Available online at www.sciencedirect.com
ScienceDirect
Journal of System Architecture 00 (2013) 1–20
JSA

TSV: A Novel Energy Efficient Memory Integrity Verification Scheme for Embedded Systems

Satyajeet Nimgaonkar[1,*], Mahadevan Gomathisankaran[2], Saraju P. Mohanty[3]

Department of Computer Science and Engineering, University of North Texas, Denton, TX 76203, USA

Abstract

Embedded systems are ubiquitous in this era of portable computing. These systems are empowered to access, store and transmit abundance of critical information. Thus their security becomes a prime concern. Moreover, most of these embedded devices often have to operate under insecure environments where the adversary may acquire physical access. To provide security, cryptographic security mechanisms could be employed in embedded systems. However, these mechanisms consume excessive energy that cannot be tolerated by the embedded systems. Therefore with the focus on achieving energy efficiency in cryptographic Memory Integrity Verification (MIV) mechanism, we present a novel energy efficient approach called Timestamps Verification (TSV) to provide memory integrity verification in embedded systems. This paper elaborates the proposed approach along with its theoretical evaluation, simulation results, and experimental evaluation. The results prove that the energy savings in the TSV approach are in the range of 36% to 81% when compared with traditional MIV mechanisms.

Keywords: Embedded Systems, Memory Integrity Verification, Timestamps Verification

*Phone: 940-206-7023

Email addresses: satyajeetnimgaonkar@my.unt.edu (Satyajeet Nimgaonkar), mgomathi@unt.edu (Mahadevan Gomathisankaran), saraju.mohanty@unt.edu (Saraju P. Mohanty)

[1]Trusted Secure Systems Lab, http://tssl.cse.unt.edu/People/SatyajeetNimgaonkar

[2]Trusted Secure Systems Lab, http://www.cse.unt.edu/~mgomathi

[3]Nanosystem Design Lab, http://www.cse.unt.edu/~smohanty

1. Introduction

An embedded system, in contrast to a general purpose computing system, is a dedicated system designed to serve a specific task within a larger system. They are high performance systems, flexible enough to perform a variety of computing tasks in a cost effective manner. Due to technological advances in computing, embedded systems have now evolved into complex systems. The modern day embedded devices are often small, portable and highly interconnected. They are capable of tracking, storing information and even transmitting essential data over the Internet.

These characteristics have made embedded systems pervade in all facets of human life. They are being used everywhere from home media systems, portable players, smart phones, automobiles, embedded medical devices to mission critical defense systems. As the dependence on these systems increase, so also does the sensitivity of information accessed, stored and communicated by these devices, increase. This information may also include confidential personal data including secret passwords, credit card information, and bank account details. Thus it has now become ever more important to secure the embedded systems from leaking out this critical information to unauthorized entities.

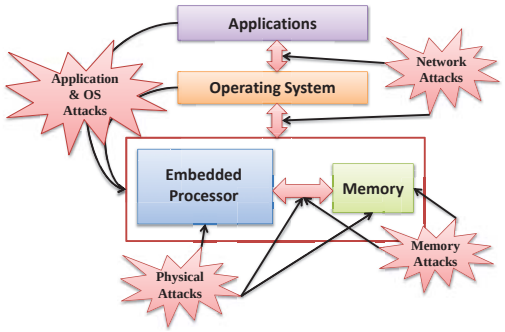


Figure 1: Threats to an Embedded System

To make matters worse, the operating environment of embedded systems allow the adversary to have complete control of the computing node. For example, supervisory privileges along with complete physical and architectural object observational capabilities. The design phase of embedded systems often does not provide for the security axis thus increasing the security risks. Moreover, embedded systems are physically dispersed to public locations that are available to potential attackers. This makes them vulnerable to physical attacks. In addition, the embedded systems are also vulnerable to attacks on memory, network based attacks, attacks from malicious applications and infected operating systems, as shown in Figure 1.

Though these embedded systems are small and flexible, their design is complex which adds to their security issues. Despite these problems, embedded systems are deployed widely. Hence motivated attackers can exploit these vulnerabilities to extract confidential information from these devices. For example, Mobile and Smart Device Security Survey [1] conducted by Mocana Corporation in Spring 2011, revealed that 65% corporate personnel require a regular attention from their information technology staff for mobile and smart phone based device attacks.

Traditionally, researchers have implemented software and hardware mechanisms for integrating security in embedded systems. Software-only security solutions include software obfuscation, software watermarking, software encryption, privilege separation, introspection, shepherding, and sandboxing, which are aimed at different applications to incorporate security mechanisms at the application and operating system (OS) level. Thus the root of trust for the entire system is entrusted in the security of the OS. However commodity OS are significantly large with huge code base and are often prone to security vulnerabilities. Moreover, the software solutions share the memory with other softwares and the OS that could potentially contain a vulnerability

Table 1: Energy Consumption of Encryption Algorithms [2]

	Algorithms							
	DES	3DES	IDEA	CAST	AES	RC2	RC4	RC5
Energy Consumption (μJ/Byte)	2.08	6.04	1.47	1.21	1.73	3.93	0.79	0.89

Table 2: Energy Consumption of Hashing Algorithms [2]

	Algorithms					
	MD2	MD4	MD5	SHA	SHA1	HMAC
Energy Consumption (μJ/Byte)	4.12	0.52	0.59	0.75	0.76	1.16

and hence provide an entry point to an attack. Thus the security they provide is inadequate. The most common software attack that exploits the software vulnerabilities in application code and OS is a Buffer Overflow Attack [3, 4]. Hardware security approaches, on the other hand, include secure processor architectures like, ABYSS [5], AEGIS [6], Arc3D [7, 8, 9], XOM [10] and HIDE [11], that propose modifications in the CPU architecture to protect the confidentiality and integrity of applications. Typically they employ hardware Encryption [12, 13, 14, 15] and Memory Integrity Verification (MIV) mechanisms to preserve confidentiality and integrity of the application and data running on the system.

Embedded systems are highly resource constrained. Most of these devices are battery powered and it is essential to have minimal energy consumption to achieve high speed and performance. The hardware security mechanisms, though highly secure, are computationally intensive and account for excessive energy consumption. This degrades the performance of the system and becomes a critical issue. As a case study, Potlapally et. al. [2] present a framework to analyze the energy consumption of security mechanisms and protocols. Their work primarily focuses on investigating the impact of security processing on the *battery-life* constraint of an embedded system. For battery powered embedded systems, the biggest challenge is the trade-off between energy and performance due to security processing. As this trade-off increases, so also does the *battery gap* in embedded systems. The Tables 1 and 2 show the energy consumption in μJ/Byte for most commonly known and used encryption and hashing algorithms. In order to provide security in embedded systems, it is essential to design energy efficient implementations of these security mechanisms and protocols. Thus their research becomes a backbone for addressing the challenges related to energy efficient security mechanisms in battery constrained embedded systems.

The possible solutions to implementing security mechanisms within the energy constraints of the embedded systems is to optimize the encryption and the memory integrity verification mechanisms. In this research, the emphasis is specifically on optimizing the energy consumption of memory integrity verification module in embedded processors. The overhead of the memory integrity verification mechanism is several hashing operations, $\log N$ for N data nodes. A large of amount of energy is spent by the processor in computing the hashes for all the data blocks it writes to the external memory. Similary, energy is also spent while verifying these hashes during a read operation. While it may be unavoidable to save the energy spent during the write operation, a lot of research is now focused towards making the

verification process during the read operation energy efficient.

Prior research [16], [17], [18], and [19], present mechanisms to reduce energy overhead in memory integrity verification mechanisms. In our previous research [20], we have demonstrated the use of sensors in an embedded system to detect and protect against the attacks on memory integrity. However, in this research our goal is to present a mechanism that is independent of any sensors and can be employed by all the embedded systems. With this motivation, we present the *Timestamps Verification (TSV)* mechanism to provide energy efficient memory integrity verification in embedded systems.

The rest of this paper is organized a follows. The Section 2, first defines Memory Integrity Verification and the possible attacks against it. This is followed by the MIV Architecture in Section 3. The Section 4 describes the proposed scheme of Timestamps Verification followed by their Experimental Evaluation in Section 5. Section 6 presents a detailed Literature Survey and finally Section 7 presents the Conclusion and Future Work of this research.

