A Cost-Effective Tag Design for Memory Data Authentication in Embedded Systems

Mei Hong[1] Hui Guo[1]

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

Technical Report
UNSW-CSE-TR-201209
March 2012

THE UNIVERSITY OF
NEW SOUTH WALES

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

Abstract

This paper presents a tag design approach for memory data integrity protection. The approach is highly area, energy and memory efficient, very suitable to embedded systems that have stringent resources. Experiments have been performed to compare our approach with the state-of-art designs, which shows the effectiveness of our design.

1 Introduction

Security becomes increasingly critical in embedded systems. Most embedded systems consist of secure processor chips and insecure off-chip memory components. To protect the system, data in the off-chip memory often need to be encrypted and authenticated.

Employing tag to protect data integrity is a common approach, where data is attached with a tag and the tag value is checked each time the data is used; if the tag value is changed, the data is deemed as tampered and invalid.

Unlike the tag design in network communication, where data are immediately authenticated upon arrival at the destination - hence no tag storing is needed, the tag for memory data should be saved since the data will only be authenticated some time later when they are fetched by the processor. Therefore, apart from the performance overhead, the tag design for memory data encounters more challenges: 1) high memory cost for tag storage, and this cost can be very prohibitive because huge number of tags are often used; and 2) increased security risk since the tag can be attacked during its prolonged life in the memory.

In terms of tag design, there are two basic data protection design paradigms. In the first design, as demonstrated in Figure 1.1(a), the tag value is generated *independently* from the data. The data is earmarked by the tag via encryption (denoted as Enc in the figure). Encrypted data and tag, $Enc(D||tag)$, are transmitted and stored in the insecure memory. Due to the diffusion feature of the encryption operation, a change to $Enc(D||tag)$, will very likely alter the tag value (tag') after decryption; hence the change can be detected during authentication. This design requires the original tag be stored on the processor chip.

The second design given in Figure 1.1(b) computes the tag based on the data, namely, the tag is *dependent* on the data value. The authentication compares the tag calculated from the received data with the provided tag value. The tag in this design can be stored either on-chip or off-chip.

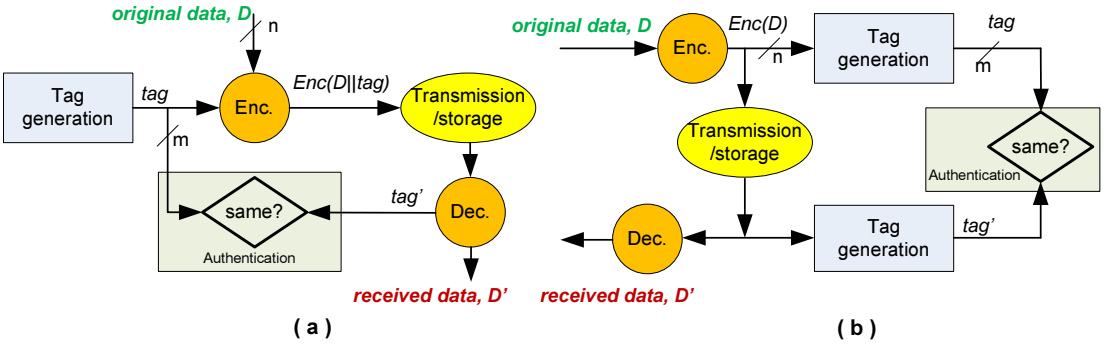

Figure 1.1: Two Memory Data Protection Design Paradigms (a) with Data-Independent Tag (b) with Data-Dependent Tag

Since the first design (data-independent tag) takes the tag into encryption (which increases the amount of encryption operations), and the tag must be saved on-chip (consuming stringent on-chip resources), and the data authentication has to be done after the decryption (namely, wasteful decryption is performed for invalid data), we focus on the second design paradigm - design with a data-dependent tag. The tag is generated based on the encrypted data. The encrypted data and their tags are stored

in the off-chip memory. **We aim to develop a cost effective tag design to counter physical attacks on the insecure off-chip memory and its buses**.

Our main contributions are

- a low overhead design approach for tag generation and data authentication,
- an analytical model for tag size selection and customization, and
- a simulation platform for design evaluation.

The rest of the paper is organized as follows. Section 2 reviews some existing works related to data authentication and tag design. Our tag design approach is presented in Section 3. The experiment results to show the effectiveness of our approach are given in Section 4, and the paper is concluded in Section 5.

2 Related work

Authentication was initially introduced to messages communicated between sender and receiver over the network so that tampering with messages (forgery, unauthorized modifications, reordering, etc.) can be detected. The fundamental idea of the message authentication is to use a checksum (a.k.a message digest, or tag) calculated from the original message by the sender such that any change to the message will lead to a different checksum. The checksum is appended to the message on transmission. At the receiver side, the checksum is re-computed from the received message and is compared with the original checksum that comes with the message. If they are matched, the message is authenticated; otherwise, the message is considered corrupted.

Blum et al. in [1] discussed possible attacks to the memory contents, demonstrating the need of authentication for memory data even if they are in an encrypted format.

