Efficient Memory Integrity Verification Schemes for Secure Processors

Waseem Ahmad
wahmad@ece.uic.edu
Department of Electrical and Computer Engineering
University of Illinois, Chicago

Abstract

Single Chip Secure Processors have recently been proposed for variety of applications ranging from anti-piracy to trusted execution of distributed processes. Off-chip memory integrity verification and encryption are two fundamental tasks of a single-chip secure processor. Memory integrity verification is regarded as the main bottleneck in improving the performance of secure processors. Different schemes for efficient memory integrity verification have been proposed to date. Out of all these schemes, only those which employ Hash Trees (Merkle Trees) are able to prevent replay attacks at reasonable cost. Cached Hash Trees are an improvement over Simple Hash Trees in reducing Memory Bandwidth Pollution. We present, in this paper, couple of schemes which are distinguishable in terms of the required compiler support. These schemes are based upon the common motive to exploit temporal locality in memory access patterns, estimation of which is provided by compiler, for efficient memory integrity verification. The proposed schemes provide robust security and yet outperform cached hash trees by a considerable margin of 169% and 79% respectively.

1 Introduction

The use of tamper resistant hardware is getting common in those systems where goal is to run an application such that both integrity and the secrecy of the application are preserved. The need for such hardware may be essential for copy-proof software [5, 10] and trusted execution of an application on distributed computing platforms such as Grid [9]. Secure single-chip processors have recently been proposed [5, 4, 3] that allow trusted execution of applications with tamper resistance. XOM (eXecute Only Memory) [5] and Aegis [3] are two such secure processor architectures. In a secure single chip processor only the processor chip is trusted and operations of all other components including off-chip memory are verified by the processor. In order to protect data in off-chip memory from getting tampered or observed, memory integrity verification and encryption are two basic primitives that each secure single-chip processor has to support. Integrity verification provides tamper evidence in which any change in the running program's state is detected. On the other hand, encryption provides privacy of the code and data and make reverse engineering almost impossible. E.Suh et al in [2] state that the off-chip memory integrity verification is the most expensive operation as for as performance of a tamper-resistant system is concerned.

Off-chip memory integrity verification in XOM is provided by addressed MACs (Message Authenticated Codes) [5] this implementation is known to be unable to prevent replay attacks [1,7,10]. Aegis team has multiple implementations of memory integrity verification [2]. They first of all used Hash Tress (Merkle Trees) [1] which are able to prevent replay attacks. They, then, modified this idea a little bit to store some part of the hash tree in the cache (called as Cached Hash Tree) thus resulting in the performance improvement in terms of off-chip memory accesses. This also resulted in low bandwidth overhead over processor-memory bus. But since hashes are now being stored alongside data in caches, the increased cache pollution results in higher cache miss rate and thus lower performance. The second scheme for memory authentication and integrity verification as proposed by Aegis team is Log Hash scheme [2]. In cached hash trees memory blocks are verified for integrity at each memory operation (fine-grained integrity check) but in log-hashes the frequency of this operation is reduced which essentially results in integrity verification of a sequence of memory operations (coarse-grained integrity check). The concept of log hashes is based upon incremental multiset hashes [6]. Lie et al in [7] proved that incremental hashes are unable to prevent replay attacks. This means that cached hash tree is the only structure which provides robustly secure memory integrity verification. Although cached hash trees are more efficient compared to simple Merkle trees but they still have performance limitations which hinder their use in practical systems.

In this paper we describe a set of new schemes for efficient memory integrity verification. The basic motive behind these schemes is to exploit temporal locality in the memory accesses while authenticating off-chip memory. Our proposed scheme builds upon the idea of Hash Trees and makes them cater for memory access patterns to have improved temporal locality. Exploiting temporal locality essentially results in reduced cache miss rate, reduced memory bandwidth usage and higher performance.

Rest of the paper is organized as follows. In section 2, we describe related work. In section 3, we go on to propose some new schemes for efficient memory integrity verification and their relative advantages and disadvantages. The analytical performance evaluation of these schemes is described in section 4. In Section 5, we draw conclusions and lay out plans for future research work.

2 Related Work

2.1 Memory Integrity Verification in XOM

XOM (eXecute Only Memory) achieves software tamper proofing by compartmentalized execution of the applications where each application is executed in a unique secure compartment. The on-chip data is tagged with the compartment ID and can be accessed and modified by the process with that specific ID only. For data that goes off-chip, XOM calculates the hash of the data and concatenates it with the data before encrypting the whole block. Each compartment has a different encryption key. Thus whenever data is loaded from memory, it can be verified that it was indeed stored by the process with particular encryption key. This data is meant only for the process running in particular compartment, all other processes can't read this data. Moreover, the hash of the data verifies that the data is indeed not tampered with, thus preventing spoofing attacks [7]. Moreover, XOM combines the address of the data with its hash, collectively named as Addressed MAC (Message Authentication Code), which eliminates the possibility of splicing (reordering) attacks [7] where an adversary can copy an encrypted block from one location to another.