2. Memory Integrity Verification - MIV

The Memory Integrity Verification (MIV) property can be defined as follows. A processor communicates with memory $\mathcal{M}$. Memory $\mathcal{M}$ has two attributes, addresses A and contents V. It maintains associations between addresses and contents. A read of memory at address A denoted by $\mathcal{M}[R, A]$ returns the value associated with A. A write into memory address A of value V is denoted by $\mathcal{M}[W, A, V]$. A write of A with value V immediately followed by a read of address A must return value V. As memory reads and writes have a notion of time, the model needs to associate time T with reads and writes as $\mathcal{M}[R, A, T]$ and $\mathcal{M}[W, A, V, T]$.

Definition 1. *(Memory Integrity) A read of address A at time T should return the value written to address A at time $T' < T$ such that no other write to A occurs between time T' and T. In other words: $\mathcal{M}[R, A, T] = V$ if an only if $\exists \mathcal{M}[W, A, V, T']$ for $T' < T$ and $\forall\, t \in [T'+1, T-1]$, $\nexists \mathcal{M}[W, A, *, t]$. At $T = 0$ the entire memory is initialized with value* 0.

The possible attacks on memory integrity are *splicing* attack, *spoofing* attack and *replay* attack. In splicing attack the adversary modifies the associations between the memory addresses and its values. For example, if value V_A is associated with address A and value $V_{A'}$ is associated with address A', a splicing attack would return the value $V_{A'}$ for a memory read corresponding to address A. In a spoofing attack the adversary modifies the value V_A to a random value V'_A. In a replay attack the adversary modifies the association between the value and the time. For example, if value V_A is associated with address A at time T and the value V^*_A is associated with address A at time T^*, and $T < T^*$, a replay attack would return the value V_A when a memory read is performed at time $t > T^*$.

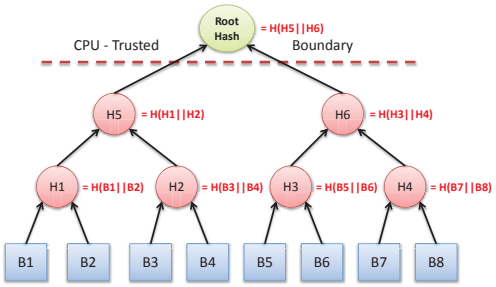


Figure 2: Merkle Hash Tree

In cryptography, a hash function – $\mathcal{H}$, is used to protect the integrity of a message. A sender creates a message authentication code (MAC) using a secret key K together with the message m as $MAC_m = \mathcal{H}(m||K)$. It sends both the message m and the corresponding MAC_m to the receiver. The receiver, which shares the secret key K with the sender, can verify

the integrity of the message by recomputing the MAC. The security properties, collision resistance, pre-image resistance and second pre-image resistance of the hash function $\mathcal{H}$ ensures that any modification to message m or MAC_m go undetected with negligible probability.

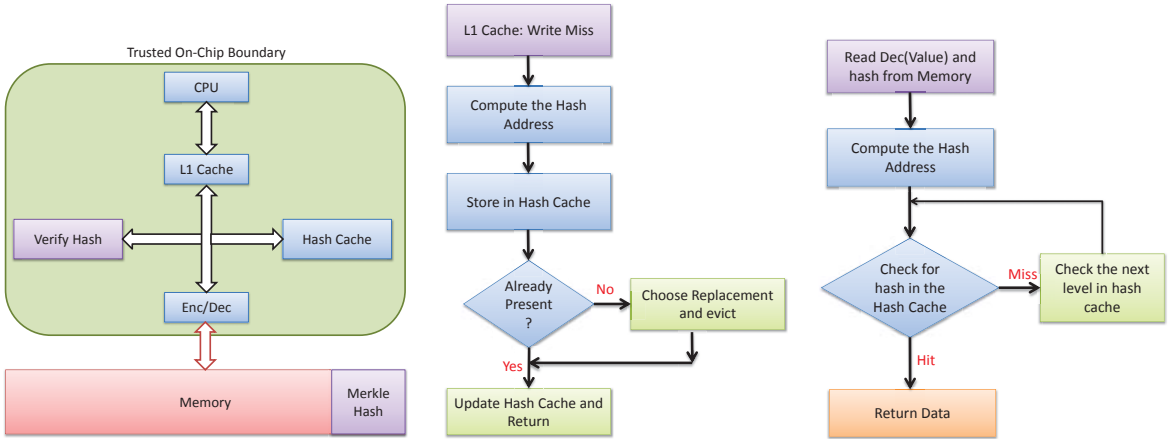
One solution for the memory integrity problem is to use message authentication codes. Processor generates a MAC $h_{V,A} = \mathcal{H}(V||A||K)$ for every memory write that stores a value V in address A. This MAC, $h_{V,A}$, is stored in the memory. On every memory read the processor computes the MAC $h'_{V,A}$ and verifies it against the MAC stored in the memory. This solution can protect memory integrity against splicing and spoofing attacks. A replay attack will still succeed as the MAC does not have any notion of time and hence the adversary could replay both the value and its corresponding MAC.

Protecting memory against replay attacks requires the processor to have some memory about the recentness of the value. A Merkle Hash Tree, shown in Figure 2, provides an optimal solution for this problem, requiring least amount of on-chip memory. A Merkle hash tree of address range $[A, A + k]$ creates a tree of hashes with $k + 1$ leaves corresponding to addresses $A, A + 1, \ldots, A + k$. Any write to an address $A + i$ in this address space can modify all the hash values from the leaf node corresponding to $A + i$ upto the root of the tree. Any read from address $A + i$ needs to check the hash values along the path from the leaf node $A + i$ upto the root of the tree. The root of the tree is stored in the trusted processor storage, so that it cannot be tampered. All the other tree nodes along with the leaf nodes can be kept in the untrusted memory.

3. Architecture of Memory Integrity Verification

The motivation for proposing the Memory Integrity Verification (MIV) architecture is to simulate the behavior of the Merkle Hash Tree based Memory Integrity Verification in a secure processor architecture. This is essential to determine the number of hash invocations required per memory access. In this implementation, this process is being simulated by creating a custom *Hash Cache* with n levels to store the hash addresses. This is a novel extension to our previous research [20], where a probabilistic model was proposed to estimate the average number of hash invocations required per memory access, based on whether the hash of a block of memory is present in the cache or not. The total energy consumed during the hash verification process is calculated using this Energy Consumption Model. The configuration of hash cache is similar to the Level 1 Data cache configuration.

Figure 3a shows the working of memory integrity verification in a secure processor architecture. Both the read and write operation of memory are considered. In the write operation, shown in Figure 3b, for every *write miss* in the level 1 data cache, the corresponding *hash* and *hash address* are calculated. This *hash* is stored in the Hash cache in the *hash address*. If the hash address already exists in the cache, then it is updated or else if it is not present in the cache a suitable replacement block is selected and evicted to store the hash. The data is then encrypted through the memory encryption block before saving it to the off-chip untrusted memory. During a read operation, Figure 3c, the data is first decrypted and the hash address of the data is recomputed. The hash address is checked against all the addresses in all the levels of the hash cache. If there is a *hit* in the hash cache, the hash of the data is checked against the hash that is stored in the hash cache. If the hash matches then it is concluded that the state of the data is valid, if no then, it is concluded that the data is corrupted and the CPU aborts any operation on this data. If the hash is not present in the hash cache then the CPU checks for the next level hash in the hash cache until it reaches the root



