage these keys on-chip (so that they are secure) can be extensive. This paper examines novel methods to improve efficiency of encryption(by reducing the amount of re-encryption due to change of key, hence improving the overall encryption speed), and to reduce the required memory resources (by using a special key construction and an implementation scheme). Experiments on a set of applications show that on average, 95% memory and on-chip cost can be saved when compared to the state of art approach.

1 Introduction

Security becomes increasingly important in embedded systems. One driving force is the ubiquitous e-services provided by portable and networked devices that hold and process sensitive and private information. Encryption is a basic technique to protect the confidentiality of the data. There have been many encryption algorithms and security designs. No matter how different they are, all make use, in some form, of encryption keys. The encryption key is extremely critical to the efficacy of the security design. Once the key is broken, the door to the system is wide open to attackers.

Encryption keys can be static and dynamic. Dynamic keys provide a much stronger security protection than static keys, and have drawn increasing attention for memory encryption [21], [24] [23] [8] [9]. However, the generation and management of dynamic keys come at a hefty cost, in terms of resource consumption, which is particularly onerous on resource restricted embedded systems.

Some existing approaches reduce such resource demands by allowing the dynamic key to be reused after some period of time (hence reducing the level of security), or using a small number of keys for a large data set (which introduces re-encryption).

In this paper, we propose a dynamic key design for embedded systems that have a secure on-chip processor and an insecure off-chip memory. **We aim to protect the data secrecy from the physical attack (for example, spoofing), on the off-chip read/write memory and its buses**.

With our approach, the key for the memory content encryption is unique to each small data block and the design does not share keys among the blocks, thus making it both highly secure and re-encryption free.

We further propose a low cost hardware implementation for the dynamic key such that the on-chip resources, especially the memory consumption, can be greatly reduced.

Our main contributions are:

- We present an improved dynamic key design that not only offers a high security but also makes memory encryption more efficient;
- We propose a novel hardware design to reduce memory consumption required for the dynamic key; and
- The low resource overhead of our design enables effective implementation of the dynamic key in embedded systems.

The paper is organized as follows. Section 2 reviews the related work on memory encryption and encryption key designs. The construction of dynamic key for the memory encryption is explained in Section 3 and the hardware implementation is given in Section 4. Section 5 presents the design evaluation platform and experiment results. The paper is concluded in Section 6.

2 Related Work

Use of encryption primitives has been a common technique in security designs, and the encryption key is crucial to the design. It is desired that the key be unique, random, and secure in distribution and storage.

One-Time-Pad (OTP)[17] was an early encryption key design which offers a high security for simple encryption operations [20]. But the key can be arbitrarily long, hence the initial OTP design approach is not feasible to large majority of applications.

Most modern designs use sophisticated encryption operations (aka algorithms) and fixed-length keys to ensure the security in a way that the secrecy cannot be uncovered within the computation ability of the computing devices [18][4][5].

For those crypto-designs, the key is often made of, or is simply just, a random number to prevent key disclosed from mathematical deduction [7]. A random number can be generated by a True Random Number Generator (TRNG) [10] [6] [13], which is often clumsy and only suitable to large in-house systems. Alternatively, pseudo-random numbers generated through a mixing function are often used [16].

In [16], the authors proposed to generate a dynamic key by using an encryption function (as a strong mixing function) that takes the input from a counter. The counter is incremented each time a key is produced. Different counter values ensure the uniqueness of the key generated.

The key generation mechanisms introduced above can also be used for memory encryption. For memory encryption, there are two common methods: *direct encryption* and *indirect encryption*. In direct encryption, the data block is encrypted with a static key. Approaches proposed in [14], [11] fall this category. The direct encryption presents a moderate security protection due to the use of static key. In addition, with the direct encryption method, the system performance will be degraded because of the significant delay incurred by the decryption, where complicated cryptographic algorithms (e.g. DES, AES) are often used for high security.

Instead of performing such a time-consuming (or heavy) encryption/decryption directly on the memory data, the indirect encryptions apply the cryptographic algorithm to a seed to dynamically produce a key. Data is then light-encrypted or decrypted via a simple bitwise XOR operation with the key [21], [19], [24]. The indirect encryption allows for the heavy decryption to be performed in parallel with the memory access, hence the long decryption delay can be hidden and has little impact on the overall system performance.

To ensure the key uniqueness for memory encryption, the input seed of the encryption function must be unique. In [15], per-block counters are used and the unique seed is formed as a concatenation of the data block's memory address and its counter. The address ensures that different locations are not encrypted with a same key. The counter for a memory block is incremented on each write-back to ensure that the key is unique for each write-back to the same address.

However, the memory access frequencies are different from memory location to memory location. Some memory blocks with a high write operation frequency will have their counters quickly overflowed. In [21], when overflowed, the counter is allowed to wrap up and be reused. This counter wrap-up will cause generating of repeated keys, hence degrading the system security.

Yang et al. proposed [24] to use large counters to reduce the counter overflow frequency. But large counters require more storage space if their values need to be saved, which is very costly and sometimes impossible for on-chip memory. So they store the counter values in the off-chip main memory, and use a cache-liked on-chip counter table to cache recently used counter values. The counter values in the main memory are encrypted. This design, however, incurs a significant performance overhead due to the memory access and decryption delay for the counter values missing in the on-chip counter table. To tackle the similar problem, Yan et al. [23] presented a split counter design where a minor counter is used for each data block and a major counter for a large data page.