2.1.1 Replay attacks in XOM

Although XOM is able to prevent spoofing and splicing attacks but unfortunately it is unable to prevent replay attacks. A replay attack occurs when the data value loaded from the memory is not the one that was last stored there. For example a typical scenario for a replay attack can be put as follows. The processor writes a value, say "x", to a memory location and then updates it to a new value "y" after some time. We say a replay attack has occurred if the data value fetched from that memory location is "x" instead of "y". This is possible to happen as an adversary can monitor processor memory traffic for a particular memory location, can capture a value and place it to that location each and every time the value there gets updated. This essentially replays the old value in that memory location. It has widely been recognized that the replay attacks can happen in XOM's addressed MAC scheme [1,10,7]. The basic problem with the addressed MACs is that although they ensure that the value read from a memory location is indeed the value written by the current process and is untampered, but they fail to ensure that the value read is actually the last value written there.

2.2 Memory Authentication and Integrity Verification in Aegis

Aegis architecture has recently been proposed [3] and is derived from XOM. The Aegis architecture tries to provide memory integrity verification which can prevent replay attacks, while still attempting to limit the performance penalties of memory authentication and encryption functions. Since, in this paper, we are concerned only with memory authentication functions, we will limit ourselves to the memory authentication and integrity verification mechanism of the Aegis. Interested reader is referred to [3] for the complete details of the Aegis architecture.

Before explaining the Aegis schemes for memory integrity verification, a few terms and assumptions that will be used extensively during that explanation and throughout the rest of the paper are as follows.

- The on-chip caches and registers are completely secure against tampering attacks.
- The processor has two levels of the cache, L1 and L2. L2 Cache is on-chip and stores both instructions and data.
- The term chunk is used for the minimum memory block that is verified by integrity checking operation.
- In a simple implementation, a chunk is the same as L2 cache block.

2.2.1 Hash Trees and Cached Hash Trees

Learning from the failure of addressed MACs in preventing replay attacks in XOM, the Aegis team first of all came up with the idea of using Merkle Trees, also known as hash trees, for memory authentication. Merkle Trees were proposed as a means to update and validate data hashes efficiently using a tree of hashes calculated over the entire data set [8]. In a Merkle tree data is located at the leaves of the tree. Each inner node in the tree contains the collision resistant hash of the data contained in all of its children collectively. The authentication scheme works as follows. Each time a data value is fetched from memory, all of its siblings in the tree representation are fetched alongside. The node's data value is concatenated with its siblings' values and a hash over the combined data is calculated. This hash is then matched with that of the parent if it matches then the same procedure is repeated recursively till the root of the tree. Root is always contained in the trusted storage with in the processor and therefore can't be tampered with. If during the authentication process, the calculated hash value does not match with the stored hash value in a particular inner node of the tree then a security exception is raised and program is halted. The hash update function works in a similar fashion. Each time a value is being updated in memory, the new value is concatenated with the values of the siblings in tree representation and a hash is calculated over the combined data, the parent's hash is updated with this new hash and this process continues till the root node. With a balanced m-ary tree, the number of chunks to check on each memory access is $\log_m$ (N) where N is the total number of chunks in the process memory space. Moreover, hash tree scheme has a constant memory consumption overhead of m/m-1. This implies that for a 4-ary tree one quarter of memory ends up being used by hashes.

This scheme based upon the addressed hashes of the data values prevents all three attacks i-e spoofing, splicing and replay attacks [3]. But a simple implementation of Merkle Tree suffers from severe performance penalties. For example, since for each data value being fetched, its siblings are also required, we end up paying a large cost in terms of memory accesses for bringing its siblings into the processor. Moreover, as we have to perform the validity checks till the root of the tree, the memory access cost for fetching higher levels of the tree makes this scheme too expensive to be practically useable. A little insight into the hash tree mechanism reveals that a careful implementation of hash trees can reduce the performance penalties considerably. Consider for example, if the arity of the tree is limited by the number of data values that can be stored in one chunk and the similar considerations are given to storing of the hash values at inner nodes. Then this will ensure that each time we bring a memory chunk, which is the same as the L2 cache block, we have all the nodes available in one chunk whose data values are to be concatenated to get a combined hash to be verified against the value of the parent hash. This is the whole idea behind Cached Hash Trees, the second scheme proposed by Aegis team. A binary cached Hash Tee is illustrated in Figure1. Cached hash trees improve performance of the memory integrity verification function by essentially reducing the cache miss rate and reducing the memory bandwidth consumption. Another advantage associated with the caches hash trees is that in the authentication process, while traversing the tree towards the root, if a node is found already in the on-chip cache, the authentication operation is stopped there indicating the node under test is authenticated. This is true because every node which gets authenticated and is still in the cache means the nodes in the levels upper than the level of this node have already verified this node and that there is no need to continue with the integrity check operation. As no one cam tamper data in on-chip caches. The cached hash tree scheme is illustrated in figure 1. This advantage with cached hash trees, if properly exploited, can lead to significant drop in memory authentication latency. We will show how we make use of this phenomenon in our proposed scheme in the sections to come.

