Towards Fast Hardware Memory Integrity Checking with Skewed Merkle Trees

Jakub Szefer
Computer Architecture and Security Laboratory
Yale University
jakub.szefer@yale.edu

Sebastian Biedermann
Security Engineering Group
TU Darmstadt
biedermann@seceng.informatik.tu-darmstadt.de

ABSTRACT

Protection of a computer's memory's integrity is crucial in situations where physical attacks on the computer system are a threat. Such attacks can happen during physical break in into a data center or when a mobile device is lost or stolen. Since the memory modules can be easily removed or manipulated, the integrity of their contents cannot be trusted under threat of physical attacks. To counter this, hardware memory integrity checking schemes have been proposed, and realized in a number of security microprocessor architectures. At the core of these schemes is usually some form of a Merkle tree. All previous work on security architectures, however, uses full, balanced Merkle trees. In this paper, we propose a new solution to hardware memory integrity checking based on skewed Merkel trees. Because not all memory locations are accessed equally frequently in a modern computer system, a skewed Merkle three offers better performance as the frequently accessed memory locations can be located on the leaves of the skewed Merkle tree that have shorter path to the root – thus fewer nodes of the tree have to be accessed during integrity checks. Skewed Merkle trees offer better system performance when considering realistic memory access patterns where some page are accessed more frequently than others, they do not impact caches as much as full Merkle trees, and they do not require more storage than full, balanced Merkle trees.

Keywords

Hardware Memory Integrity; Merkle Trees; Skewed Merkle Tree

1. INTRODUCTION

When a threat of physical attacks on memory modules in a computer system is a possibility, then the confidentiality, integrity and availability of the DRAM memory contents needs to be protected. In this work, we focus on the integrity property. To protect integrity of DRAM memory's contents, various integrity protection schemes have been proposed. Most common approaches are based on Merkle trees. A Merkle tree is a tree data structure where the leaves contain the items which need to be checked for integrity (e.g. memory pages of DRAM). Each interior node's value is generated using a cryptographic hash over contents of its children nodes (or in case of last level of nodes, the contents of the leaves). Through this construction, the root node of the Merkle tree contains a hash of the entire set of leaf nodes (e.g. all the memory pages of DRAM). If the root node is stored securely, such as in a special on-chip register, then the tree nodes can be stored in the insecure DRAM memory (details about Merkle trees and certain caveats, e.g. use of nonces for freshness, are discussed in Section 2).

If a secure microprocessor architecture uses Merkle trees for memory integrity checking, then special hardware is introduced inside the microprocessor chip to realize the Merkle tree and to walk the nodes of the tree on memory accesses to verify that the DRAM contents has not been maliciously modified. The root node is securely stored in the microprocessor register and can always be trusted. If a Merkle tree was not used, then on each memory access the whole memory would have to be read, hashed and the hash value compared to the one stored securely on-chip. With Merle tree, only the nodes (and their sibling nodes) on the path for the leaf (which represents the particular memory page) to the root need to be read and hashes compared at each level, until the root is checked. Finally, the memory access can proceed. These architectures can protect against realistic attacks, such attacks on memory contents have been previously done in the real world, for example the Windows 7 password injection attacks using Firewire [1].

Problem Overview: The various hardware memory integrity checking designs which have been previously proposed use full, balanced Merkle trees. Even when certain optimizations are introduced, such as Bonsai Merkle trees [16], to significantly improve performance, still, at the heart of the scheme is a full, balanced Merkle tree. With a full, balanced tree, each path from a leaf node to the root is of the same length. This means, that each memory access requires same amount of work to check the integrity of the memory; even if cached trees are used [11].

In a modern computer system, however, not all memory pages are made equal: some memory pages are accessed more often than others. Thus, the efficiency of the memory integrity checking schemes can be improved if paths from the leaf nodes representing the frequently accessed memory pages could be shorter. We propose that this can be realized through the use of a skewed Merkle trees.

HASP '14, June 15, 2014, Minneapolis, MN, USA.

Our insight to use skewed Merkle trees for integrity verification is based on observations, such as those shown in Figure 1. The graph shows frequency of physical memory address accessed during the first 1.4×10^9 cycles of boot up and runtime of an ARM-based system with 256MB of DRAM. It can be clearly seen that the addresses at low physical memory and high physical memory are accessed disproportionately frequently.

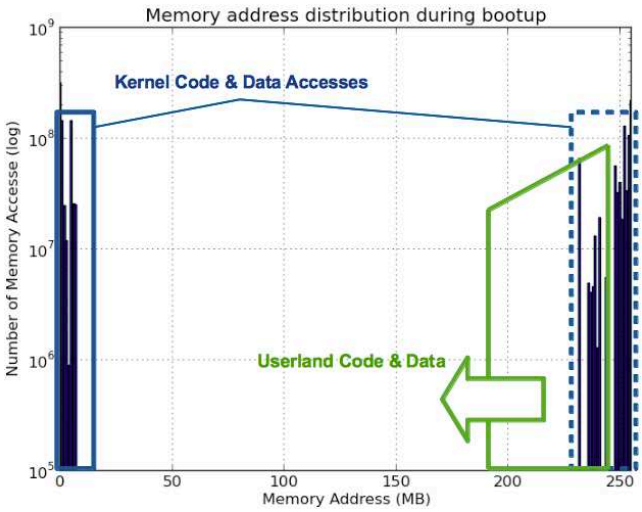

Figure 1: Distribution of accessed to physical memory, during first 1.4×10^9 cycles of boot up and runtime of an ARM-based system with 256MB of DRAM. OS related accesses are much more frequent and located at the low and high ends of the physical memory. Userland accesses are more in-between, also, as more applications are started, more accessed will be in the middle address ranges as indicated by the arrow in the figure.