Elbaz et al. in [8] proposed to manage counter values in a binary tree structure, where each tree node is a counter value which is hashed by using a hash function. The hashed counter value can, therefore, be stored in insecure memory without security risk. However, the hashing operations take considerable computational time, hence degrading the system performance.

In [9], the authors proposed to only encrypt partial of memory contents in order to reduce the total counter storage size. But this saving is at the cost of security.

In this paper, we propose a dynamic key design and an implementation strategy that incurs a small memory overhead and hence allows for the on-chip key generation, store and management.

Our design is similar to the approach proposed in [23] in that both use a hierarchy design and aim to reduce design overhead for memory encryption. But in their design, a major counter is used as part of the dynamic key and is shared by a large set of blocks. Therefore, the re-encryption for the large data set will be incurred when the counter is overflowed; while we use a different key structure and design with which re-encryption problem is avoided. Moreover, our design is memory efficient, which allows for the on-chip implementation of the security design.

3 Dynamic Key Design

Our dynamic key design targets an embedded system with an insecure memory.

For memory encryption, there are two common methods: direct encryption and indirect encryption, as explained in Section 2. Our design can be used in the both methods. Figure 3.1 shows the general view of dynamic key generation logic, where the encryption can be used for further security enhancement for a direct encryption, or for producing the dynamic key to the indirect encryption [1]. In this paper, we focus on the first level design and the related design approach is elaborated in the following sections.

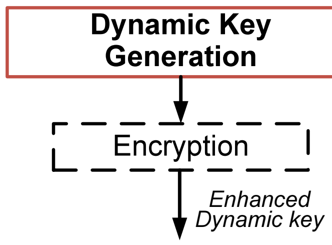


Figure 3.1: Two-Level Structure of Dynamic Key Generation

In traditional dynamic key designs, the key changes from time to time. We regard this dynamic feature as temporal oriented, and the key is referred to as **temporal key**.

For a temporal key, the key is valid for a short period; during the period, encryption and decryption use the same key. This temporal dynamic key design is suitable to communication-like applications, where encryption and decryption are performed in an immediate sequence. For our target system, the encrypted data that is stored in the memory may be accessed and decrypted in a much later time, for example, later than the time when the temporal key is expired. Upon expiry of the old key, all data in the memory need to be re-encrypted with a new key, which is very costly. If each key is, however, associated with a small part of memory (namely, the key is spatial oriented),

[1]With the indirect encryption, the value generated from the first level can be regarded as a nonce, but it is still a key in a general sense.

the re-encryption, due to the key expiry, only needs to be applied to the related small memory section.

Therefore, our dynamic key includes both temporal and spatial features. It differs with memory locations and the key value associated with a memory location lasts very shortly - only for a single memory write. Each time, a new data is written to the memory, a separate (unique) key is used.

The key is constructed with three fields: 1) spatial ID, associated with the memory location of the data encrypted, 2) random value, for high unpredictability of the key, and 3) temporal control, to determine the lifetime of the random value for the memory location.

Since data transfers between memory and the processor are on a cache line basis, we associate one key with a memory section that corresponds to a cache line, and we simply call such a memory section **a cache line**. We partition the memory space into cache lines and encryption is only performed when a cache line is transferred to memory. This memory partition in the key design avoids extra memory accesses and computations for re-encryption when a key is expired, which would be necessary if the key was assigned to a memory section with more than one cache line. Therefore, the spatial ID field in the key takes the address of a cache line.

Like any other key designs, the random number plays an important role in keeping key secret and unpredictable. The larger the random number, the longer and harder for attackers to uncover the key. Change of the random number in the key will make an attack in vain if only the old random value was revealed.

We use the third field in the key to control the life time of the random number associated with a memory cache line. For the lifetime control, instead of using a real time measurement system for the expiry time, we employ a counter to time the age of the random number for a key; the counter is incremented for each memory access to the cache line; When the counter reaches to its maximal value, the random number for the key is expired and a new random number will be used for the memory cache line.

An example of key values for a cache line is given in Figure 3.2. The cache line is associated with a memory location at $0xF0000$. When it is first accessed, a random number $r1$ is created, this random number stays unchanged for subsequent accesses to the same cache line until the counter value reaches its maximum value, MAX. The key value for the same memory location varies from one access to another, being differentiated by the combination of the random number and the counter value, as listed in the last column in the table, where $||$ represents the concatenation operation.

Memory Access Sequence	***Cache line address***	***Random number***	***Counter value***	***LAS Value***
1	**0xF0000**	**r1**	**0**	**0xF0000||r1||0**
2	**0xF0000**	**r1**	**1**	**0xF0000||r1||1**
3	**0xF0000**	**r1**	**2**	**0xF0000||r1||2**
⋮	⋮	⋮	⋮	⋮
i	**0xF0000**	**r1**	**MAX**	**0xF0000||r1||MAX**
i+1	**0xF0000**	**r2**	**0**	**0xF0000||r2||0**
i+2	**0xF0000**	**r2**	**1**	**0xF0000||r2||1**

Figure 3.2: Example of Dynamic Key Values

As can be seen from the example, the number of random values is exponentially reduced with the counter size.