XOM (eXecution Only Memory) [2], proposed by Lie et al., is a hardware-based design for authenticating application code from insecure external memory. In this design, applications are encrypted and stored in separate memory sections. Each of these sections has a unique and fixed numerical identifier and the identifier is used to generate a data-dependent tag for the code in the memory section. When an application is executed, the tag is compared with the tag stored in a secure table. If they are same, the execution is allowed to continue, otherwise, an exception will be generated and the execution is disabled. By this way, applications are isolated and protected from each other. This design uses static data-dependent tags, suitable to read-only memory data; it is not effective for protecting dynamic read-write data from replay-attacks.

To protect the dynamic data, Suh et al. [3] proposed a design, called AEGIS(Architectural EnGines for Information Security), for tamper-evident and tamper-resistant processing. The design uses a nonce to generate the tag for dynamic memory data. The nonce is memory access specific, so that replaying old data can be detected from the un-matching tag values. The design requires a large on-chip memory space to store nonces for authentication.

Since on-chip memory is very limited and expensive, to reduce on-chip memory cost, Suh et al. [4] then applied a hash tree proposed by Merkel[5]. The tree leaves are memory content blocks, and the nodes are hash values of their immediate children in the tree. The tree captures the integrity state of all memory contents. In their design, only the root node is stored on-chip that is inaccessible by the adversary; Other nodes are stored off-chip. On a memory read, a path of tree nodes from the leaf (that is associated with the requested memory block) to the root will be re-calculated. If the resulting root matches the reference root stored on-chip, the memory contents are validated; otherwise, the contents are invalidated. The tree is updated from leaf to

root on a memory write operation. Since the node path recalculation/update involves heavy computation and multiple memory accesses, long delays will be incurred. To reduce the delay, Gassend et al. [6] proposed to save tree nodes that have been previously authenticated in a fast on-chip cache. With the node caching, the path nodes re-calculation process will stop as soon as it hits a cached node and the authentication can be completed earlier by just comparing the cached tree node with the newly computed node. Elbaz et al. [7] further improved the hash tree design by constructing a Tamper-Evident Counter tree (TEC-tree) where both authentication and tree update processes are parallelizable. Rather than waiting for lower level nodes' calculation from the memory data, the tree allows for calculation of nodes from counter values at different tree levels simultaneously .

The above designs can be categorized as using a Generic Composition (GC) approach, where encryption on the plain memory data is performed first, followed by the tag generation for data authentication.

Since both encryption and authentication are often computational intensive, the sequential execution of the two processes in GC designs has a significant impact on the system performance. Some authenticated-encryption (AE) algorithms have, therefore, been proposed [8] [9] [10]. AE algorithms usually use the block cipher and mode operations for encryption and authentication. Since the mode operations are parallelizable, the performance overhead can be moderated. Moreover, instead of in two sequential execution steps, AE algorithms mix encryption and authentication in one step, enabling further parallel operations for performance improvement.

In [11], Yan et al. applied an AE algorithm, Galois Counter Mode (GCM) [9], which was initially proposed for message authentication, for general-purpose processor computing systems. With their design, the memory authentication is in parallel with encryption. Nonce values are used in generating authentication tags; A nonce is formed by a non-repeated counter and memory address. Tags are updated when memory contents are modified, to resist the old tag replay attacks. Tags are placed in off-chip memory, and counters are maintained in a counter tree, with only the root counter stored on-chip to save on-chip memory. Each time a nonce is used for memory data authentication, its counter value is retrieved from the counter tree. Maintaining a counter tree is more resource efficient than that for the tag tree, since counters usually have a small size and low computation overhead.

Rogers et al. [10] used a similar tag generation approach as proposed by Yan et al. in [11]. Unlike Yan's approach, where GCM is used, Roger's design applies a Parallel Message Authentication Code (PMAC) algorithm [8] that allows for using a single hardware encryption component for both encryption and authentication, hence it is computing resource-wise and cost effective.

In [12], Elbaz et al. proposed an Added Redundancy Explicit Authentication (AREA), which eliminates tag calculation during authentication. AREA is an AE approach in a sense that both encryption and tag generation is completed in one step. The principle of the AREA scheme is to insert tag as redundancy into the plaintext before encryption and to check it after decryption. Memory addresses and nonces are used to form tags. Because of the diffusion property of encrypt function, the tag and data are mingled in encrypted bit string. Any change to the bit string could cause change to the tag after decryption. This method requires nonces be stored on-chip. In order to save on-chip memory cost, the scheme treats Read Only (RO) memory contents and Read Write (RW) memory contents differently. For RO memory data, which is never changed during runtime, no nonce is needed and only memory address is used in the tag. For the RW data that may change during execution, a nonce and its memory address are used

in the tag.

Our approach is similar to Yan's and Roger's in that all are AE algorithm based, the tag generation is nonce-controlled, and tags are saved off chip. But there are major differences: 1) Both of Yan's and Roger's use an existing algorithm (GCM, PMAC) original developed for message authentication, which is computation intensive and consumes large hardware resources, and 2)None of the two works address customization of tag size, which has a great impact on memory consumption and system security. Since most embedded systems are application specific, we can customize the design to achieve high security while at a low overhead. Specifically, we propose a design approach to customize tag size and tag generation algorithm to achieve an optimal tradeoff between security and on-chip overhead and performance.

3 Tag Design

We target an embedded system that has a secure processor chip and insecure off-chip memory. The processor chip also contains cache and components for encryption/decryption and tag generation/data authentication.