(a) Integrity verification architecture (b) Write operation (c) Read operation

Figure 3: Memory Integrity Verification in Secure Processor Architectures

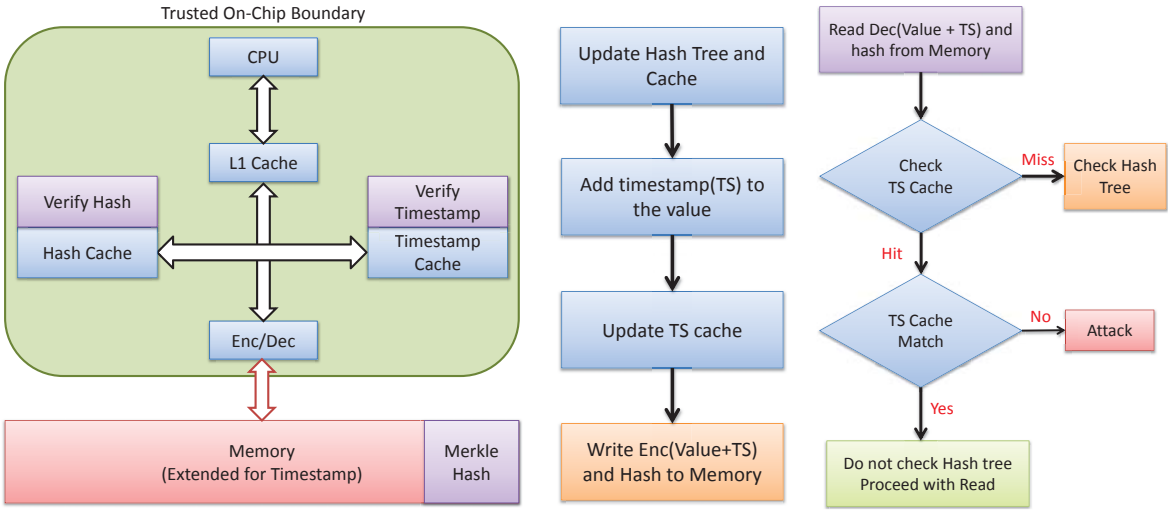
hash.

Algorithm 1 Function hash_addr(level, block_addr, index) to compute Hash Address at each Level in a Merkle Hash Tree

Require: hash_size; block_size; tree_size
Require: max_hash_levels; BlockOffset
Ensure: level $<$ max_hash_levels
Ensure: index $<$ tree_size
block_addr $\Leftarrow$ block_addr - BlockOffset[level]
if level $==$ 0 **then**
block_number $\Leftarrow$ block_addr/block_size
else
block_number $\Leftarrow$ block_addr/hash_size
end if
block_number $\Leftarrow$ block_number / tree_size
block_number $\Leftarrow$ block_number $\wedge$ index
hash_addr $\Leftarrow$ BlockOffset[level+1] + (block_number $\times$ hash_size)
return (hash_addr)

The pseudo code of the function to compute hash address at each level in a Merkle hash tree is given in Algorithm 1. The function takes three arguments — level of *hash tree*, *data block address* and *index for the hash level*. The *hash block size, data block size, tree size, maximum levels of hash tree* and *offset address* for each level are pre-initialized. At first, the block address is calculated based on the level and its offset address. Depending on the level, the block number is calculated which is then computed with tree size and index to fetch the hash address.

The overhead of integrity verification architecture is several hashing operations, $\log N$ for N leaf nodes. One can cache some of these hash tree nodes to increase the efficiency. The granularity of a leaf node can be increased beyond a single word to an entire cache block. Despite these optimizations, such hash trees are expensive primarily due to the cost of the underlying hash function. The two types of cost associated with hash function are performance cost and energy cost. AEGIS [6] charges 160 cycles for each hashing operation presumably at a cost of 2 cycles per round for 80 rounds of SHA [21]. This is a very high performance cost for a memory integrity architecture that spawns many hash function instantiations for each read and write. Hence, more efficient mechanisms for memory integrity protection are required.



(a) Timestamp Verification Architecture (TSV) (b) Write operation (c) Read operation

Figure 4: Timestamps Mechanism

4. Proposed Timestamps Verification (TSV) Mechanism

As discussed in Section 3, the hash verification process imposes severe performance and energy costs on the entire system. This cannot be tolerated in energy constraint embedded systems like smart phones, network sensors, network routers, etc. Hence it is imperative to design novel energy efficient MIV mechanisms without compromising the security of the system. With this motivation, the paper proposes a novel memory integrity verification mechanism using Timestamps, in this section.

4.1. Timestamps Mechanism

This approach exploits the principle of locality. Most of the programs exhibit this behavior of using a small set of addresses, called working set, during an epoch of execution. The intuition is to make the processor remember the timestamps of working set of addresses. The proposed TSV architecture is shown in Figure 4a. In this approach the processor timestamps every value on a write operation and the timestamp is stored with the data in encrypted form. The timestamps can be generated using any pseudorandom number generator such as block ciphers. These timestamps are stored on-chip in a separate cache known as the Timestamp cache with limited space. When the cache is full, older timestamps are evicted, hence only a limited set of timestamps are stored. In this mechanism, there are two operations — Write and Read as shown in Figures 4b and 4c respectively.

During a write operation, the CPU first constructs the hash tree and stores the hash in the hash cache. Then, a unique timestamp is generated for the memory block, which is also stored in the Timestamp cache for later verification. The memory block and the associated timestamp are then encrypted through the encryption module and saved in the main memory. On a read operation, the CPU first fetches the decrypted memory block and its timestamp. It then compares the associated timestamp with the ones already present in the timestamp cache. If its a miss, then it performs the Merkle hash tree verification to check for

its integrity. On the other hand, if it is a hit then it compares the value of the two timestamps. If both the values match, it is concluded that the integrity of the data is intact else it is concluded that there is an attack on the system and the CPU aborts any further operation on that data. The security of the integrity protection is derived from the security of the encryption function. Any splicing, spoofing, and replay attack will be detected as the decryption function will create a pseudo-random timestamp which is very less likely to match with the timestamp stored on-chip. The security is proportional to the size of the timestamp. For an N-bit timestamp the probability that any attack on memory will not be detected (attack succeeds) is $\frac{1}{2^N}$.

The primary advantage of this approach is in the hash verification phase. If the timestamp is available in the timestamp cache then the CPU is no longer required to re-construct the hash tree to check the integrity of each memory block it reads from the memory. Also, while verifying the hash, the CPU has to spend energy in comparing the hash values at each level, till there is a hit. This may be required until the root hash is verified. Hence a lot of energy is spent by the processor in this process. Thus the timestamps mechanism aims at mitigating both the performance cost and energy cost during the verification phase.

The Timestamp Cache is an energy efficient data cache, whose configuration is optimally selected by performing a series of performance simulations. This approach requires the memory to be modified to store the timestamps. This modification is transparent to the processor.

5. Experimental Evaluation

This section presents the details about the simulation framework/test bench with some of the specifics about the configurations used to arrive at the given results. It then presents the baseline simulations illustrating the performance of traditional Merkle hash tree verification followed by proposed TSV scheme. Finally the performance analysis section is concluded with an elaborate comparison between the baseline results and the TSV mechanism.

5.1. Simulation Framework

The simulation framework is based on Simplescalar Tool Set [22], which is configured to execute ARM binaries. Since the primary goal of this paper is to demonstrate the energy efficiency of the proposed memory integrity verification mechanism in embedded systems, MiBench [23] embedded benchmark suite has been used, that best replicates the variety of practical applications run on embedded devices. The results obtained from different benchmark programs are presented to thoroughly demonstrate the efficiency of the proposed TSV mechanism. All the simulations performed are cache based simulations, using the *sim-cache* simulator in simplescalar. The cache configurations used for the simulations are given in Table 3. Here, the configuration of the Level 1 Data Cache is optimally selected to complement the typical configurations of an Embedded ARM processor [24]. Moreover, the TSV mechanism is only implemented for data and hence no emphasis is given to instructions or the I-cache in the simulations.

Table 3: Cache Configurations

Cache	Specifications
L1-D Cache	4KB, 1-way, 32B Line
L2-D Cache	None

5.2. Basecase Simulations

The approach of Merkle hash tree in MIV accounts to excessive energy consumption. To measure this, a MIV architecture has been proposed in Section 3. Here the algorithm is used

to compute and count the *number of hash invocations* required per data miss in the Level 1 Data Cache, to verify the integrity of the data. This technique is a novel contribution of this paper. Hence, given that the energy consumption per hash invocation is known, the results obtained from the algorithm can be used to calculate the total energy consumption of the MIV mechanism. For comparison, the configuration of hash cache is kept similar to the L1 Data cache configuration. Therefore the hash cache is *1 way associative* i.e. Direct Mapped cache. At this point it should be noted that the number of hash invocations grow incrementally as the verification proceeds from the first level to the root level.