4 Hardware Implementation for Key Generation and Management

For an application with code and data being stored in the memory, the number of encryption keys used can be very large, and hence a large portion of on-chip memory can be consumed. To save the on-chip memory, we store only the random number and counter value for a key. In addition, we allow a random number to be shared among different memory cache lines so that only a small group of random numbers occupy memory space at one time for an application execution. The design is explained below.

4.1 Design Overview

The logic design for the key generation and management is shown in Figure 4.1.

It contains five components: a counter, a counter table, a random number table, a random number generator, and a control logic for random number update. We use one counter to calculate the counter value for each cache line and the counter value is stored in the counter table, which is indexed by the memory address of the cache line; the counter table also provides the link (I_i) to the random number table for each cache line. For a cache line, when the counter reaches to the maximum value, the output carry of the counter enables the random number update.

The random number table contains all random numbers currently used by the cache lines for the application. Initially, there is only one random number that is shared by all cache lines. The number of cache lines that are concurrently using a same random number, is recorded in the last column of the table. A next random number in the table will be used when the random number of a cache line has expired (namely, the counter value exceeds the maximal value, MAX). When all random numbers in the table have been used up by a cache line, a new random number will be generated and saved in the table. If a random number r_i, on the other hand, has been used by all cache lines (namely, u_i is reduced to 0), the related table entry is released for future storage of new values. The entries occupied by the random numbers form a chain and the newly generated random number is normally saved in the top of the chain, pointed by top. Assume there are k entries in the table, the pointer top will loop around in the range from 0 to $k-1$. As can be seen, the table is structured as a FIFO (First In First Out), where the entry that is first phased in will be first phased out.

In case that a memory cache line is frequently updated, a row of new random numbers for the cache line may be used and each newly generated random number will occupy one table entry, which would result in a large portion of the table being consumed by a single cache line. To alleviate the problem, we use the same entry to store the next generated random number for the same cache line as long as the current random number in the entry is not used by other cache lines. The control logic of the random update is given in Algorithm 1, where RN_{index}, RN, and U are arrays for the table columns used in Figure 4.1.

Algorithm 1 Random Number Table Update Control

```
/* For a cache line $i$, when its counter value reaches the maximum value, $MAX$, ... */

/* Determine where to store a new random number: */
/* IF the cache line is the only user of the current random number entry, */
**if** $U(RN_{index}(i)) = 1$ **then**
    /* reuse the entry for the new random number; */
    /* generate a new random number. */
    r=new_random_number();
    $RN_{RN_{index}(i)} = r$;
    /*else if the random number is the most latest one (all random numbers in the
    table have been used by the cache line)*/
**else if** $RN_{index}(i) = top - 1$ **then**
    /* if there is an empty entry available, */
    **if** top is not NULL **then**
        /* put the new random number in the entry pointed by the $top$, */
        /* generate a new random number. */
        r=new_random_number();
        $RN(top) = r$;
        /* and $top$ points to the next available entry. */
        $top = mod(top + 1)$;
        /* if the next entry is not available, top is set to Null. */
        **if** $U(top)! = 0$ **then**
            top = NULL;
        **end if**
        /* else if top is NULL, */
    **else**
        /* The RN table is full, the execution cannot continue. */
        stop the execution;
    **end if**
    /* Else, the random number is not the latest one, the next random number in the
    table can be used. */
**else**
    $U(RN_{index}(i)) - -$;
    $RN_{index}(i) = mod(RN_{index}(i) + 1)$;
    $U(RN_{index}(i)) + +$;
**end if**
```

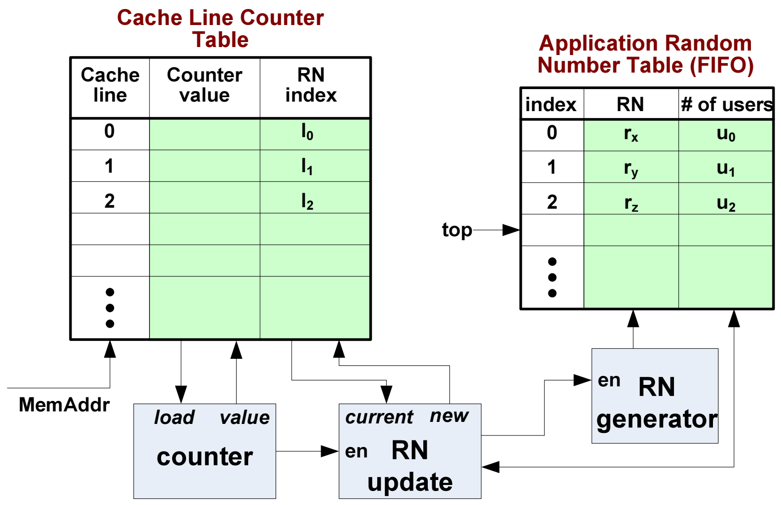


Figure 4.1: Design for Key Generation and Management

4.2 Counter Table Logic Design

As can be seen from the design shown in Figure 4.1, the counter table increases with the size of application. Assume an application has a code of $1MB$ and the cache line size is $32Byte$, then the related counter table should have $1MB/32B = 32K$ entries. And the table can go even huge if the code is spread over a large memory space since the memory address is used to index the counter table. For example, if the code is not stored in a consecutive section in the memory, instead spanning over a space with the address in a range of $32MB$, then the counter table will have over million entries.