3.1 Design Problem and Approach

Given such a system, for a cache line to be written to the off-chip memory, a tag is first generated on the processor chip, then the tag together with the encrypted data is transferred to and stored in the off-chip memory. When the data is later required and fetched into the processor chip, a tag value of the fetched data is calculated and compared with the tag obtained from the memory. If both values are same, the data is authenticated and can be further decrypted for use. Otherwise, the data should be discarded.

Figure 3.1 illustrates the flow of tag movement in our target system, where the bus and off-chip memory can be under physical attacks.

For a cache line L, its encrypted data $Enc(L)$ and related tag, T, can be tampered in three circumstances: 1) on the bus during transmission to the memory (denoted as **bus attack**), where $Enc(L)$ is replaced with $Enc^{(1)}(L)$ and/or *tag* is replaced with $T^{(1)}$; 2) in the memory (denoted as **in-memory attack**, where the data and tag can be changed to $Enc^{(2)}(L)$ and $T^{(2)}$); and 3) on the bus when the data is fetched from the memory to the processor (with altered $Enc^{(3)}(L)$ and $T^{(3)}$ for the memory data and the related tag).

Based on the illustration, a data alteration can escape authentication if the tag of the fetched memory data ($En^{(3)}(L)$) and the tag value from the memory ($T^{(3)}$) are same, which allows for the following types of attacks:

- Replacement *type A* attack, with random and known values. $En^{(3)}(L)$ is randomly selected or purposely picked, and the tag for $En^{(3)}(L)$ is guessed correctly (namely, $T^{(3)}$=$T^{(4)}$).
- Replacement *type B* attack, with known value and tag pairs. $En^{(3)}(L)$ and $T^{(3)}$ are a valid copy of a different memory location.
- Replay Attack. $En^{(3)}(L)$ and $T^{(3)}$ are a copy of previously observed valid data and tag pair of the same memory location.

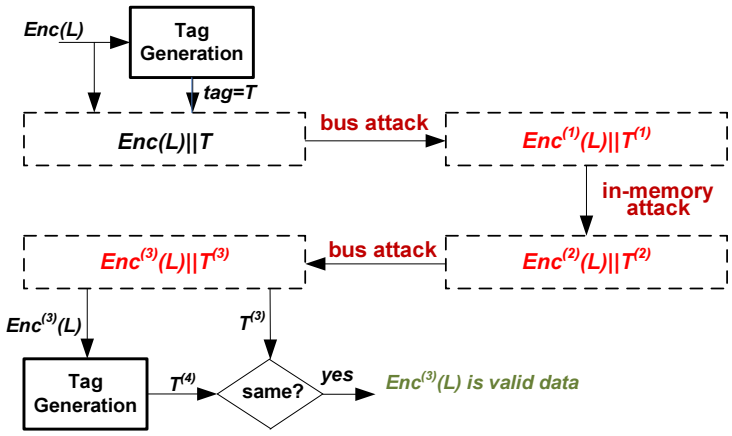

Figure 3.1: Tag Movement and Possible Attack Places in the Target System

For the replacement type A attack, a brutal force approach can be applied to explore a correct tag value. For the replacement type B or replay attack, it can be quite easy since the encrypted cache line and tag are accessible to the adversary.

Therefore, it is desirable that

- The tag is highly sensitive and unique to a change on the data. Namely, tag(data1) $\neq$ tag(data2), if data1 $\neq$ data 2.
- The tag possesses a strong resistance to replacement type B and replay attacks. The tag value will change with the memory location of the data and the time the data was accessed.
- The tag is difficult to guess correctly, and
- The tag design is easy to implement, with low hardware, performance and power overheads, especially we want as low memory consumption as possible.

In our design approach, we

- Apply a nonce to tag generation and the nonce value is memory location and access specific, to counter replay related attacks;
- Use a small yet effective logic for tag generation to reduce the on-chip hardware implementation overhead; and
- Use a as small tag as possible to tag a larger cache line data, to reduce the off-chip memory overhead for tag storage.

The tag design is elaborated below.

3.2 Tag Generation

Given an encrypted cache line of n bits, the tag generation is to convert the n-bit value to a random m bit value; In our design, $m < n$.

Since the cache line data after encryption are usually uniform random [13], we can utilize an *inverse transform* method [14] in our tag design - **generating a uniform random tag from a uniform random encrypted cache line**. It is worth to note that the

function	function decryption	uniform rand?
$X + c$	add with constant	yes
$X + Y$	add with variable	no
cX	multiply with constant	yes
$X * Y$	multiply with variable	no
$X << c$	logic shift with constant bits	no
$X << Y$	logic shift with variable bits	no
$X \ \ RS$ c	rotate shift with constant bits	no
$X \ \ RS \ \ Y$	rotate shift with variable bits	yes
$X \ \ AND \ \ Y$	bitwise logic AND	no
$X \ \ OR \ \ Y$	bitwise logic OR	no
$X \ \ XOR \ \ Y$	biswise logic XOR	yes
$NOT \ \ X$	bit inverse	yes
$Swap(X, i, j, n)$	swap two n-bit sections in X	yes

Table 3.1: Is It Uniform Random Distributed?

uniform random distribution provides a higher level of difficulty in arbitrarily searching a value in a space than other random distributions, hence reenforcing the design to counter tag attacks.

To develop such an inverse transform function, we have investigated a set of operations to see whether they can result in a uniform random given a uniform random input.