Table 4 shows a relationship between the DL1 misses and hash verification at each level, for 14 embedded benchmark applications. The Merkle hash tree constructed is a $4-ary$ hash tree with 14 levels. The hash level (HL) indicates the number of times the hash verification was invoked. Here the total DL1 misses are distributed amongst 14 hash levels to indicate in which level the miss was verified. Hence in general, the total DL1 misses is equal to the summation of misses verified at each hash level as given in equation 1. Here n represents maximum hash level.

$$\text{Total DL1 Misses} = \sum_{i=1}^{n} \text{Misses at each HL}_i \quad (1)$$

For example, in the case of *bitcount_large* application, there are a total of 893 misses in DL1 cache ($= HL1 + HL1 + HL2 + ... + Hl14$). Out of these misses, 113 are verified in Level 1. The number of hash invocation in Level 1 are $113 * 1 = 113$. Similarly, 719 misses are verified in Level 14 accounting for $719 * 14 = 10066$ hash invocations. Hence the total number of hash invocation for a particular benchmark can be calculated using the equation 2.

$$\text{Total Hash Invocations} = \sum_{i=1}^{n} \text{Misses at each HL}_i \times i \quad (2)$$

This can then be related to average energy consumption of the integrity verification hash function per invocation to calculate the total energy consumption of the integrity verification module as given in equation 3.

$$\text{Total Energy} = \text{Total Hash invocations} \times \text{Energy per invocation} \quad (3)$$

For instance, from Table 2, if it is assumed that the energy consumed per hash invocation by the SHA-1 algorithm is 0.76 μJ, then for *bitcount_large* application, the energy consumption for hash verification will be $0.76 * 10066 = 7.65mJ$ at Level 14 alone. Hence the total energy consumption is equal to the addition of energy consumption at each hash level. From this discussion, it is evident that as the verification process traverses up the levels, the energy consumption increases rapidly. Moreover, for all the applications in the above simulation, a majority of the DL1 misses are verified in the last level-14, thus consuming maximum energy possible. These statistics are used as a baseline for comparisons with the proposed Timestamps Verification mechanism.

5.3. Evaluation of TSV mechanism

In the timestamps mechanism, a TS cache is created to store unique timestamps associated with each memory block. This timestamp is later used by the processor to verify integrity of the block, before reverting to the conventional approach of Merkle hash tree. The timestamps generated are small in size and hence the size of TS Cache is also small compared to DL1 cache. This is a significant advantage when related to embedded systems, that are stringent with size and energy requirements. The timestamps stored in the TS Cache may have a size of either 8 Bytes or 16 Bytes. Therefore the size of the TS cache may vary depending on the size of the timestamps. The performance of TS cache is analyzed based on the number of timestamps it can store. These can be 8, 16,

Table 4: Hash Accesses at different Levels

Benchmarks	HL1	HL2	HL3	HL4	HL5	HL6	HL7	HL8	HL9	HL10	HL11	HL12	HL13	HL14
dijkstra_small	93	46	626	0	2	0	0	11	0	3	1	0	8	1500344
jpeg_large	65	35	56	10	0	0	0	0	0	0	7	0	0	2923220
lame_large	66	21	44	12	15	6	0	1	0	0	0	4	1	39221183
lame_small	66	21	44	12	15	6	0	1	0	0	0	4	1	3135942
patricia_large	130	19	9	4	1	0	0	0	0	0	6	8	0	9137840
patricia_small	130	19	9	4	1	0	0	0	0	0	6	8	0	1529667
qsort_small	61	14	21	18	0	0	121	0	0	0	0	0	8	1876005
math_large	74	30735	18	7	7	0	0	0	0	0	0	12	0	1287768
sha_large	31556	18	94	3	2	0	0	1	0	0	4	0	2	232566
sha_small	2950	18	94	3	2	0	0	1	0	0	4	0	2	23269
stringsearch_small	125	140	21	0	0	0	0	3	0	0	1	198	0	1756
bitcount_large	113	20	37	3	1	0	0	0	0	0	0	0	0	719
bitcount_small	112	21	38	3	1	0	0	0	0	0	0	0	0	706
dijkstra_large	93	46	626	0	2	0	0	11	0	3	1	0	8	7020003

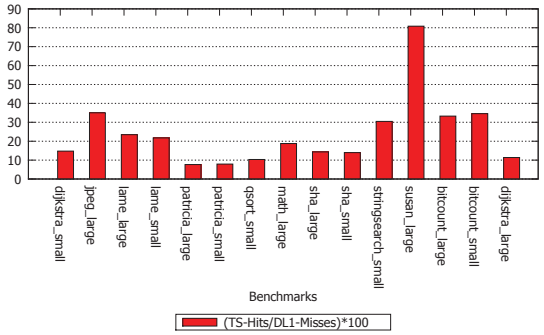
32 and 64. For each type, four separate configurations are analyzed, based on the *Number of Ways* and *Number of Sets* in the TS cache. The details about various configurations is given in Table 5. The TS cache size is equal to product of *Number of Timestamps* stored in the cache and the *Size of each Timestamp*. Hence assuming the size of timestamps is 8 Bytes, the size of TS cache storing 8 timestamps is 64 Bytes. Similarly the size of TS cache storing 16, 32 and 64 timestamps is 128 Bytes, 256 Bytes and 512 Bytes respectively.

For each TS cache capable of storing different number of timestamps, detailed performance results have been presented along with their relative performance. Figure 5a, shows the percentage of TS hits w.r.t the DL1 misses for a TS cache capable of storing 8 timestamps, for 15 embedded benchmark applications. Recall from the discussion in Section 4.1, a hit is TS cache reduces the energy consumption of integrity verification as the values in the TS cache can be trusted and no Merkle hash tree verification is required. In this case, the percentage of TS hits is highest for benchmark application - *susan* with almost 81% and the least for benchmark application *patricia* with almost 8%. The average percent of TS hits is 36.5%. Figure 5b, shows the percentage of TS Hits w.r.t the DL1 misses for a TS cache capable of storing 16 timestamps. Here, the percentage of TS hits is highest for benchmark application - *susan* with 88% and the least for benchmark application *patricia* with almost 17%. The average percentage of TS hits is almost 56%. Figure 5c, shows the percentage of TS hits w.r.t the DL1 misses for a TS cache capable of storing 32 timestamps. Here, the percentage of TS hits is highest for benchmark application - *susan* with 91% and the least for benchmark application *sha* with almost 18%. The average percent of TS hits is almost 71%. Finally, Figure 5d shows the percentage of TS hits w.r.t the DL1 misses for a TS cache capable of storing 64 timestamps. Here, the percentage of TS hits is highest for benchmark application - *stringsearch* with 96% and the least for benchmark application *sha* with almost 18%. The average percent of TS hits is almost 77%.

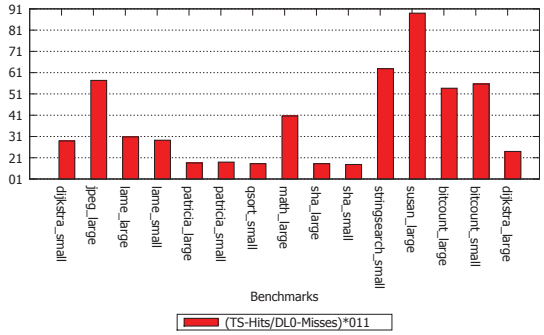
The results about the four individual configurations of each timestamps - 8, 16, 32 and 64 are presented in detail. These give an idea of which configuration could yield the best energy savings for a particular TS cache. Figure 6a, shows the relative performance of the 4 possible configurations of TS cache capable of storing 8 timestamps. Here, the configuration *1Sets-8Ways* results in higher TS hits and thus greater energy savings for most of

Table 5: TS Cache Configurations

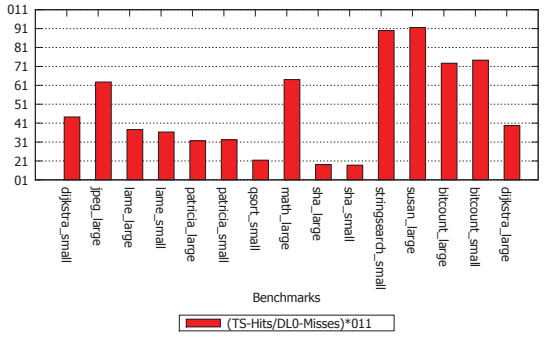
# Timestamps stored in TS Cache	Configurations
8	1 Set-8 Ways; 2 Sets-4 Ways; 4 Sets-2 Ways; 8 Sets-1 Way
16	1 Set-16 Ways; 2 Sets-8 Ways; 4 Sets-4 Ways; 8 Sets-2 Way
32	1 Set-32 Ways; 2 Sets-16 Ways; 4 Sets-8 Ways; 8 Sets-4 Way
64	1 Set-64 Ways; 2 Sets-32 Ways; 4 Sets-16 Ways; 8 Sets-8 Way