We want to reduce the table for a given application - based on the application footprint in the memory, not the address space it covers - so that the entries related to the cache lines that is not used by the application are removed. However, such a design may require the cache line addresses be stored in the table and lead to a full table scan search for a cache line counter value, which is very costly in terms of performance. Here, we propose an encoding approach, where the real memory address trace of an application is encoded with a smaller number of bits that is used as the address to the counter table. The table address is then decoded to index each entry in the table, as shown in Figure 4.2(a), where the number of bits of the encoder's output, M, is smaller or much smaller than that of input memory address bits, N.

Since the hardware complexity (hence the cost) of the address decoder goes up exponentially with the input size, we use two dimensional array for the counter table entries.

Given the cache line address of N bits, we partition the cache line memory address bits into two groups: one group with n high frequent bits (i.e. bits that change frequently in the trace) that form the address for the table column, and another group with m low frequent bits *to be encoded* (with a smaller number of bits, k) to form an address for the table row, as illustrated in Figure 4.2(b).

The approach for partitioning the cache line address bits is summarized in Algorithm 2, with some explanation given below.

Given an application execution trace T, its N-bit address trace form N columns, which can be classified into three categories: the redundant bit columns, the fully rep-

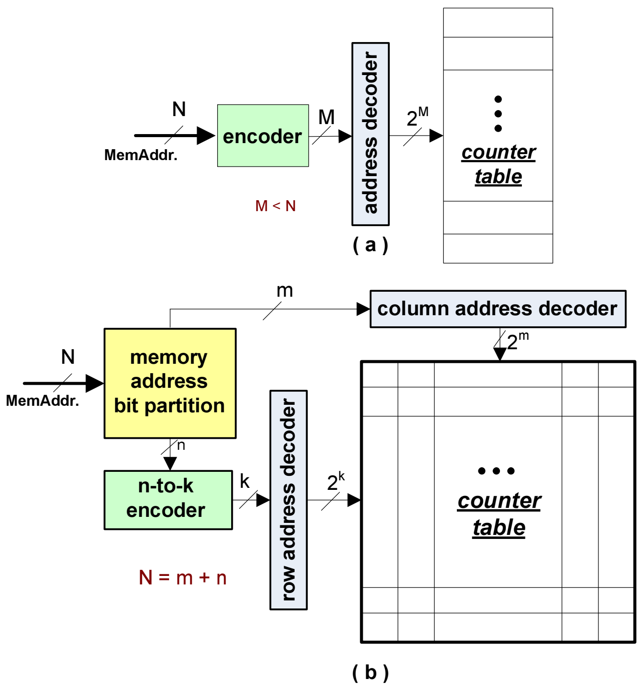


Figure 4.2: Counter Table Design (a) one-dimension design (b) two-dimension design

resentative columns, and the columns not belonging to the first two categories.

A **redundant bit column** in the address trace satisfies one of the two conditions: a) the column has a fixed constant value, and b) the column is a repetition of another column.

A set of columns are **fully representative, FR**, if they cover all possible binary code values for the number of bits given by the column set. For example, the 2-bit code has four possible values: $\{00, 01, 10, 11\}$. There are 2^k different values (from 0 to $2^k - 1$) for the k-bit code.

Figure 4.3 shows an example of the design, with the application address trace given in Figure 4.3(a) for an 8-bit memory space.

In the example, the size of the application is 11 cache lines, hence a counter table of 11 entries would be ideal. We partition the address bits into two groups $\{b_1, b_0\}$ and $\{b_7 - b_2\}$. $\{b_1, b_0\}$ are fully representative columns (containing all four 2-bit values) and they form the first group in the partition. The second group has 4 individual values, which can be encoded with 2 bits $\{r_1, r_0\}$, as given in Figure 4.3(b). Therefore, we use $\{b_1, b_0\}$ as the column address and $\{r_1, r_0\}$ as the row address (see Figure 4.3(c)). The counter table has therefore $2^4 = 16$ entries, which would have been $2^6 = 64$ entries if the cache line address was used to index the table.

It must be pointed out that the same application with different input data may result in a different execution trace, hence different memory access sequence; but that difference is often reflected only to the order of the low fully representative address bits since the application memory address trace often changes locally, thus having little impact on the address bit partition for the counter table design.

Algorithm 2 The Row and Column Address Design for Counter Table

```
/* For an application with execution address trace $T$; */

/* delete the redundant columns from $T$;*/
T' = redundant_bit_deleteion(T);
/* Get the bits of fully representative columns from $T'$;*/
B = FR_bit(T');
/* Assign B as the counter table column address.*/
column_address = B;
/* Encoding the rest of columns, $T''$, with a minimal number of bits; */
E = encoding($T''$);
/* Get the output bits of the encoding */;
R = bit(E);
/* Assign R as the counter table row address. */
row_address = R;
```

4.3 Minimum Overall Table Size

With the above key construction and the design for key generation and management, we can see that the random number table size is adversely related to the counter size. The smaller the counter size, the larger the random number, and hence the wider and longer the random number table.

Assume the encryption key is K bits long, the cache line memory address is N bits, the number of entries in the counter table is L_{CT} (i.e., the length of the table), and the number of entries in the random number table is L_{RN}, and that the counter size and the random number size are S_{CT} and S_{RN}, respectively. According to the table design shown in Figure 4.1, the overall memory consumption, S, for the key generation and management can be calculated

$$S = L_{CT}(S_{CT} + log_2 L_{RN}) + L_{RN}(S_{RN} + log_2 L_{CT}). \tag{4.1}$$

$$S_{RN} = K - N - S_{CT}. \tag{4.2}$$

Based on our experiments, the maximal chain length is inversely related to the counter size, and can be approximated as

$$L_{RN} = e^{(a-b*S_{CT})}, \tag{4.3}$$

where a and b are positive numbers. (More discussion will follow in Section 5.2.)