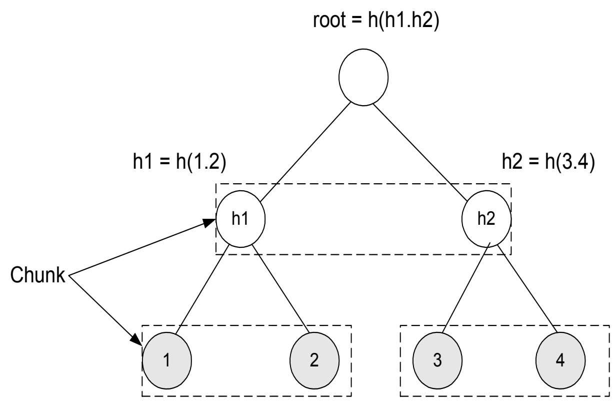

Figure1. Figure shows a binary hash tree. Each internal node is a hash of its children [1].

2.2.2 Log hash Integrity Checking

Log hash integrity checking scheme was proposed by Aegis team to reduce the memory integrity verification latency further down compared to cache hash trees. Log hashes are based upon incremental multiset hash functions to reduce the hash storage requirements. The main idea was to authenticate a *sequence* of memory operations as opposed to individual memory operations as in the case of cached hash trees. Two log hashes are maintained, Read hash and Write Hash along with a counter in trusted on-chip storage. At initialization all the memory chunks that need to be verified are brought into memory and the write hash is updated with the address- chunk- time stamp triple. At run time, each read operation leads to update of the read hash with the address- chunk- time stamp triple of the chunk being read and similarly write hash is updated on the write operation. After a sufficient number of memory operations, integrity check operation is performed. During the integrity check operation, all chunks that are not in the cache are read thus updating read hash. This essentially makes sure that all the chunks have been added to the read and write hashes same number of times. At the end of this, if read hash is equal to the write hash then the sequence of memory operations is assumed to be authentic otherwise it is not and a security exception is raised. Although log hash based integrity verification scheme significantly improves the efficiency of the memory authentication operation but it has been recently proved by Lie et al in [7], that incremental hash function, upon which log hashes are based, can't prevent replay attacks.

Problem with the incremental hashes is essentially that when we read in the old value to remove it we do not actually verify that the value is the correct value. The adversary can exploit this weakness to replay a value in memory.

Next section describes the proposed schemes for efficient memory integrity verification.

3 Proposed Approaches for Memory Integrity verification

In the wake of above discussion, one can easily conclude that only cached hash trees provide robust security as for as off-chip memory integrity verification is concerned. But the performance penalties associated with this scheme renders it less useful in practical scenarios. We believe that modifying hash trees so that they can make use of memory access patterns can reduce these performance penalties considerably. We have seen in the previous discussion that cached hash trees reduce the performance penalty if during the authentication operation we can find an authenticating node in the cache. This authenticating node's proximity with the leaves of the tree dictates the amount of performance improvement; the closer it is to the leaves the better is the performance. Based upon this observation, if we arrange leaves of the tree such that two temporally local nodes are siblings to each other then, after one node has been authenticated, the probability that during the authentication process of the other node, an authenticating node is found in lower levels of the tree (close to the leaves) is much higher than the cached hash tree scheme. The next couple of sections describe the two proposed schemes. First one is more ambitious in asking for compiler help as it requires almost exact memory access pattern to be provided. The second one requires only the estimated frequency of access or probability of access of each chunk to build an authenticating structure. The authenticating structure in case of Scheme 1 is an undirected graph while for the second scheme it is rooted tree.

3.1 Graph Based Memory Integrity Verification Scheme

The proposed structure for first memory integrity verification scheme is based upon undirected Graph. The choice of graph stems from the fact that the memory access patterns themselves are a graph and modeling the authentication structure as a tree built from graph is much easier than to build the tree directly.