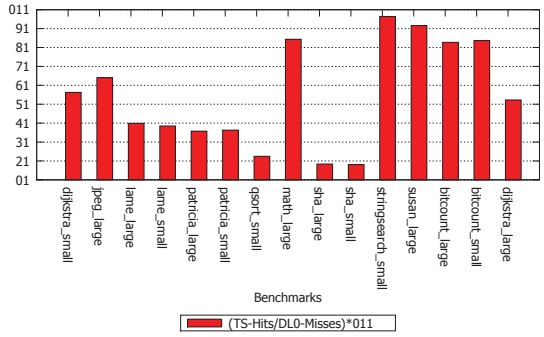
(a) Average percentage of TS Hits for TS Cache with 8 Timestamps



(b) For 16 Timestamps



(c) For 32 Timestamps

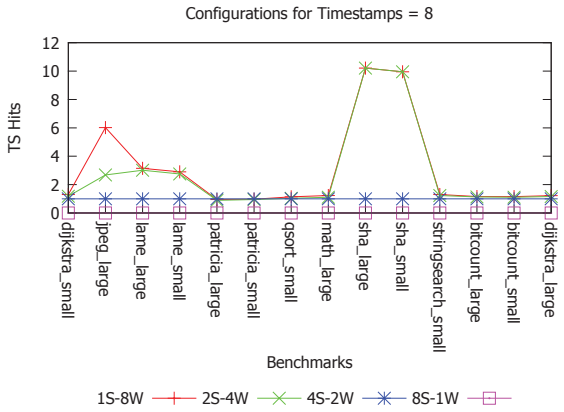


(d) For 64 Timestamps

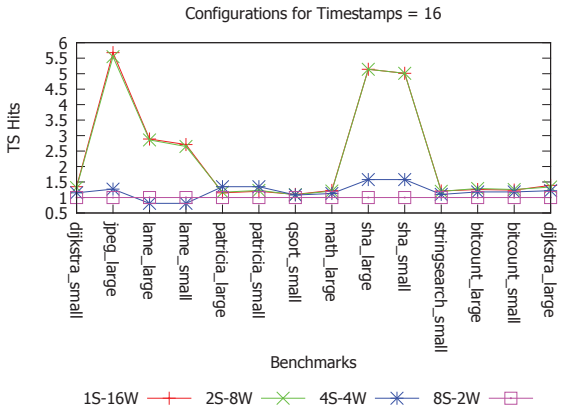
Figure 5: Average percentage of TS Hits

the benchmark applications. For most of the benchmarks, the configuration *2Set-4Ways* yields similar TS hits as compared to *1Sets-8Ways*. On the other hand, the configuration *8Sets-1Ways*, results in no TS hits and thus no energy savings for any benchmark applications. The TS hits are greater for configuration *2Sets-4Ways* than *4Sets-2Ways*. Therefore it can be said that the impact of *Number of Ways* on TS hits is more than *Number of Sets* in case of TS cache with 8 timestamps. The TS hits and in turn energy savings increase with an increase in *Number of Ways*.

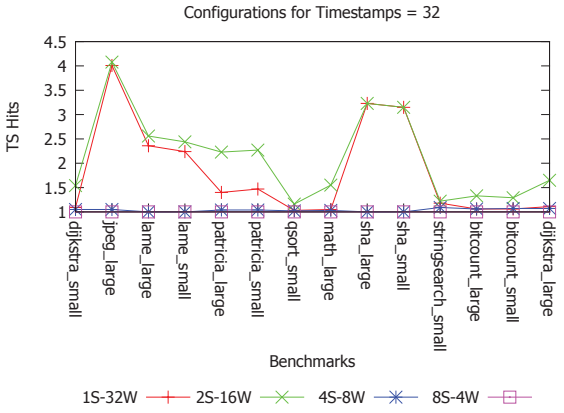
Figure 6b, shows the relative performance



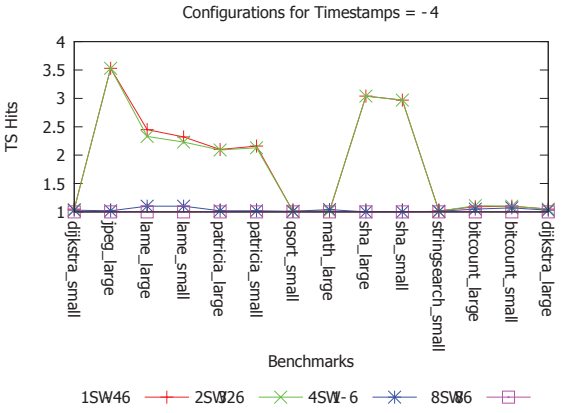
(a) Relative performance of four configurations for TS cache with 8 timestamps



(b) For 16 timestamps



(c) For 32 timestamps



(d) For 64 timestamps

Figure 6: Relative performance of various TS cache configurations

of the 4 configurations of TS cache capable of storing 16 timestamps. Here, the configurations *1Sets-16Ways* and *2Sets-8Ways* result in the highest TS hits and thus highest energy savings for most of the benchmark applications. Whereas, the configuration *8Sets-2Ways* results in the least TS hits for most of the benchmark applications. Here again, the TS hits are more with an increase in the *Number of Ways*. Figure 6c, shows the relative performance of the 4 configurations of TS cache capable of storing 32 timestamps. Here again, the configurations *1Sets-32Ways* and *2Sets-16Ways* result in the highest TS hits and thus highest energy savings for most of the benchmark applications. On the contrary, the configurations *4Sets-8Ways* and *8Sets-4Ways* yield similar results with lower TS hits for most of the benchmark applications. For the benchmark application of *math_large*, the 3 configurations of *2Sets-16Ways*, *4Sets-8Ways* and *8Sets-4Ways* result in similar TS hits. Therefore again in this case, the TS hits are more with an increase in the *Number of Ways*. Fig-

ure 6d, shows the relative performance of the 4 configurations of TS cache capable of storing 64 timestamps. Here, the configurations *1Sets-64Ways* and *2Sets-32Ways* provide similar results with higher TS hits whereas the configurations *4Sets-16Ways* and *8Sets-8Ways* provide similar results with lower TS hits. For the benchmark application *math_large*, all the 4 configurations yield similar results. Here again, the TS hits are more with an increase in the *Number of Ways*.

Based on the discussion in Sections 2 and 5.2, the percentage of TS hits are directly related to the energy savings. Recall, that for each miss in the DL1 cache, a lot of energy is consumed in re-constructing the Merkle hash tree and verifying the root hash. But in the case of Timestamp mechanism, for every hit in the TS cache corresponding to a DL1 miss, there is no need spend energy in Merkle hash verification. Hence these hits directly relate to energy savings in an embedded system. At this point, it is important to emphasize that an increasing amount of energy is lost during Merkle hash verification as it moves from level 1 to level 14. To show this effect, the hash invocations at each level are computed, for TS cache storing 8, 16, 32 and 64 timestamps using the same approach described in Section 5.2. This is used to calculate the *weighted averages* of all the benchmarks in the timestamps simulation. The geometric mean of these weighted averages for TS = 8, 16, 32 and 64 is compared with that of basecase simulations. This is shown in Table 6. Here, the geometric mean of weighted averages of hash invocations steadily decreases for timestamps configurations as compared to basecase simulations. Thus fewer invocations result in increased energy savings.

Table 6: Geometric Mean of Weighted Averages of Hash Invocations

Basecase	TS = 8	TS = 16	TS = 32	TS = 64
48672.78	30537.76	27925.43	10106.65	7674.06

Using the equation 3 and the values in Table 6, we present a synopsis of average energy savings in the TSV approach when compared to basecase simulations. The energy savings for TS cache with 8 timestamps is least with 36% and that for TS cache with 64 timestamps is the highest with 81%, as shown in Table 7. The energy savings presented here are the averages of all the configurations in a particular TS cache. The Section 5.3.2 presents a detailed analysis of the energy savings in each configurations and its impact on the entire system. Also, it is important to emphasize that the goal of this research is to present a variety of options using the TSV mechanism instead of just presenting the best option. It is ultimately up to the chip designer to evaluate all the possible options and their impact on the system before choosing the most suitable option.