Then Formula 4.1 can be approximated as

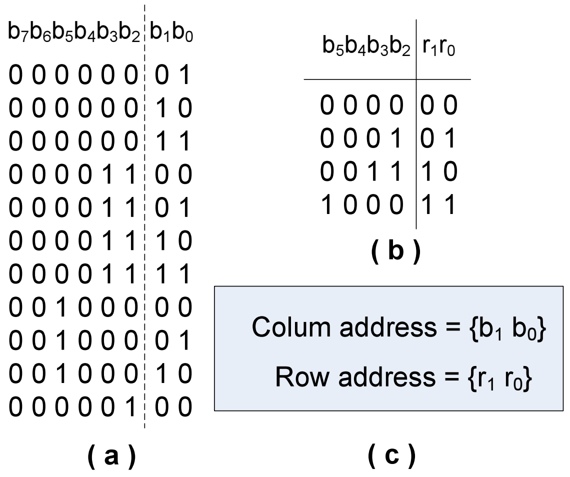
$$\begin{aligned} S &= L_{CT}(S_{CT} + log_2 e^{(a-b*S_{CT})}) + e^{(a-b*S_{CT})} * (K - N - S_{CT} + log_2 L_{CT}) \\ &= L_{CT} * \frac{a}{ln2} + L_{CT}(1 - \frac{b}{ln2})S_{CT} + e^{(a-b*S_{CT})} * (K - N - S_{CT} + log_2 L_{CT}) \end{aligned}$$

The differential of S over the differential S_{CT} is

$$\frac{dS}{dS_{CT}} = L_{CT} * (1 - \frac{b}{ln2}) - (b * (K - N - S_{CT} + log_2 L_{CT}) + 1) * e^{(a-b*S_{CT})}. \tag{4.4}$$

The second order of the differentiation is

$$\frac{d^2S}{dS_{CT}^2} = b * e^{(a-b*S_{CT})} + (b * (K - N - S_{CT} + log_2 L_{CT}) + 1) * e^{(a-b*S_{CT})}. \tag{4.5}$$



$b_7b_6b_5b_4b_3b_2$	b_1b_0
0 0 0 0 0 0	0 1
0 0 0 0 0 0	1 0
0 0 0 0 0 0	1 1
0 0 0 0 1 1	0 0
0 0 0 0 1 1	0 1
0 0 0 0 1 1	1 0
0 0 0 0 1 1	1 1
0 0 1 0 0 0	0 0
0 0 1 0 0 0	0 1
0 0 1 0 0 0	1 0
0 0 0 0 0 1	0 0

(a)

$b_5b_4b_3b_2$	r_1r_0
0 0 0 0	0 0
0 0 0 1	0 1
0 0 1 1	1 0
1 0 0 0	1 1

(b)

Colum address = $\{b_1\ b_0\}$

Row address = $\{r_1\ r_0\}$

(c)

Figure 4.3: Counter Table Design Example (a) execution address trace (b) encoding (c) column/row address bits for counter table

Based on Formula 4.5, we can see $\frac{d^2S}{dS_{CT}^2} > 0$. Therefore, analytically there is a minimum value for S.

4.4 Design Enhancement with 2-Level Counter Table

In the above design, a fixed counter size is used for a given application. Since the memory access frequency varies with memory locations, the actual counter size required by different memory cache lines may be dramatically different. A large counter required by some locations may not be necessary for other cache lines.

Figure 4.4 shows a case of the counter usage frequency of the different counter size, from the experiment on an application.

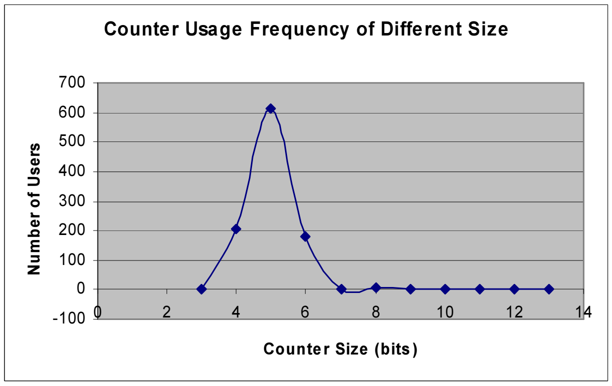


Figure 4.4: Counter Usage Frequency of Varied Size

As can be seen from the distribution, the counter of 6 bits can cover most cases for cache line accesses.

Therefore, we use a 2-level counter table design. The first level counter table targets the common case for most cache lines and the second level is for the special case with an extreme memory access behavior.

5 Design Evaluation

To evaluation our design approach, we have built a design platform and carried out some simulations.

5.1 Design Platform and Experimental Setup

We implemented our dynamic key design approach to an existing memory data protection design, proposed in [3]).

The design uses a same system architecture as the one we targeted in this paper: the secure processor chip and insecure off-chip memory. The on-chip contains a processor, instruction and data caches, and a component for off-chip memory data protection. The component provides functions for encryption and tag generation for a cache line written to the memory, and functions for decryption and tag verification when a cache line is fetched from the memory. Here we apply our design for the dynamic key generation and management required by the data protection component.