The proposed graph based authentication mechanism works as follows. When a process is loaded for execution, all of its cache blocks are brought into processor. Based upon the access pattern, an undirected graph is built over all these memory accesses. The proposed architecture provides three primitives to build the authenticating structure. These primitives are `build_graph`, `prune_graph` and `calculate_hash`. The `build_graph` function builds the adjacency list for the undirected graph based upon the memory access pattern. This adjacency list is implemented in two parts. First is the vertices array. Each element in the vertices array has two parts, one contains the address of the memory chunk being referred to by this vertex and the other contains the address (pointer) to the array of neighbors of the vertex. Each element in the neighbors array contains the address of the respective neighbor of this vertex. This is in agreement with the conventional adjacency list representation of the graph. The `prune_graph` function makes sure that the neighbors list contains only those neighbors of the vertex which have not been added to the neighbors list of any other vertex i-e each vertex can be a neighbor to only one vertex. This essentially removes the possibility of a circuit and we are left with the tree representation of the graph. Moreover those vertices which have number of distinct neighbors less than a certain threshold, are removed from the vertices array during the `prune_graph` operation. Now after the `prune_graph` function is complete, `calculate_hash` function is used to calculate the hashes. This function is performed as follows. A new array is allocated for storing hashes. There is a hash for each entry in the vertices array. This hash is calculated over combined data of the vertex with its neighbors (vertices in the neighbors array). Now since authentication of any vertex will require hash to be calculated over the combined data of this vertex with all of its neighbors, there is a limit to the maximum number of neighbors that a vertex can have. This limit is imposed as MAX_THRESHOLD and is taken care of during `build_graph` function. Note that we need to have one vertex that corresponds to root node in the Merkle tee and which is stored on-chip. This is represented as source vertex in this scheme. The source vertex contains the combined hash of the hashes in the hash array. In most of the cases, the number of such hashes will be quite large and combining them all would not be a good idea. In that case, a tree can be induced on these hashes where source vertex plays the role of root. But we predict that there will be few occasions where we have to traverse through the whole structure to authenticate a chunk. This will become clear in the illustrations for the analytical analysis for this scheme. The `build_graph`, `prune_graph` and `calculate_hash` functions are formally given below.

In the given functions N refers to the number of chunks to be verified. MAX_THRESHOLD refers to the maximum number of neighbors that a vertex can have.

```
Function
build_graph(mem_access_sequence [])
    Graph mem_access_graph;
    mem_access_graph =
build_adjacency_list[mem_access_seq
uence];
end function
```

```
Function prune_graph (graph
mem_access_graph)
    /* traverse through the
neighbors list of all vertices and
remove the redundant entries
leaving only one which is
encountered first. This procedure
can be implemented as BFS (Breadth
First Search) where cross edges
(entries in the neighbors array
corresponding to those edges) are
explicitly deleted from the graph
leaving us with a BFS tree. We call
this BFS implementation as
BFS_prune. Moreover, during the
BFS_prune operation, vertices which
have neighbors less than a certain
MIN_THRESHOLD are marked for
deletion and are removed in the
subsequent operation.*/
    BFS_prune(graph
mem_access_graph, MIN_THRESHOLD)

mem_access_graph.remove(vertices_ma
rked_for_deletion);
End function
```

```
function calculate_hash
/*concatenate data of each vertex
in the neighbor list of this vertex
with the data of this vertex and
calculate the hash of the combined
data and store it in the hash list
indexed by this vertex.*/
   hash (vertex) = hash
(vertex.data +
getCombinedData(vertex.neighbors))
end function
```

All these functions are better explained through illustrations for a sample memory access pattern as shown below.

Consider a program shows following access pattern of chunks.

1, 2, 3, 2, 1, 4, 5, 3

The graph built over this access pattern is shown below.

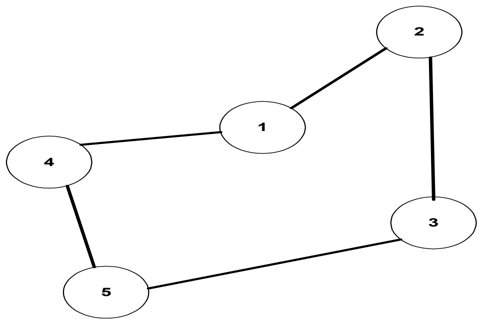

Figure3. Graph for given memory access pattern of 1,2,3,2,1,4,5,3

The adjacency list, after the `build_graph` function is performed, looks like as shown in figure 4.

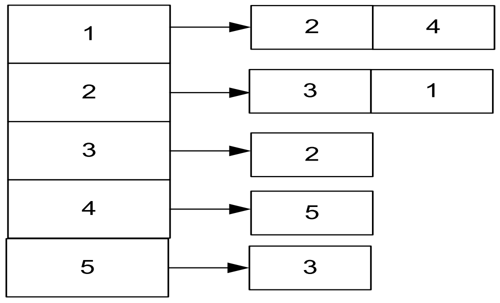

Figure 4.Adjacency list representation of the graph in figure 3.