Table 7: Energy Savings in Timestamps Approach

Timestamps (TS Cache)	Energy Savings
8 timestamps	36%
16 timestamps	62%
32 timestamps	73%
64 timestamps	81%

5.3.1. Theoretical Evaluation for TSV Mechanism

In this section, we present a theoretical basis for the TSV mechanism and aim at theoretically justifying our results. There has been significant prior research [25, 26, 27, 28, 29] that is focused on modeling cache and its performance. We leverage this previous research to present a model that encapsulates TS cache and its performance. Similar to the cache organization described by Agrawal et.al [26], TS cache organization - TS_C, is denoted as by (S, A, B), where S is the number of sets, A is the degree of associativity and B is block size. The

TS cache size in bytes is the product of number of sets, degree of associativity and the block size. The number of blocks (i.e. the timestamps) and each block size (i.e. the timestamp size) is fixed in the TSV mechanism. Therefore the number of sets and the degree of associativity manifests the working set of the TS cache. Further, analysis is presented to show how altering the size of the working set influences the energy savings of the TS cache.

The Working Set Model [30] was proposed by Peter J. Denning, to model the behavior of programs in a general purpose computer system. It was observed in operating system, that programs obey the principle of locality (i.e. a program's past referenced pages) may be used as a good predictor for near future, to be re-referenced pages. Thus the working set model was proposed to provide a general resource allocation solution in operating systems. From the program's standpoint, the working set of information $W(t, \tau)$ of a process at time t is the collection of information referenced by the process in time interval $(t - \tau, t)$, where τ is the working set parameter.

In terms of the TS cache, the working set is dependent on the possible combinations of number of sets and the degree of associativity. This is represented by the TS cache configurations. Also, in the TSV mechanism, the *time* factor in the working set model is represented by the *total number of accesses*. It is important to note that the TS cache configurations resemble any general purpose cache configurations - Direct Mapped, Fully Associative and Set Associative. For instance 1 Sets - 8 Ways, 1 Sets - 16 Ways, 1 Sets - 32 Ways, 1 Sets - 64 Ways resemble a fully associative cache. Whereas 8 Sets - 1 Ways resembles a direct mapped cache with the rest as set associative.

The Independent Reference Model [25] was proposed by Rao to analyze the performance of a cache. This model is analytically tractable and presents miss rate estimates for direct-mapped, fully-associative, and set-associative caches using the arithmetic and geometric distributions for page reference probabilities. The effectiveness of the TSV mechanism is analyzed by the energy savings offered by the TS cache. The energy savings are directly related with TS cache hit rate (or 1 - TS cache miss rate). Therefore the Independent Reference Model serves best to analyze the TS cache performance in terms of its miss rate for varying TS cache size.

With reference to [25], the miss rate or the fault rate in a cache is written as

$$F_f = \sum_{t=1}^{n} p_t q_t \tag{4}$$

where F is the fault rate, f is the replacement policy, n is the logical pages in the backing store, p_t is the page reference probability and q_t is the probability of not finding a page in the cache.

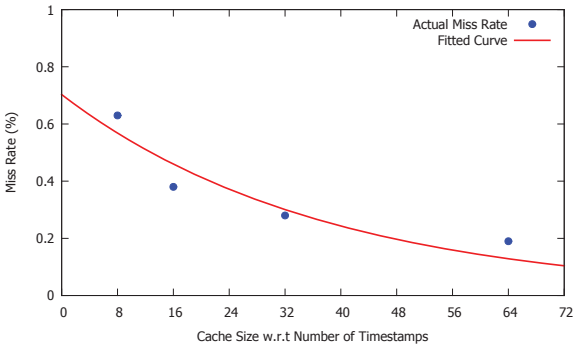


Figure 7: Miss Rate vs. TS Cache Size in terms of Number of Timestamps

If a graph of the fault rate/miss rate is to be plotted versus the cache size, then it is observed that the fault rate decreases exponentially with an increase in the cache size. This trend is also exhibited in the TS cache and is shown in Figure 7. This figure represents a graph of miss rate vs. TS cache size in terms of number of timestamps. Since the block size is kept constant at 8 bytes, the TS cache increases from 64 Bytes for 8 timestamps to 512 Bytes for 64 timestamps. The miss rate shown in this

figure is calculated by averaging the miss rates obtained from all the configurations i.e. working sets in a particular TS cache. It is therefore evident that the miss rate decreases exponentially as the size of the TS cache increases.

5.3.2. Overhead Evaluation of TSV Mechanism

The proposed timestamps mechanism does not require a unique timestamp to be stored in the TS cache for all the data blocks accessed by the L1 Data Cache. A L1 Data cache of size 4KB with 32 Byte block size contains a total of 128 cache blocks. However, the TS cache is configured to store only 8, 16, 32 or 64 timestamp blocks. Moreover, a timestamp is unique data pertaining to a particular cache block and hence its block size is significantly smaller than the cache block size. The timestamp block size can be either 8 Bytes or 16 Bytes as compared to 32 Bytes in a L1 Data cache. In this research, the timestamp block size is set to 8 Bytes. Table 8 below presents an analogy between the size of the TS cache, its energy savings and its area overhead with respect to L1 Data cache of size 4KB. The area overhead is calculated using the equation 5 below.

$$\text{TS Cache Area Overhead} = \frac{\text{Area of TS Cache}}{\text{Area of L1 Data Cache}} \times 100 \quad (5)$$

Since the size of each TS block is fixed at 8 Bytes, the size (in bytes) of the TS cache storing 8, 16, 32 and 64 timestamps is 64, 128, 256 and 512 respectively. For each TS cache mentioned above, the energy savings (in percentage) for every configuration and its area overhead (in percentage) is presented. The area overhead for the TS cache of size 64, 128, 256 and 512 bytes is 1.56%, 3.12%, 6.23% and 12.50% respectively. Therefore the overhead of the TS cache is significantly low as compared to the energy savings it can offer.

The energy savings of TS cache with 8 timestamps is in the range of 33.11% to 56.92% with an area overhead of 1.56%. The energy savings of TS cache with 16 timestamps is in the range of 46.91% to 74.10% with an area overhead of 3.12%. The energy savings of TS cache with 32 timestamps is in the range of 57.46% to 89.57% with an area overhead of 6.25%. And finally the energy savings of TS cache with 64 timestamps is in the range of 68.84% to 92.36% with an area overhead of 12.5%. It is important to stress that the impact of *number of ways* or *associativity* on the energy savings is much more than that of *number of sets*. The energy savings decrease as the *associativity* decreases. This is expected as a general characteristic in caches.

Table 9: Energy Savings/Area Overhead (E/O) factor for TS Cache Configurations

TS Cache Configurations	E/O Factor (%)
1 Sets-8 Ways	36.49
2 Sets-4 Ways	34.23
4 Sets-2 Ways	21.23
8 Sets-1 Ways	0.00
1 Sets-16 Ways	23.75
2 Sets-8 Ways	23.70
4 Sets-4 Ways	17.06
8 Sets-2 Ways	15.03
1 Sets-32 Ways	13.36
2 Sets-16 Ways	14.33
4 Sets-8 Ways	9.57
8 Sets-4 Ways	9.19
1 Sets-64 Ways	7.39
2 Sets-32 Ways	7.37
4 Sets-16 Ways	5.60
8 Sets-8 Ways	5.51

To emphasize the importance of the energy savings offered by the proposed TSV mechanism and to provide a comparison between the configurations of various TS caches, we calculate a new parameter — the *Energy Savings/Area Overhead* factor, in short the *E/O* factor. The value of the E/O factor is in the