To streamline the design and evaluation, we build a design platform based on the Xtensa design tool [22], which allows for different cache/memory configurations and varied architectural settings. The design platform is shown in Figure 5.1.

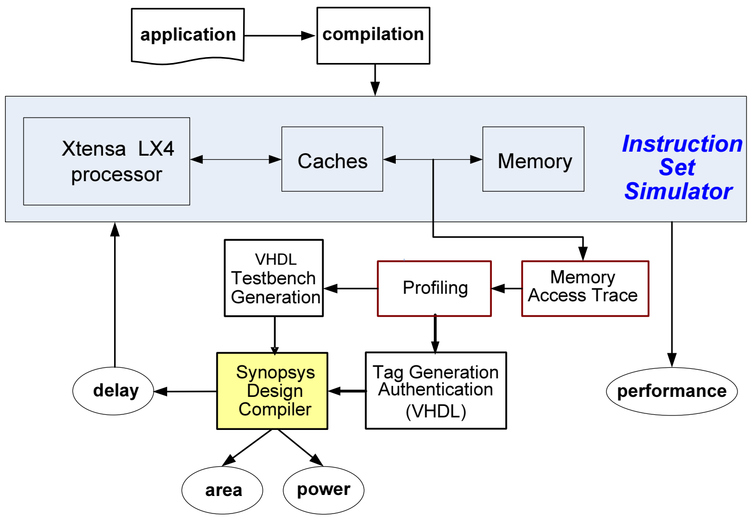


Figure 5.1: Design Platform

In our experiment, the architectural parameters of Xtensa processor core are configured to model a typical embedded processor. The detail configurations are shown in Table 5.1.

Based on the settings, applications are compiled by Xtensa XCC compiler to generate binary executables, which are run on the cycle-accurate Instruction Set Simulator ISS for an execution trace. Through the trace analysis, the required information for

Parameter	Config.
Core Speed	694 MHz
I-Cache	2KB
D-Cache	2KB
System RAM	1MB
System ROM	4MB
Instruction Bus Width	4B
Data R/W Bus Width	4B

Table 5.1: Xtensa Base Processor Configuration

memory accesses is extracted to determine the size of the counter, counter table, and random number table for the dynamic key.

We implement the logic design for the dynamic key using the Hardware Description Language, VHDL. The extracted cache/memory access trace is used to generate the VHDL testbench. The functionality of the design is verified with ModleSim simulator [1]. After the functionality is verified, the design is synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler [2] with 65*nm* standard cell library, to evaluate the area, delay and power cost of the design.

5.2 Experimental Results

Eight applications are selected from the embedded system benchmark suite MiBench [12]. We investigate the design for each application, and the evaluations are performed on two design schemes: design with 1-level counter table, and the design with 2-level counter table.

The experiment results for the designs with 1-level counter are given in Table 5.2. With different counter sizes (*Sct*), as listed in the first column of the table, the length of the random number table (*Lrn*) and the overall design memory consumption (*S*) are given in the rest columns, for each application (shown in the first row). The counter size is measured in bits, the length of random number table in entries, and the memory consumption in bytes.

From Table 5.2, we can see that the length of the random number table is inversely related to the counter size, as expected.

And we use the *Sct* and *Lrn* data in the table to derive a regression formula between the counter size and the random number table size. Six typical regression functions, given in Table 5.3, have been considered. Table 5.4 presents the RMSE (Root Mean Squared Error) value from each of the regressions. The last row of the table gives the average RMSE value.

As can be seen from Table 5.4, on average, the exponential regression has a minimal error as compared to other regressions. Therefore, we have the following approximation:

$$Lrn = e^{(a-b*Sct)},$$

as has been used in Section 4.3.

Also from Table 5.2, we can see that for some applications, when the counter size increases in the range of 1-16 bits, the memory consumption (*S*) changes convexly with the counter size, and there is a minimum memory cost, as highlighted in bold. For other applications (*adpcm*, *jpeg*, *susan*, and *rijndael*), the convexity is not so obvious. This

	adpcm		dijkstra		jpeg		qsort		rijndael		sha		stringsearch		susan	
Sct	Lrn	S	Lrn	S	Lrn	S	Lrn	S	Lrn	S	Lrn	S	Lrn	S	Lrn	S
1	19	583	195	4787	5	491	49	1490	44	773	40	925	29	752	18	1562
2	19	644	131	4032	3	498	35	1368	42	772	29	812	18	627	15	1709
3	19	706	87	3548	2	538	28	1360	40	771	24	791	10	532	13	1884
4	19	768	58	3270	1	525	20	**1320**	36	743	17	732	6	497	12	2096
5	19	829	36	3063	1	653	16	1354	32	716	11	679	3	459	11	2306
6	14	801	25	**3040**	1	781	14	1430	27	675	8	**675**	2	473	10	2513
7	9	761	18	3085	1	908	11	1475	22	633	6	687	1	**460**	7	2598
8	6	750	14	3196	1	1036	6	1428	19	620	5	721	1	524	7	2853
9	5	785	14	3450	1	1164	4	1457	14	576	3	713	1	588	4	2865
10	4	816	13	3665	1	1292	3	1519	10	544	2	728	1	652	3	3002
11	3	841	8	3680	1	1420	3	1647	9	558	2	792	1	716	2	3096
12	3	904	4	3632	1	1548	2	1688	7	554	2	855	1	780	1	3084
13	2	919	3	3769	1	1676	1	1676	4	526	2	919	1	844	1	3340
14	1	907	2	3863	1	1804	1	1804	3	532	2	983	1	907	1	3596
15	1	971	1	3852	1	1931	1	1931	2	534	2	1047	1	971	1	3852
16	1	1035	1	4107	1	2059	1	2059	1	523	2	1110	1	1035	1	4107