After the prune_graph operation is complete, the graph essentially becomes a tree as shown in figure 5.

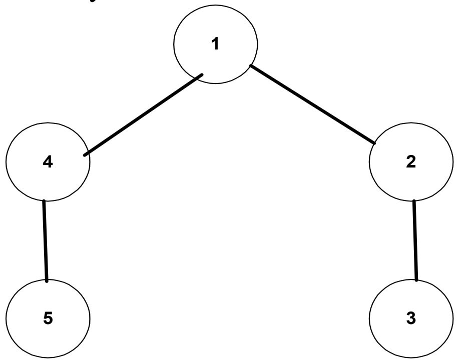

Figure 5.The graph after `prune_graph` operation.

The neighbor list representation of the pruned graph is given in figure 6.

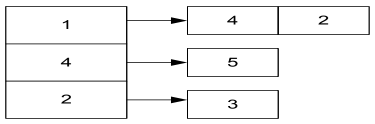

Figure 6.The neighbor list representation of the graph in figure 5

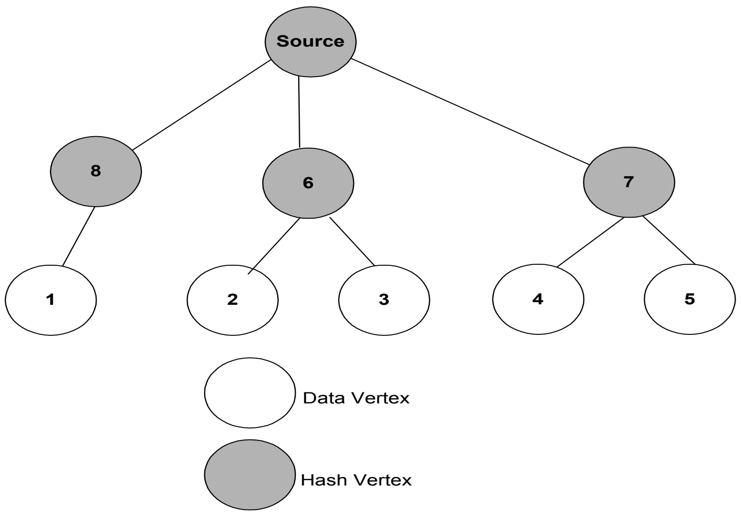

Figure7. The graph after `calculate_hash` function is performed

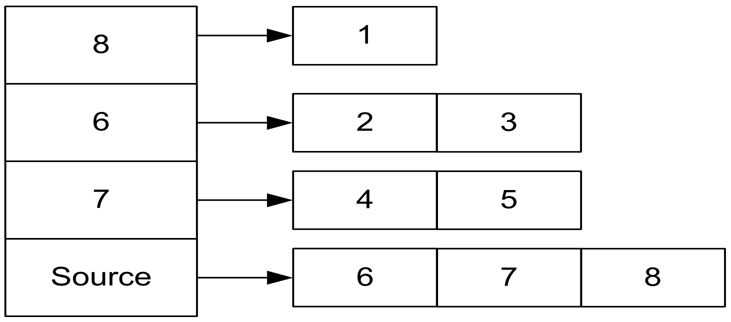

Figure8. The neighbors list representation of the graph in figure 7

After the prune_graph operation is complete, next step is to calculate the hashes and store them in separate chunks. Each hash chunk contains hash computed over the combined data of a vertex (in the reduced vertices array) and its neighbors. After the calculate_hash operation, the graph for the sample memory access pattern looks like as shown in figure 7.

This scheme has obvious advantage over cached hash tress as it caters for the temporal locality in the memory access patterns much more effectively. But this performance improvement comes with cost. Firstly, this scheme requires adequately accurate memory access pattern from compiler. This access pattern can be inferred using static profiling or other compiler techniques. Which may be a time consuming process given the state of the art of compiler research.

Secondly it needs extra amount of trusted storage space to store the graph structure. The graph structure is stored in an on-chip dedicated storage space. Note that we can use L2 cache for storage as well. But in that case we will have to authenticate the structure itself once it is evacuated to memory. Although MACs can be used for authenticating the structure as it is essentially a static data after `calculate_hash` function is completed, but true performance benefits are achieved if the structure is stored in a dedicated storage space. Dedicated storage space for the structure will reduce the cache pollution as well as the processor memory bandwidth consumption. Above all the structure lookup time will be greatly minimized if it is stored on-chip. Moreover, every effort is made to reduce the size of the pointers in the structure. This is why vertices are addressed by their offset from a reference address. This reference address can be the address of the very first instruction in the secure programme. This instruction is usually the one which asks processor to run in trusted mode like for example `enter-tee` instruction in Aegis.