Table 3.1 is a summary based on our analytical and experimental investigation (the detail is omitted due to the space limitation). In the table, X, Y, i, j are uniform random variables and c is a constant.

As can be seen from the table, a few operations can convert a uniform random value to another value that is also uniform random distributed. We call such operations, *Uniform Convert (UC)* operation. To ensure the tag value is uniform distributed, we use *UC* operations, *swap*, *bit rotate shift* and *XOR* in generating tags.

Furthermore, to allow the parallel execution (for performance improvement), we use block operations in our tag generation. Since the block-operation based design may invite the slicing attack (an attack that replaces a block in the original data with a known block), we apply a shuffle operation on the input data. The shuffle operation mixes the bits in the original data and makes the slicing attack difficult. Figure 3.2 shows an overview of our design.

It consists of three steps: The encrypted cache line is first shuffled and then evenly divided into multiple blocks. The block size is same as the tag size. Each block is next transformed through a permutation function. The results of the transformed blocks are finally XORed to form the tag. Both the shuffle and permutation steps are controlled by a nonce value, which is random and unique to each memory cache line access.

The security and hardware cost of the shuffle design depend on the level of granularity and the number of shuffle rounds applied. To increase the security, we want the shuffle as thorough as possible, for example, on a bit base and with sufficient shuffle rounds.

A typical design based on play-card shuffle is shown in Algorithm 1, where the granularity of shuffling is in units. A cache line of multiple units is divided into two parts. The unit size can be in a range of 1 bit to a half of the cache line size.

Figure 3.3 shows the hardware cost, execution time, and power consumption of the

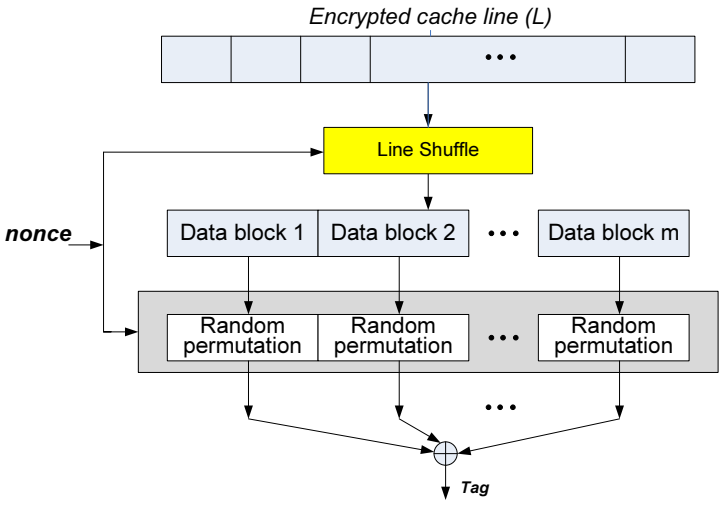

Figure 3.2: Tag Generation Design Overview

design of different granularity for one round shuffle. As can be seen from the plots, the costs of the shuffle design increases exponentially when the unit size in the shuffle decreases.

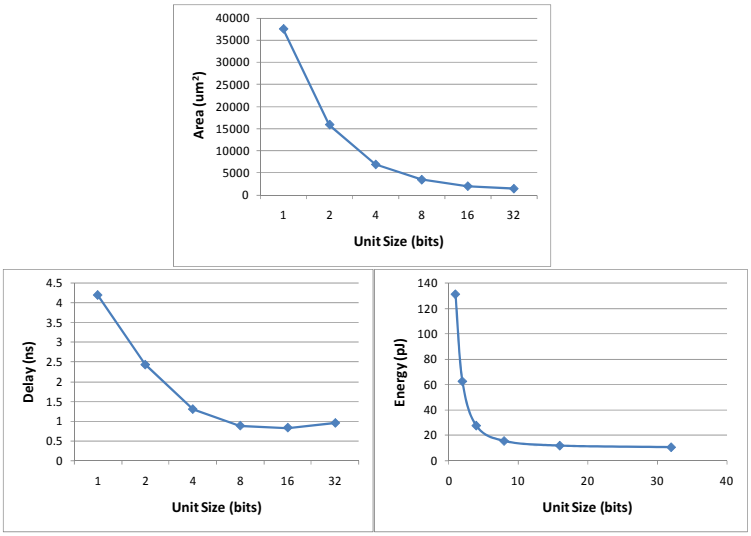

Figure 3.3: Costs of the Example Shuffle Design

Therefore, we use a simplified shuffle design. The cache line is first partitioned into blocks of the same size as the tag; the shuffle is performed between blocks and only a segment from each block participates in one shuffle (we call it **segment shuffle**). The size of the segment in the shuffle varies and is randomly determined by the nonce value. For each shuffle operation, a random block pair are selected, and the location of the block segment and its size are also varies over their value range. Each block is treated as wrapped so that shuffles of different segment sizes are possible.