Table 5.2: Design with 1-level Counter Table and Impact of Counter Size

name	formula
linear	$y = a - b * x$
exponential	$y = e^{(a-b*x)}$
quadratic	$y = (a - b * x)^2$
reciprocol	$y = 1/(-a + b * x)$
logorithmic	$y = a - b * ln(x)$
power	$y = e^{(a-b*ln(x))}$

Table 5.3: Different Regressions

may be explained by the two extreme situations: 1) the memory access is low (such as *susan*) and the counter table is dominant in the memory cost (S), the S value increases with the counter size; 2) the memory access is very extensive (such as $rijndael$), a long chain of random numbers need to be stored; namely, the random number table is dominant. Increasing the counter size will bring the random table size down, hence reducing the total memory cost.

Table 5.5 shows the results of the design with a 2-level counter table for each application, where $L1_{Sct}$ and $L1_{ct}$ are the counter size and the counter table length for the first-level counter table, respectively, and $L2_{Sct}$, $L2_{ct}$, for the second level counter table; Lrn is the random number table length, and Srn the random number size. The design with a customized 1-level counter table is also presented in the table. The overall memory costs for the two designs are given in columns 8 and 10, respectively. It must be noted that for applications $jpeg$, *sha* and *stringsearch*, since their counter size distribution is very focused, there is no need for a secondary counter table.

For comparison, we also estimate the memory consumption for the design proposed by [23]. In our estimation, the six bits for the minor counter size is adopted and the major counter size is 64 bits for 32 byte cache block. The total storage size from the two types of tables is calculated based the application code size and data size in the memory space.

The memory reduction rates of our two designs over the existing design [23] are given in columns 9 and 11 (Table 5.5), respectively. As can be seen from the table, on

	linear	exp.	quadr.	recipr.	log.	power
adpcm	2.45	4.07	2.19	16.01	3.26	10.79
dijkstra	31.70	6.58	24.80	70.02	15.05	90.09
jpeg	0.83	0.84	0.81	0.92	0.57	0.48
qsort	6.51	1.03	4.27	21.81	2.09	20.13
rijndael	2.70	8.80	1.86	44.09	4.56	23.23
sha	6.11	3.98	4.77	21.92	2.65	8.22
stringsearch	5.41	5.46	5.02	6.57	3.22	1.95
susan	1.52	1.89	0.78	39.77	1.33	7.39
average	7.15	**4.08**	5.56	27.64	4.09	20.29

Table 5.4: RMSE of Different Regressions of Random Table Size over Counter Size

average, about 93% memory cost can be reduced when the 1-level counter design is used; with the 2-level counter design, further memory savings can be achieved, with an average saving of 95%. The dramatic reduction of the memory cost makes it possible to implement the security design on chip.

It is worth to note that the design with 2-level counters is especially effective for applications, such as *dijkstra*, where memory accesses are very locally.

	L1_Sct	L1ct	Lrn	L2_Sct	L2ct	Srn	S (1-level)	red. Rate (%)	S (2-level)	red. Rate (%)
adpcm	1	512	6	7	23	88	801	98	348	99
dijkstra	6	1024	8	5	339	85	3040	81	1883	89
jpeg	1	1024	5	-	-	91	491	99	491	99
qsort	6	1024	6	2	22	88	1430	91	1205	93
rijndael	6	256	7	6	35	84	675	98	424	99
sha	6	512	8	-	-	90	675	98	675	98
stringsearch	6	512	2	-	-	90	473	97	473	97
susan	6	2048	7	2	9	88	2513	85	2356	86
AVG	5	864	6	4	86	87	3303	**93**	1118	**95**

Table 5.5: Custom Designs

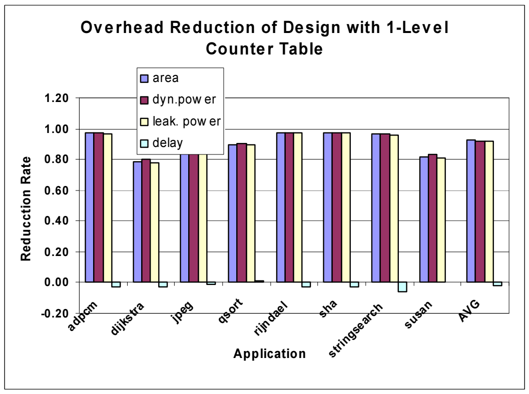
To evaluate the on-chip overhead if the design is implemented on the chip, we model the logic design for each application in VHDL and synthesize each design with Synopsys Design Compiler. The design proposed in [23] is also implemented and synthesized for comparison.

Figure 5.2(a) presents the overhead reduction on area, dynamic power, leakage power and delay for the design with 1-level counter table as compared to the existing design. The average reduction rates are given by the last bar group under the name *AVG*. As can be seen from the experiment results, on average, about 93% of area and 92% of power can be saved, with the delay only degraded about 2%.