The requirements on the information required from the compiler can be relaxed by asking for estimated reference count or probability of access of a chunk as opposed to the exact access pattern. This idea is the base of the second scheme that we propose and is explained in the next section.

3.2 Reference Count or Probability of Access based Scheme

In this scheme, an authentication tree structure is built which tries to exploit the temporal locality based upon the frequency of references or probability of access. The reference count information is similar in nature to the probability of access information which is already being provided by compilers on various architectures. This scheme does not require a lot from the compiler yet has improved performance as compared to cached hash trees. This will become clear when we give analytical comparisons in the next section. Another advantage of this scheme compared to graph based scheme is the reduced computation complexity. Two primitives are provided as part of this scheme namely `build_tree` and `calculate_hash`. The `build_tree` function builds a tree from the given reference counts of chunks such that each level of the tree corresponds to a specific set of reference counts. The mapping of reference counts on tree levels again is dependent upon the number of nodes in the level. We obviously don't want to clutter a level with too many nodes neither we want to have too few of nodes at each level such that the overall height of the tree becomes large to that extent that performance starts degrading. The `calculate_hash` function works the same way as described above in context of first scheme.

The illustration of this scheme for the same sample memory access pattern as given above is given as follows.

Given reference pattern 1,2,3,2,1,4,5,3

Reference counts for each chunk. 1(2),2(2),3(2),4(1),5(1)

Figure9 gives the illustration of `build_tree` operation while figure

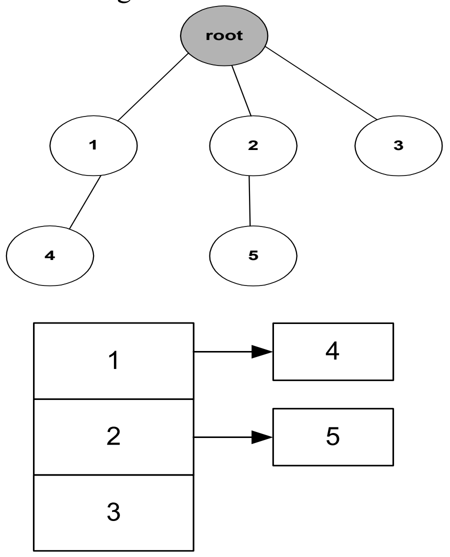

Figure9. The `build_tree` operation on the given memory access pattern. Figure illustrates Graphical and structural representations of the resulting tree.

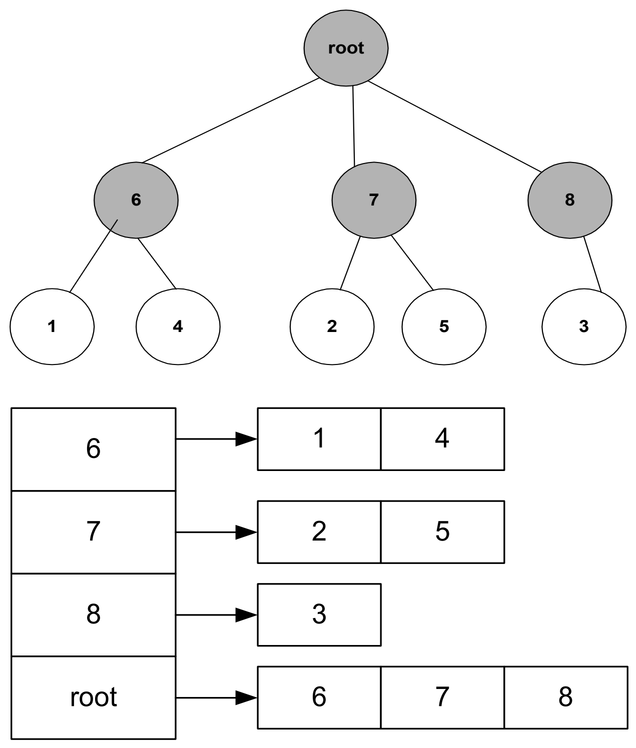

Figure10. The `calculate_hash` function on the given memory access pattern and the resulting graphical and structural representations.

3.3 Proof of Security of the Proposed Schemes

As is obvious from the above discussion, the proposed schemes are based upon hash trees in their essence which have been proved to be secure against all kinds of adversarial attacks [1,7]. Thus we can imply that the security of the proposed schemes is also established. For formal proofs of security of hash trees, interested reader is referred to [8].