Table 8: Energy Savings vs. Area Overhead in TS Cache

Number of Timestamps	Size of TS Cache with Block Size = 8 Bytes	TS Cache Configurations	Energy Savings (%)	TS Cache Area Overhead w.r.t L1 Data Cache of size 4KB (%)
8	64	1 Sets-8 Ways	56.92	1.56
		2 Sets-4 Ways	53.40	
		4 Sets-2 Ways	33.11	
		8 Sets-1 Ways	0.00	
16	128	1 Sets-16 Ways	74.10	3.12
		2 Sets-8 Ways	73.94	
		4 Sets-4 Ways	53.23	
		8 Sets-2 Ways	46.91	
32	256	1 Sets-32 Ways	83.50	6.25
		2 Sets-16 Ways	89.57	
		4 Sets-8 Ways	59.78	
		8 Sets-4 Ways	57.46	
64	512	1 Sets-64 Ways	92.36	12.50
		2 Sets-32 Ways	92.10	
		4 Sets-16 Ways	70.02	
		8 Sets-8 Ways	68.84	

range of 0% to 100% and it should be as high as possible. The Table 9 below, shows the E/O factor for all the TS cache configurations. The E/O factor is the highest for the TS cache configuration of 1 Sets-8 Ways (Fully Associative) with 36.49%. Whereas it is the lowest for TS cache configuration of 8 Sets-1 Ways (Direct Mapped) with 0%. It shows that this configuration did not get any TS hits and hence resulted in no energy savings. However if this configuration is ignored, it can be seen that the E/O factor decreases steadily with 5.51% as the lowest for the TS cache configuration of 8 Sets-8 Ways. This trend suggests that the energy savings offered by the TS cache does not increase at same rate at which its area overhead increases. Since the impact of area overhead is more, it pulls down the E/O factor down gradually. This is shown in figure 8. This figure presents the average E/O factor along with its lower and upper bounds for TS cache with 8, 16, 32 and 64 timestamps. The fitted curve shows that the average E/O factor decreases exponentially with increase in the size of the TS cache. The fluctuation in the upper and lower bounds, also known as the *error fluctuation* reduces as the size of the TS cache increases. This trend closely resembles the *Law of Diminishing Returns*. It also important to stress that as the size of the TS cache increases, the impact of associativity on the energy savings, decreases. Hence a for a smaller TS cache, it is important to select the fully associative cache configuration to achieve higher energy savings. Whereas for a larger TS cache, even the direct mapped configuration could yield comparable energy savings as the fully associative cache configuration.

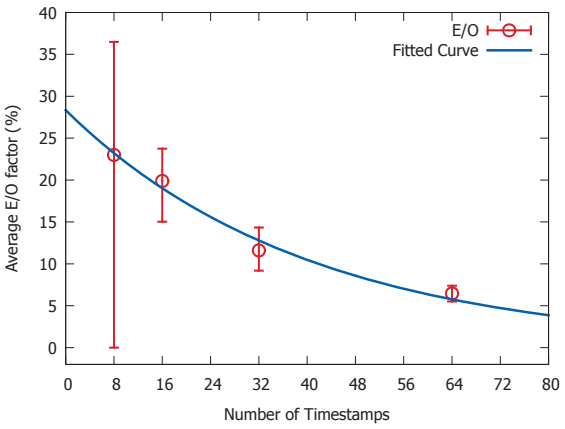


Figure 8: Average E/O factor for each TS cache

6. State-of-the-Art in Memory Security of Embedded Systems

It is conspicuous that security in embedded systems is significant concern considering the unsafe environment in which they have to operate and the sensitivity of data they handle. Conventional security mechanisms are impeded due to severe energy constraints imposed by the limited resourced embedded system. Hence considerable modifications to existing security solutions are required before they can be deployed. It can be safely said that software-only security approaches fail in securing embedded systems as the adversary can physically observe or tamper with the state of applications. To achieve high level of security, the root of trust must be entrusted on the hardware of these embedded systems. Hence as seen in Section 1 a lot of research has been done in developing secure processor architectures to provide privacy and integrity in embedded applications. Even though the hardware approaches provide reliable security against physical and software attacks, considerable modifications are needed to be done before they can be adopted in an energy constrained embedded system.

A substantial amount of research is dedicated to come up with energy efficient architectures for embedded systems that aim at providing utmost security while consuming limited energy. For instance, Shi et al. in [31] present a secure and fast architecture for authenticating shared memory. To incorporate memory authentication in their architecture, the authors propose a new scheme of Authentication Speculative Execution that not only is efficient but also offers lower average performance degradation of less than 5%. In another research [32], the authors present a model for key masking to achieve minimal energy overhead in embedded systems. The experimental results reveal that the technique supports up to 2.5% energy overhead savings. Power-Smart System-On-Chip Architecture [33], presents an architecture for preventing sensitive information leakage via timing, power and electromagnetic channels. This architecture depends on a current sensing module to measure the power and current consumption of the system. They achieve significant success in measuring the current consumption of the system while limiting the power overhead to less than 12% of the total power. In [34], Roger et al. describe an efficient hardware mechanism to protect integrity of softwares by signing each instruction block during program installation with a cryptographically secure signature. This technique serves as a secure and performance efficient alternative to conventional memory integrity verification module. While [35], presents an architectural approach to address memory spoofing attacks. The data protection techniques proposed in this paper, achieves high level of security with significantly low performance overhead.

The research in energy efficient embedded system security is directed in various other areas in computing. Along with architectural techniques, there has been significant research in improving the performance of existing cryptographic memory integrity verification mechanisms. In [16], Gassend et al. combine caches and hash trees to deliver a performance ef-

ficient memory integrity verification scheme. The paper presents CHash and LHash with varying cache block sizes to analyze the performance overhead of each configuration. While in [19], an Address Independent Seed Encryption (AISE) is proposed along with Bonsai Merkle Trees (BMT). Simulation results prove that this technique reduces the overhead on the system by 12% to 2% as compared to traditional approaches. Some other researches [36] and [37], present new low overhead encryption algorithms and cache level tuning to achieve relatively low performance degradation.

Although, all the above papers, present new and energy efficient mechanisms for embedded system security, there are still new avenues to improve energy efficiency in embedded system security. This is the primary motivation of this paper. The proposed TSV technique yield energy savings in the range of 36% to 81% for integrity verification process. The architecture and algorithms presented for both the schemes make them plausible to be implemented in next generation embedded devices.

7. Conclusions and Future Research

7.1. Summary and Conclusions

In designing computing systems there is always a trade-off between security and energy consumption. A classic example of this are the Embedded Systems. Embedded devices are typically fast, miniaturized and specific to their application and hence often have severe energy constraints. As they now handle a lot of critical information, security is of utmost importance in them. Therefore the need arises to design new security mechanisms so that their energy consumption is minimal while still preserving the security of the system.

In this paper, the focus is specifically on reducing the energy consumption of Memory Integrity Verification mechanisms in embedded systems. The paper contains two novel contributions. First, it presents a scheme to precisely measure the number of hash invocations per DL1 misses and compute the energy consumption based on the hash levels. Then it presents a novel energy efficient technique using Timestamps for integrity verification. The results show that the energy savings with TSV mechanism can range from 36% to 81%, compared to basecase results, based on the number of timestamps that can be stored in the TS cache.

7.2. Future Research

Currently, research is in progress to measure the total energy consumption of an embedded processor and compare it with the energy consumption of the memory integrity verification mechanisms. This would give an exact measure of the energy savings that can be achieved by the proposed energy efficient MIV techniques, with respect to the entire embedded processor. This is an intriguing as well as a challenging prospect as the total energy consumption of an embedded system processor is difficult to measure. This would usher new sets of designs and challenges in energy efficient security mechanisms for embedded systems.

References

[1] Mobile and Smart Device Security Survey 2011, http://images.tmcnet.com/tmc/whitepapers/documents/whitepapers/2011/4683-mobile-smart-device-security-survey-2011.pdf (2011).

[2] N. R. Potlapally, S. Ravi, A. Raghunathan, N. K. Jha, Analyzing the energy consumption of security protocols, in: Proceedings of the 2003 international symposium on Low power electronics and design, ISLPED '03, ACM, New York, NY, USA, 2003, pp. 30–35. doi:10.1145/871506.871518. URL http://doi.acm.org/10.1145/871506.871518

[3] C. Cowan, F. Wagle, C. Pu, S. Beattie, J. Walpole, Buffer overflows: attacks and defenses for the vulnerability of the decade, in: DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, Vol. 2, 2000, pp. 119 –129 vol.2. doi:10.1109/DISCEX.2000.821514.