Figure 5.2(b) gives the comparison between the designs with 1-level counter and 2-level counter, which shows a further savings (26% area, 25% dynamic power, and 20% leakage power) can be achieved if the two-level counter design is used. For some applications (*rijndael* and *susan*), the delay is even reduced, and on average, almost no extra delay is incurred.

In other words, if implemented on chip, our design only takes a small fraction of hardware resources - less than 5% of the area and power consumption are consumed with a similar delay overhead, as compared to the design proposed in [23].

(a)



(b)

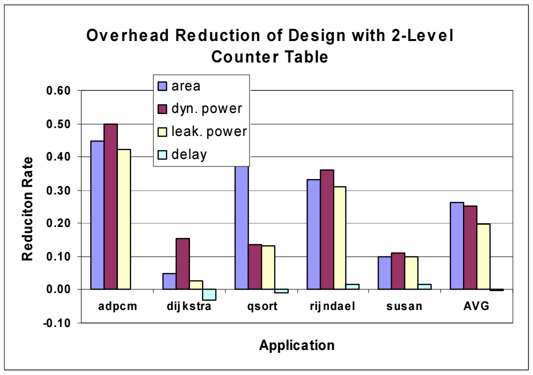


Figure 5.2: On-chip Overhead Reduction

6 Conclusions

High resource overhead is a critical issue in using dynamic key for memory encryption in the embedded systems.

In this paper, we presented a dynamic key design, where the key changes with different memory cache lines and is unique to each memory access, therefore it offers a high level of security. To reduce on-chip hardware resources, we proposed a key storage approach and control algorithms for key generation and management, and we demonstrated a two-dimension and multiple-level implementation for the memory table used for the dynamic key storage.

We implemented our design in a processor design systems so that designs can be systematically generated and the testing can be performed on real applications. We compared our design with a state of the art design approach. Our experiments on a set of applications demonstrated that about 95% memory cost can be saved from our design as compared to the existing approach, which makes it possible for on-chip implementation. And if implemented on chip, our design only consumes less than 5% of the area and power consumption that would be required by the state of the art design.

It must be pointed out that our approach was mainly designed to protect data from physical attacks in the read/write memory. For read only memory data, the key can be designed differently, which will be investigated in the future.

Bibliography

[1] Mentor Graphics Corp. http://www.mentor.com.

[2] Design compiler. Synopsys Inc. (http://www.synopsys.com).

[3] removed for blind review.

[4] FIPS Pub. 197. Specification for the advanced encryption standard (AES). Technical report, National Institute of Standards and Technology, Federal Information Processing Standards, 2001.

[5] FIPS Pub. 46-1. Specification for the data encryption standard (DES). Technical report, National Institute of Standards and Technology, Federal Information Processing Standards, 1981.

[6] D. Davis, R. Ihaka, and P. Fenstermacher. Cryptographic randomness from air turbulence in disk drives. In *International Conference on Advances in Cryptology*, 1994.

[7] D. Eastlake, S. Crocker, and J. Schiller. Request for comment (RFC) 1750: Randomness recommendations for security. Technical report, Network Working Group, MIT, 1994.

[8] R. Elbaz, D. Champagne, R.B. Lee, and L. Torres. Tec-tree: a low-cost, parallelizable tree for efficient defense against memory replay attacks. In *9th International Workshop, Cryptographic Hardware and Embedded Systems*, 2007.

[9] O. Gelbart, E. Leontie, B. Narahari, and R. Simha. A compiler-hardware approach to software protection for embedded systems. *Computers and Electrical Engineering*, pages 315–328, 2009.

[10] D.K. Gifford. Natural random number. Technical report, MIT/LCS/TM-371, 1988.

[11] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing security in the memory management unit. In *Proceedings 25th EUROMICRO Conference. Informatics: Theory and Practice for the New Millennium*, 1999.

[12] M.R. Guthaus and J. S. Ringenberg. Mibench: a free, commercially representiative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[13] B. Jun and P. Kocher. The intel random number generator. Technical report, Intel White paper, 1999.

[14] D. Lie, T. Chandramohan, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *9th Internatinal Conference Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, 2000.

[15] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES modes of operations: CTR-mode encryption. http://csrc.nist.gov/cryptotoolkit/modes/prposedmodes, 2000.

[16] C. Meyer and S. Matyas. *Cryptography: A New Dimension in Computer Data Security*. John Wiley & Sons, 1982.

[17] Frank Miller. *Telegraphic code to insure privacy and secrecy in the transmission of telegrams*. C.M. Cornwell, 1882.

[18] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, 1994.

[19] W. Shi, H.H. Lee, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *32nd International Symposium on Computer Architecture*, 2005.

[20] W. Stalling. *Cryptography and Network Security: Principles and Practices, 4th Edition*. Pearson Prentice Hall, 2006.

[21] G. E. Suh, D. Clarke, B. Gasend, M.van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processor. In *36th International Symposium on Microarchitecture*, 2003.

[22] Tensilica. Xtensa customizable processor. http://www.tensilica.com.

[23] Chenyu Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings. 33rd International Symposium on Computer Architecture*, 2006.

[24] Jun Yang, Lan Gao, and Youtao Zhang. Improving memory encryption performance in secure processors. *IEEE Transactions on Computers*, 54(5):630–640, 2005.