Algorithm 1 Example of Shuffle Operation

/* Given a cache line of n bits and the unit size of a bits, the cache line has $q = n/a$ units and is to be shuffled r rounds,*/

/* divided the cache line into two parts: units 0 to $q/2-1$ form one part and the rest another part */
/* start from the first round */
round_count =1;
while round_count $<$ r **do**
  /* generate an array of $q/2$ random numbers RN, $RN(i) \in \{0, q/2-1\}$ */
  $RN = randomNumberGen()$;
  /* one round shuffle */
  for $k = 0$ *to* $q/2-1$ **do**
    $i = q/2+k$;
    $j = RN(k)$;
    temp = unit(i);
    unit(i) = unit(j);
    unit(j) = temp;
  end for
  round_count++;
end while

Figure 3.4 shows two examples of shuffle operation between block i, $B(i)$, and block j, $B(j)$, with the size of shuffle segments are 2 bits and 5 bits, respectively. For the 5-bits segment shuffle, the segment in $B(i)$ is wrapped; and the segment value "11001" is to be replaced by "01010" from $B(j)$.

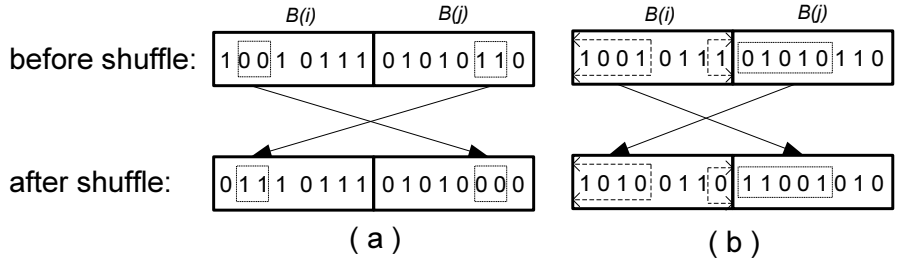

Figure 3.4: Segment Shuffle between Blocks of Different Segment Size (a) 2 bits (b) 5 bits

After the line shuffle, a permutation will be performed on each of the new blocks. The rotate shift operations are used for permutation. Each block will be left rotated shifted a random number of bits, which is controlled by the nonce value.

The nonce is an encryption of a unique value that has three fields, as shown in Figure 3.5: 1) memory cache line address, associated with the memory location of the data encrypted, 2) random value, for high unpredictability of the unique value, and 3) the counter, for a different access to the same memory cache line.

With the nonce constructed in this way, same memory data of different memory location or different access to even a same memory location, will have a different tag. Therefore, replaying any previous observed data and tag pair in the attack will be de-

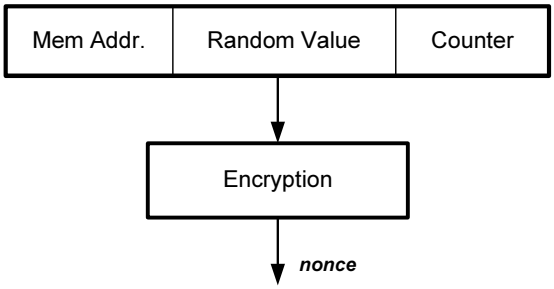

Figure 3.5: Nonce Construction

tected in the authentication. In addition, the encryption operation will make the nonce uniform random, which is required in our tag generation.

The operation of tag generation design is given in Algorithm 2.

Algorithm 2 specifies an inverse transform function to generate an m-bit uniform distributed value (for tag) from an n-bit uniform random variable (i.e encrypted cache line).

3.3 Tag Size Selection

We want that the tag size is small while effective for a maximal security.

Here the security can be measured in the exploration space of the brutal force attack **since our tag in the space is uniform distributed**. The bigger the space, the secure of the design.

There are two ways that the brutal force attack can be applied: on the tag value, and on the tag generation process. The space for each case is derived below.

Again, assume the cache line size is n, tag size m, the number of blocks in the cache line is n/m. The shuffle space with r shuffle rounds is

$$S_{shuffle} = ((C^2_{n/m}.m^2.\alpha)^{\beta} \tag{3.1}$$

where C^b_a is the combinatorial function, α and β are, respectively, used for controlling the segment size and shuffle round. The values of α and β affect the $S_{shuffle}$ value and also influence the design complexity for tag generation logic. High α and β will lead to a high hardware cost.

The permutation space is

$$S_{permutation} = m^{n/m}. \tag{3.2}$$

The total exploration space for tag generation through the inverse transformation algorithm is

$$\begin{aligned} S' &= S_{shuffle}.S_{permutation} \\ &= ((n(n-m)/2.\alpha)^{\beta}.m^{n/m} \end{aligned} \tag{3.3}$$

On the other hand, the exploration space for tag value is

$$S'' = 2^m. \tag{3.4}$$

Therefore, the effective exploration space for tag is

$$S = \min\{S', S''\}. \tag{3.5}$$

Algorithm 2 Tag Generation

/* Given a cache line of n bits, the tag size of m bits, the shuffle rounds β, and the nonce V for control the tag generation*/

/* determine the number of blocks, q */
$q = n/m$;
V=getNonce(n, q, r);
/* divide the cache line, L */
$L = B(1)||B(2)\ldots||B(q)$;
/* shuffle the cache line r rounds*/
for *each round* $k, k \in \{1, \beta\}$ **do**
  /* get the round control for each block */
  $v(k) = getRandomControl(V,k)$;
  /* select two random blocks, $B(i)$ and $B(j)$ */
  $(i, j) = get2blocks(v(k))$;
  /* get the size of the segment for shuffle operation in the range of $[0, \alpha]$*/
  $segSize = getSegSize(v(k))$;
  /* get the segment position pos_a, pos_b for block B(i) and B(j), respectively. */
  $(pos_a, pos_b) = getSegPosition(v(k))$;
  /* swap the segments between Block(i) and Block(j) */
  $BsegSwap(B(i), B(j), pos_a, pos_b, segSize)$;
end for
/* left rotate shift each block */
for *each block*, $B(i), i \in \{1, q\}$ **do**
  /* get the number of bits to be shifted from v(i) */
  $shiftBits = getNumberOfShiftBits(v(i))$;
  $leftRotateShift(B(i), shiftBits)$;
end for
/* Bit-XOR all blocks to obtain the tag value */
tag=B(1);
for *each block*, $B(i), i \in \{2, q\}$ **do**
  $tag = tag \oplus B(i)$;
end for

By using the above analytical model, one can adjust the tag generation algorithm and customize the tag size for given security and design requirements.