4 Performance Evaluation

This section describes the analytical comparison between the Cached Hash trees and the two schemes proposed in the previous section. The comparison is being done on the basis of cache miss rate which also effects memory bandwidth consumption and is a decisive factor in estimating performance. The underlying model is assumed to be containing a cache of size 3 blocks. This is intentionally kept small to ease the process of analysis. In the analytical model, LRU (Least recently Used) based replacement scheme is employed. Moreover, the structure for the proposed schemes is assumed to be stored in the on-chip dedicated storage. A little modification to the replacement scheme in case of proposed schemes is that the evicted block is matched against the neighbors' list of the node being authenticated. If it is required for authenticating the node then the block is stored on-chip in a victim cache like structure. Since there is no structure available for cached hash trees, we can't have this implementation for them.

We start by giving the cache misses for the cached hash tree scheme for the given memory access pattern. The cached hash tree looks like figure 11 for this access pattern. The status of cache at each access is shown in figure 12. The red blocks indicate a miss.

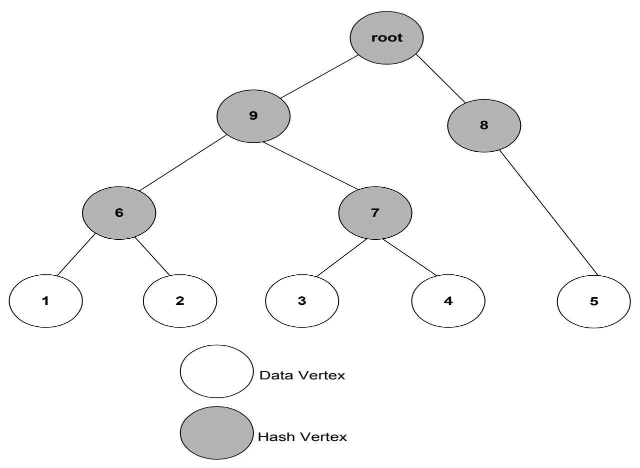

Figure11. The cached hash tree for the given memory access pattern.

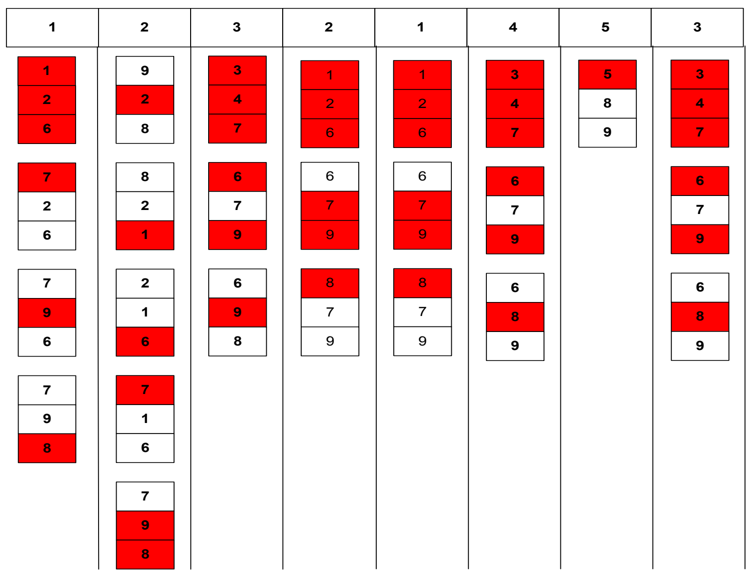

Figure12. Figure shows the L2 cache status at each memory reference in cached hash trees. Total cache misses=43

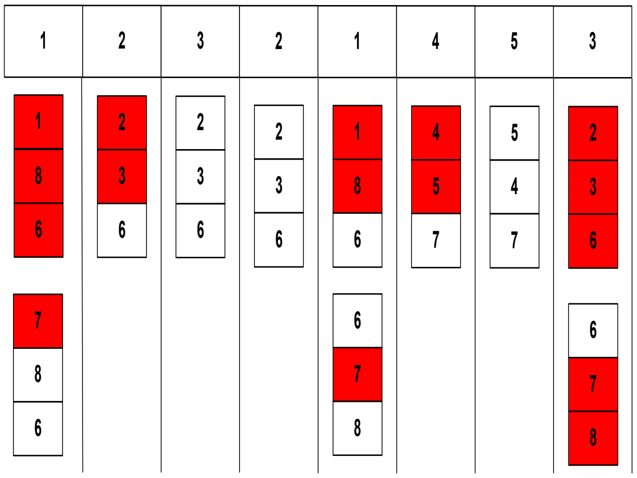

Figure13. L2 Cache status at each memory access in case of graph based proposed scheme. Total Cache misses = 16