[4] Z. Shao, C. Xue, Q. Zhuge, M. Qiu, B. Xiao, E.-M. Sha, Security protection and checking for

embedded system integration against buffer overflow attacks via hardware/software, Computers, IEEE Transactions on 55 (4) (2006) 443 – 453. doi:10.1109/TC.2006.59.

[5] S. White, L. Comerford, Abyss: an architecture for software protection, Software Engineering, IEEE Transactions on 16 (6) (1990) 619 –629. doi:10.1109/32.55090.

[6] G. Suh, C. O'Donnell, S. Devadas, Aegis: A single-chip secure processor, Design Test of Computers, IEEE 24 (6) (2007) 570 –580. doi:10.1109/MDT.2007.179.

[7] M. Gomathisankaran, A. Tyagi, Architecture Support for 3D Obfuscation, IEEE Trans. Computers 55 (5) (2006) 497–507. doi:10.1109/TC.2006.68.
URL http://viper.eng.iastate.edu/gmdev/pubs/ieeeTC06.pdf

[8] M. Gomathisankaran, A. Tyagi, Arc3d: A 3d obfuscation architecture, in: T. M. Conte, N. Navarro, W. mei W. Hwu, M. Valero, T. Ungerer (Eds.), HiPEAC, Vol. 3793 of Lecture Notes in Computer Science, Springer, 2005, pp. 184–199.

[9] M. Gomathisankaran, A. Tyagi, Tiva: Trusted integrity verification architecture, in: R. Safavi-Naini, M. Yung (Eds.), DRMTICS, Vol. 3919 of Lecture Notes in Computer Science, Springer, 2005, pp. 13–31.

[10] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, M. Horowitz, Architectural support for copy and tamper resistant software, in: Architectural Support for Programming Languages and Operating Systems, 2000, pp. 168–177.
URL citeseer.nj.nec.com/lie00architectural.html

[11] X. Zhuang, T. Zhang, S. Pande, Hide: an infrastructure for efficiently protecting information leakage on the address bus, SIGARCH Comput. Archit. News 32 (5) (2004) 72–84. doi:10.1145/1037947.1024403.
URL http://doi.acm.org/10.1145/1037947.1024403

[12] J. Daemen, V. Rijmen, The Design of Rijndael: AES - The Advanced Encryption Standard, Springer Verlag, Berlin, Heidelberg, New York, 2002.

[13] M. Gomathisankaran, R. B. Lee, Maya: A novel block encryption function, in: International Workshop on Coding and Cryptography, 2009.
URL http://viper.eng.iastate.edu/gmdev/pubs/maya.pdf

[14] M. Gomathisankaran, K.-M. Keung, A. Tyagi, Rebel - reconfigurable block encryption logic, in: E. Fernández-Medina, M. Malek, J. Hernando (Eds.), SECRYPT, INSTICC Press, 2008, pp. 312–318.

[15] M. Gomathisankaran, R. Lee, Tantra : A fast prng algorithm and its implementation, in: H. R. Arabnia, K. Daimi (Eds.), Proceedings of the 2009 International Conference on Security & Management, SAM 2009, July 13-16, 2009, Las Vegas Nevada, USA, 2 Volumes, CSREA Press, 2009, pp. 593–598.

[16] B. Gassend, G. Suh, D. Clarke, M. van Dijk, S. Devadas, Caches and hash trees for efficient memory integrity verification, in: High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on, 2003, pp. 295 – 306. doi:10.1109/HPCA.2003.1183547.

[17] G. Suh, D. Clarke, B. Gasend, M. van Dijk, S. Devadas, Efficient memory integrity verification and encryption for secure processors, in: Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on, 2003, pp. 339 – 350. doi:10.1109/MICRO.2003.1253207.

[18] M. Gomathisankaran, A. Tyagi, Relating boolean gate truth tables to one-way functions, Integrated Computer-Aided Engineering 16 (2) (2009) 141–150, 10.3233/ICA-2009-0307. doi:10.3233/ICA-2009-0307.

[19] B. Rogers, S. Chhabra, Y. Solihin, M. Prvulovic, Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly, in: Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on, 2007, pp. 183 –196. doi:10.1109/MICRO.2007.16.

[20] S. Nimgaonkar, M. Gomathisankaran, Energy efficient memory authentication mechanism in embedded systems, in: Electronic System Design (ISED), 2011 International Symposium on, 2011, pp. 248 –253. doi:10.1109/ISED.2011.34.

[21] Secure Hash Standard, National Institute of Standards and Technology, Washington, 2002, federal Information Processing Standard 180-2.
URL http://csrc.nist.gov/publications/fips/

[22] T. Austin, E. Larson, D. Ernst, Simplescalar: an infrastructure for computer system modeling, Computer 35 (2) (2002) 59 –67. doi:10.1109/2.982917.

[23] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, R. Brown, Mibench: A free, commercially representative embedded benchmark suite, in: Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on, 2001, pp. 3 – 14. doi:10.1109/WWC.2001.990739.

[24] ARM Architecture Reference Manual, http://www.altera.com/literature/third-party/ddi0100e_arm_arm.pdf/

(2000).

[25] G. S. Rao, Performance analysis of cache memories, J. ACM 25 (3) (1978) 378–395. doi:10.1145/322077.322081.
URL http://doi.acm.org/10.1145/322077.322081

[26] A. Agarwal, J. Hennessy, M. Horowitz, An analytical cache model, ACM Trans. Comput. Syst. 7 (2) (1989) 184–215. doi:10.1145/63404.63407.
URL http://doi.acm.org/10.1145/63404.63407

[27] E. Rothberg, J. Singh, A. Gupta, Working sets, cache sizes, and node granularity issues for large-scale multiprocessors, in: Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on, 1993, pp. 14 –25. doi:10.1109/ISCA.1993.698542.

[28] A. Smith, A comparative study of set associative memory mapping algorithms and their use for cache and main memory, Software Engineering, IEEE Transactions on SE-4 (2) (1978) 121 – 130. doi:10.1109/TSE.1978.231482.

[29] G. E. Suh, S. Devadas, L. Rudolph, Analytical cache models with applications to cache partitioning, in: Proceedings of the 15th international conference on Supercomputing, ICS '01, ACM, New York, NY, USA, 2001, pp. 1–12. doi:10.1145/377792.377797.
URL http://doi.acm.org/10.1145/377792.377797

[30] P. J. Denning, The working set model for program behavior, Commun. ACM 11 (5) (1968) 323–333. doi:10.1145/363095.363141.
URL http://doi.acm.org/10.1145/363095.363141

[31] W. Shi, H.-H. Lee, M. Ghosh, C. Lu, Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems, in: Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on, 2004, pp. 123 – 134. doi:10.1109/PACT.2004.1342547.

[32] C. Gebotys, Low energy security optimization in embedded cryptographic systems, in: Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on, 2004, pp. 224 – 229. doi:10.1109/CODESS.2004.240744.

[33] H. Vahedi, S. Gregori, Y. Zhanrong, R. Muresan, Power-smart system-on-chip architecture for embedded cryptosystems, in: Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on, 2005, pp. 184 –189. doi:10.1145/1084834.1084883.

[34] A. Rogers, M. Milenkovic, A. Milenkovic, A low overhead hardware technique for software integrity and confidentiality, in: Computer Design, 2007. ICCD 2007. 25th International Conference on, 2007, pp. 113 –120. doi:10.1109/ICCD.2007.4601889.

[35] O. Gelbart, E. Leontie, B. Narahari, R. Simha, Architectural support for securing application data in embedded systems, in: Electro/Information Technology, 2008. EIT 2008. IEEE International Conference on, 2008, pp. 19 –24. doi:10.1109/EIT.2008.4554261.

[36] C. Yan, B. Rogers, D. Englender, D. Solihin, M. Prvulovic, Improving cost, performance, and security of memory encryption and authentication, in: Computer Architecture, 2006. ISCA '06. 33rd International Symposium on, 2006, pp. 179 –190. doi:10.1109/ISCA.2006.22.

[37] A. Gordon-Ross, F. Vahid, N. Dutt, Automatic tuning of two-level caches to embedded applications, in: Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings, Vol. 1, 2004, pp. 208 – 213 Vol.1. doi:10.1109/DATE.2004.1268850.