4 Experimental Evaluation

To verify our design approach, we have developed an evaluation platform and compared our design with the most related and state of art designs.

4.1 Evaluation Platform

Our baseline system is built based on a configurable Xtensa $LX4$ embedded system processor from Tensilica [15]. The processor system can be configured with different instruction set, clock speed, memory hierarchy, and related cache/memory access time.

The details of system parameters configuration is list in Table4.1

Table 4.1: Xtensa Base Processor Configuration

Parameter	Value
Core Speed	694 MHz
I-Cache	2KB
D-Cache	2KB
Cache Line	256 bits
Cache Access Time	1.44ns
System RAM	1MB
System ROM	4MB
Memory Access time	25.92ns (18ccs)
Instruction Bus Width	4B
Data R/W Bus Width	4B

Our design for memory authentication is modeled in VHDL hardware description language at RTL level. It is simulated with ModelSim [16] for the design functional verification and is synthesized by Synopsys Design Compiler [17] with the 65*nm* technology library. The Design Compiler provides the area, power consumption and delay of the design. The extra delay incurred by the tag generation/data authentication logic is then incorporated back to the application simulation.

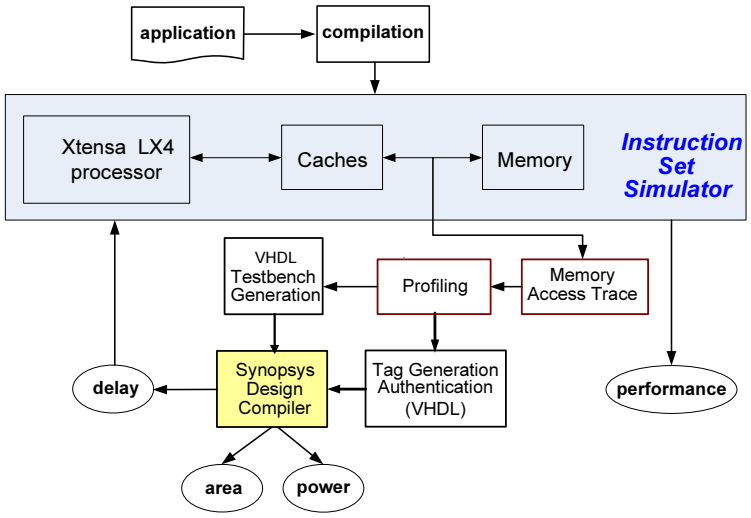

Figure 4.1: Evaluation Platform

Eight applications are selected from the embedded system benchmark suite MiBench [18]. The C codes of these applications are compiled with Xtensa XCC compiler and simulated on the Xtensa cycle-accurate Instruction Set Simulator (ISS).

We compare our design with other two closely related memory authentication approaches, Yan's work [11] and Rogers' work [10]. All three works use the block cipher encryption in tag generation. To be fair, we use a same encryption function, AES, in each of the designs. All three designs also use a similar approach: storing tags off-chip to save on-chip memory costs.

4.2 Simulation Results

The experiment results are given in the following two sub sections.

Tag Selection

From Section 3.3, we know that the tag search space is related to two parameters: α and β.

For demonstration of how those two parameters affect the tag size selection, Table 4.2 shows the search space size of S', S'', and S with different tag size (listed in the first column) under varied α and β values. The space size is measured in number of different values in the space. The tag value search space S" is purely determined by the tag size and is given in the second column; the space S' from the tag generation process and the effective search space S with different tag generation designs are given in Columns 3&4 and 5&6, respectively. As can be seen from the table, a large tag size may not result in a large search space (S) for high security; similarly, a sophisticated and expensive tag generation algorithm can be totally nullified by a small size tag. For example, for the tag generation design with $\alpha = 16$ and $\beta = 1$, a tag size of 48 bits will be an optimal selection, with a maximum of 2.81E+14 values in the search space S; A smaller tag will reduce the search space, hence security. On the other hand, for the same 48-bit tag size, use of the design with $\alpha = 32$ and $\beta = 2$ will bring no security enhancement, rather than consuming more on-chip resources. For $\alpha = 32$ and $\beta = 2$, the best tag size is 64 bits, as has been highlighted in the table.