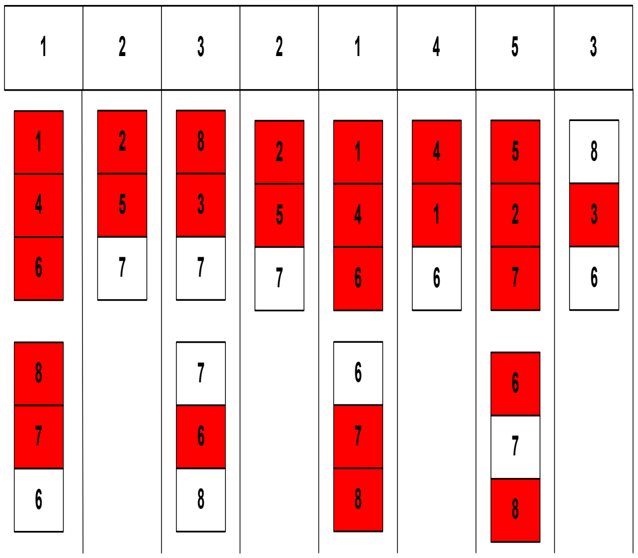

Figure14. L2 Cache status at each memory access in case of proposed reference count/probability of access based scheme. Total cache misses = 25

As is clear from the illustration in figs 11, 12 and 13, the number of L2 misses in cached hash tree scheme is 43, compared to 16 and 25 for proposed graph based and reference count based schemes respectively. The graph based scheme, which requires accurate information regarding memory access pattern, has 169% improvement, in terms of cache misses, over cached hash trees. On the other hand the reference count or probability of access based scheme has an improvement of 79% over the cached hash trees.

5.1 Conclusions

The analytical evaluation of the performance for the integrity verification schemes reveal that the proposed graph based memory integrity verification schemes outperforms cached hash trees by a big margin. But it has drawbacks of requiring extensive compiler support and paying relatively heavy computation cost for building the required structure. Although this computation of the structure is a one time operation only but it reduces the overall performance gain to some extent. On the other hand the reference count based scheme has reasonable performance improvement over cached hash trees and requires little extra support from compiler. Moreover its structure computation cost is also less compared to graph based approach. Therefore, we recommend using Reference Count or probability of access based approach in the shorter term till the compilers are mature enough to give us the information regarding access patterns required by the graph based scheme.

5.2 Future Research

In this paper we have focused on improving hash tree mechanism for efficient memory integrity verification by employing certain information regarding memory access patterns from the compiler. This is essential in order to exploit temporal locality in memory accesses which leads to significant performance improvement. In future, we plan to extend compiler research in this area so that we get the required information as accurately as possible.

5.3 Acknowledgements

This research work was done as part of ECE594 course titled" Special Issues in Microprocessor Design" under Prof. Gyungho Lee at ECE UIC. I would like to acknowledge Prof. Lee for his consistent feedback and encouragement on the work. I would also like to thank Prof. Ashfaq Khokhar, my PhD Advisor, for giving me enough time and opportunity to work on this topic.

6 References

1. B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. "Caches and merkle trees for efficient memory integrity verification". In Proceedings of Ninth International Symposium on High Performance Computer Architecture, February 2003.

2. G.Edward Suh, Dwaine Clark, Blaise Gassend, Marten van Dijk, Srinivas Devadas, “Efficient Memory Integrity Verification and Encryption for Secure Processors”. Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
3. G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. “AEGIS: Architecture for tamper-evident and tamper-resistant processing”. In Proceedings of the 17th Int'l Conference on Supercomputing, June 2003.
4. A. Carroll, M. Juarez, J. Polk, and T. Leininger.Microsoft \Palladium": A Business Overview. In Microsoft Content Security Business Unit, August 2002.
5. D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. “Architectural Support for Copy and Tamper Resistant Software”. In Proceedings of the 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pages 169{177, November 2000.
6. D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. “Incremental multiset hash functions and their application to memory integrity checking”. In Technical Report MIT-LCS-TR-899, May 2003.
7. David Lie, John Mitchell, Chandramohan, A. Thekkath, Mark Horowitz, “Specifying and verifying hardware for tamper resistant software”, In the Proceedings of the 2003 IEEE Symposium on Security and Privacy. May, 2003.
8. Ralph C Merkle, “Protocols for public key cryptography”, In IEEE Symposium on Security and Privacy, pages 122-134, 1980
9. I. Foster, C. Kesselman, S. Tuecke, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations” International J. Supercomputer Applications, 15(3), 2001.
10. Jun Yang, Youtao Zhang, and Lan Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering", Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, Pages 351-360, San Diego, California, December 2003.