Table 4.2: Tag Exploration Space

Tag Size (bit)	S"	β=1, α=16		β=2, α=32	
		S'	S	S'	S
8	2.56E+02	4.02E+34	2.56E+02	8.18E+40	2.56E+02
16	6.55E+04	9.07E+24	6.55E+04	1.78E+31	6.55E+04
24	1.68E+07	2.51E+20	1.68E+07	4.76E+26	1.68E+07
32	4.29E+09	5.04E+17	4.29E+09	9.26E+23	4.29E+09
40	1.10E+12	7.92E+15	1.10E+12	1.40E+22	1.10E+12
48	2.81E+14	3.94E+14	**2.81E+14**	6.72E+20	2.81E+14
56	7.21E+16	4.02E+13	4.02E+13	6.58E+19	7.21E+16
64	1.84E+19	6.60E+12	6.60E+12	1.04E+19	**1.04E+19**
72	4.72E+21	1.51E+12	1.51E+12	2.28E+18	2.28E+18
80	1.21E+24	4.43E+11	4.43E+11	6.39E+17	6.39E+17
88	3.09E+26	1.56E+11	1.56E+11	2.15E+17	2.15E+17

Given a value pair for α and β, we can find an optimal tag size. Similarly, for a given tag size, we can tune α and β for a largest tag search space, hence achieving as high as possible security.

Since neither Yan's nor Roger's work includes any investigation on how the tag size should be selected and they use a fixed tag size in their design, for comparison, we implement two different designs based on our design approaches: one with the tag size of 64 bits for comparison with Yan's work (where 64 bit tag is adopted), and another with the tag size of 128 bits for comparison with Rogers' 128-bit tag design.

Overhead Savings

Table 4.3 gives hardware resource overheads of the tag generation designs from the three design approaches. Each design is modeled in VHDL, and their area cost, power consumption and delay are obtained from the Synopsys Design Compiler. For the designs from our approach and Yan's for 64-bit tag, their overheads are given in Rows 3&4; and for the 128 bit tag designs, they are presented in Rows 6&7. The relative overhead savings of our design as compared to each of the existing designs are given in row 5 and 8, respectively. As can be seen from table, our design incurs a low on-chip resource overhead than the two existing designs.

Table 4.3: On-Chip Overhead

		Area (μm^2)	Leak. Power (μW)	Delay (ns)
64-bit tag	Ours	304935.50	1134.50	99.31
	Yan's	508423.50	2365.60	207.42
	Overhead Saving	40%	52%	52%
128-bit tag	Ours	355407.14	1320.45	100.12
	Rogers'	582367.00	2131.40	280.50
	Overhead Saving	39%	38%	64%

As can be seen from Table 4.3, our design incurs less overhead in area, power and delay as compared to each of the existing designs.

Based on the tag generation design, we derive the extra clock cycles incurred by the tag generation logic for each memory access, and run each application in the instruction simulator with the delay overhead. Table 4.4 shows the normalized execution time (based on the baseline system without any data protect logic).

Table 4.4: Performance Overhead

Applications	Ours	Yan's	Rogers'
adpcm	28.19%	88.25%	77.11%
dijkstra	261.59%	814.19%	711.73%
jpeg	170.97%	533.50%	466.28%
qsort	220.94%	695.35%	607.38%
rijndael	362.27%	1123.87%	982.65%
sha	109.37%	347.83%	303.61%
stringsearch	260.41%	712.55%	622.76%
susan	114.04%	354.82%	310.17%
Average	190.97%	583.80%	510.21%

As can be seen from Table 4.4, our design brings about a smaller performance overhead than the Yan's and Rogers' designs.

We calculate the memory consumption of the tag off-chip storage. Table 4.5 shows the memory overhead, measure in KB, of the three designs for each application. The size of each application is given in Column 2. The relative memory overhead as compared to the application size is given in the last row. As can be seen from the table, our design has a relative smaller off-chip cost than Yan's and Rogers'.

Figure 4.2 shows the relative overhead savings in terms of on-chip area, power consumption, delay, application performance, and off-chip memory. Compared to Yan's

Table 4.5: Memory Overhead(KB)

Applications	Total Size (KB)	64-bit tag		128-bit tag	
		Ours	Yan's	Ours	Rogers'
adpcm	308	77	88	154	178
dijkstra	436	109	124	218	251
jpeg	759	190	216	380	438
qsort	403	101	115	202	232
rijndael	334	84	95	167	193
sha	301	75	86	151	174
stringsearch	295	74	84	148	170
susan	459	115	131	230	265
AVG		25%	29%	50%	58%

design, our design saves 40% area, 52% power, an average of 67% execution time, and 12% off-chip memory. The savings can also be found when comparing with Rogers', with 39%, 38%, 63%, and 13% reductions in area, power consumption, execution time, and off-chip memory, respectively.

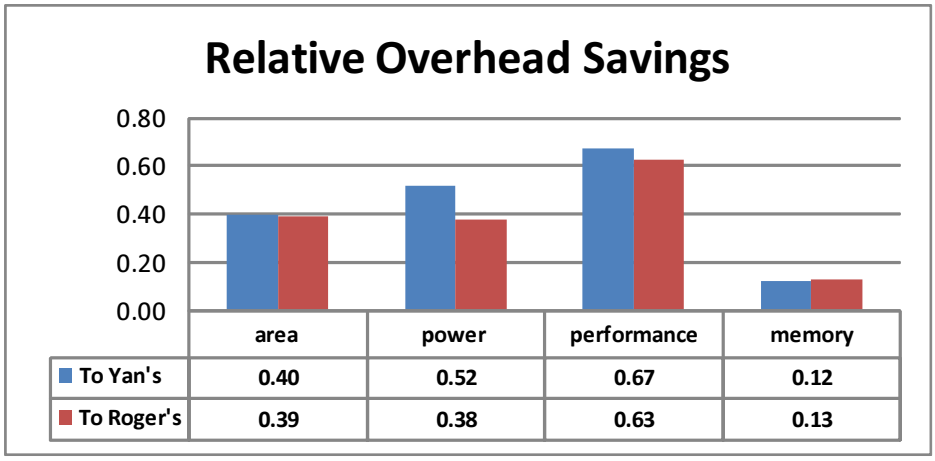

Figure 4.2: Overhead Savings over Yan's

5 Conclusion

Use of tag for memory data integrity protection is an effective approach for secure embedded system design. This paper addresses the cost issue related to the tag design, which is critical to the resource stringent embedded systems. We have presented a design approach for tag generation and tag size selection, where the tag size is closely related to the tag generation logic, and the tradeoff between the cost and security can be played.

Our design offers a high flexibility to meet different levels of security while at lowest cost as possible.

Experiments have been conducted to evaluate our design, which shows our design is most cost effective than the existing state-of-art designs, significant savings - about 40% on area and power consumption, 60% on performance, and 10% on off-chip memory - can be achieved.

6 Appendix: Uniform Distribution

We developed some C code to generate the results for a set of operations on the random inputs that are uniform distributed. Those results are then tested under a distribution fitting scheme with the EasyFit distribution fit tool [19]. Table 6.1 shows the results. For each operation listed in Column 1 (where X and Y are uniform random, and RT represents rotate for shift operations), its most four fit distributions are given in Columns 2&5. The last four columns provide the rank for four typical distributions: Normal, Lognormal, Exponential and Uniform, for each type of operations. For example, for the operation of variable X plus a constant show in Row 3, the first four best fit distributions are: Uniform, Error, Johnson SB and Power; If the operation results is fitted to the Normal distribution, the fitting rank will 12.

As can be seen from the table, multiplication/addition/AND/OR of two uniform random variables do not generate a uniform random result. But for other operations, their results best fit to the uniform distribution. We call such operations *Uniform random Conversion enable (UC) operation*. The fitting errors of the UC operations are given in Table 6.2.

As can be seen from Table 6.2, the fitting errors are very small, about 0.0153925 on average.

Table 6.1: Distribution Fit

operation	First Four Most-Fit Distributions				Fitting Rank of Four Typical Distributions			
	1st fit	2nd fit	3rd fit	4th fit	Normal	Lognormal	Exponential	Uniform
x+6	Uniform	Error	Johnson SB	Power	12	37	46	1
4*x	Uniform	Error	Gen. Pareto	Johnson SB	9	34	37	1
x/8	Uniform	Error	Johnson SB	Gen. Pareto	5	34	35	1
x mod 16	Uniform	Error	Johnson SB	Gen. Pareto	6	33	27	1
x+y	Cauchy	Johnson SB	Error	Normal	4	32	15	53
x*y	Triangular	Error	Logistic	Normal	4	22	31	32
NOT x	Uniform	Error	Gen. Pareto	Johnson SB	9	33	36	1
x AND y	Log-Logistics	Dagum	Frechet	Gen. Pareto	29	7	33	49
x OR y	Kumarsawamy	Pareto 2	Exponential	Expoential (2P)	24	34	3	53
x XOR y	Uniform	Error	Johnson SB	Gen. Pareto	8	33	36	1
RT x $\gg$ y	Uniform	Error	Johnson SB	Gen. Pareto	7	32	35	1
swap	Uniform	Error	Johnson SB	Gen. Pareto	9	33	38	1

Table 6.2: UC Random Fit Error

UC operation	Uniform Distribution Fit Error
4*x	0.00658
6+x	0.00565
x/8	0.03001
x mod 16	0.05293
NOT x	0.00526
x XOR y	0.00731
rotate shift	0.00855
swap	0.00685
avg	0.0153925

Bibliography

[1] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *32nd Annual Symposium on Foundations of Computer Science*, 1991.

[2] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 168 – 177, 2000.

[3] G. E. Suh, D. Clarke, B. Gasend, M.van Dijk, and S. Devadas. AEGIS:architecture for tamper-evident and tamper-resistant processing. In *International Conference on SuperComputing*, 2003.

[4] 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.

[5] R.C. Merkle. Protocols for public key cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*, pages 122 – 34, 1980.

[6] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. pages 295 – 306, 2003.

[7] 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, CHES 2007.*, 2007.

[8] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: a block-cipher mode of operation for efficient authenticated encryption. In *ACM conference on Computer and communications Security*, 2001.

[9] A. McGrew, D and J Viega. The galois counter mode of operation (GCM). Technical report, Submission to National Institute of Standards and Technology, Federal Information Processing Standards, 2004.

[10] A. Rogers and A. Milenkovic. Security extensions for integrity and confidentiality in embedded processors. *Microprocessors and Microsystems*, 33(5-6):398 – 414, 2009.

[11] 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.

[12] C Fruhwirth. New methods in hard disk encryption. Technical report, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005.

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

[14] Barry L. Nelson Jerry Banks. *Discrete-event system simulation*. Prentice Hall, 2010.

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

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

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

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

[19] Easyfitxl. (http://www.mathwave.com/articles/fit-distributions-excel.html).