On the bottom range of physical address are the logical Linux kernel addresses (used during early boot) and on the top are the Linux kernel virtual addresses [6]. The userland (virtual) memory is usually mapped to the physical addresses in between the two ends, so as more applications are started after boot up, more accessed will be made to memory and the graph will grow inwards form the high-memory side. Again, note this graph shows the physical memory of a Linux system, as this is what memory integrity checking is concerned with.

Solution Outline: We propose to use skewed Merkle trees as basis for hardware memory integrity checking. Section 3 gives details of the solution and the skewed Merkle tree structure and we evaluate design in Section 4. In summary, we design a skewed Merkle tree which has shorter paths corresponding to low and high physical memory address ranges, while the paths for less frequently accessed middle pages are extended. This results in a design that requires accessing fewer internal tree nodes for verification as the system runs. Moreover, the total memory overhead of the skewed trees is the same as for a full, balanced tree, thus not adding any storage overhead. These improvements can be applied to the variety of other proposals for different schemes based on Merkle trees, as discussed in the section on research challenges and opportunities, Section 5.

Paper Organization: Section 2 provides background on hardware memory integrity checking. Section 3 presents the new skewed Merkle tree approach to memory integrity checking. Section 4 presents results of the evaluation. Section 5 lists interesting research challenges and opportunities that skewed Merkle tree design open. Related work is described in Section 6 and the paper concludes in Section 7.

2. HARDWARE MEMORY INTEGRITY CHECKING

This section presents background on Merkle trees and their use in hardware memory integrity checking. Note that all proposals we are familiar with are based on full, balanced Merkle trees. Next section, Section 3 presents details of a new approach to hardware memory integrity checking: skewed Merkle trees. Related work section, Section 6 lists in detail various types of previously proposed trees and optimizations of trees that have been proposed or implemented in different security architectures.

2.1 Memory Integirty Checking

Integrity checking is crucial when the integrity of the contents of a computer system's (DRAM) memory cannot be trusted. This is true whenever physical attacks are considered. Today, as much personal data is stored on mobile devices which can be easily lost or stolen, physical security of the memory is very difficult to guarantee. To tackle this threat, also present in server setting, not just mobile devices, secure architectures such as AEGIS [20] or Bastion [5] have been proposed which incorporate memory integrity checking. Figure 2 shows a high-level diagram of a computer system and the trusted and untrusted parts in threat models such as those considered by AEGIS, Bastion or similar designs. In particular, the memory is not trusted. This is the DRAM memory that holds the code and data, as well as, it can hold values used to help integrity checks (e.g. internal nodes of a Merkle tree). What is trusted, is the processor chip, on-chip caches and registers inside the processor which can hold values such as the root hash of a Merkle tree, as described next.

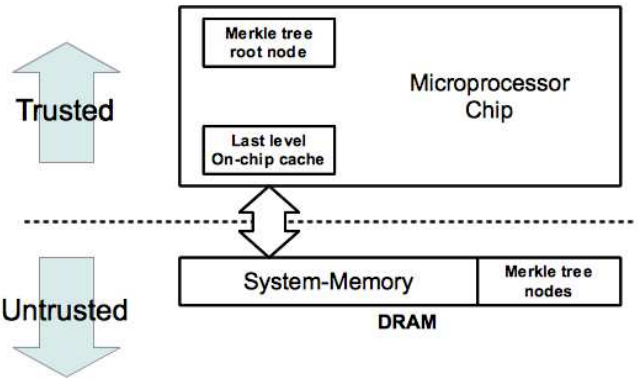

Figure 2: Trusted and untrusted components of a typical secure microprocessor design that incorporates Merkle trees for memory integrity checking.

2.2 Integrity Checking with Merkle Trees

Merkle trees are used as basis of most hardware memory authentication schemes. The leaves of a Merkle tree are notes which values need to be checked for integrity, i.e. these are the DRAM memory pages that need to be checked. Merkle trees use hash functions to generate values for the intermediate notes in the tree; for example in a 2-ary Merkle tree, given child nodes $Node1$ and $Node2$ values, a parent node's value is $hash(Node1||Node2)$ (where "||" represents concatenation). A sample integrity checking tree is shown in Figure 3. Using this construction, the root node of the tree is a hash value that depends on all the leaf nodes' values. If the root node is stored securely, then all the intermediate values need not be stored securely[1]. Consequently, as Figure 2 showed, the root node is most often stored in an on-chip register, while the other nodes are stored in memory.

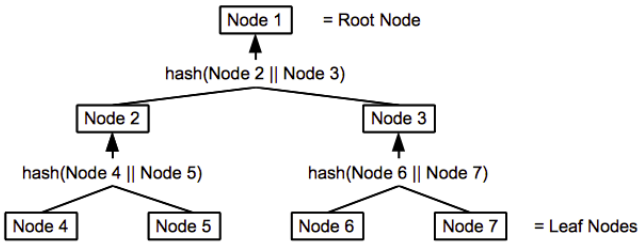

Figure 3: General model of a 2-ary Merkle tree used for integrity checking of 4 leaf nodes.

All memory integrity checking constructions, of which we are aware, that have been previously presented are based on a full, balanced Merkle tree; such sample tree is show in Figure 3. To validate the integrity of a leaf node, all the intermediate nodes on the path from the leaf node to the root node need to be consulted. Thus, the validation or checking depends directly on the number of levels of the tree. With a full, balanced tree, the number of tree levels N_L is defined by: $N_L = log_A(M)$, where A is the arity of the tree.

Often, the node size of the tree is selected such as to match the cache line size. E.g. SHA-256 algorithm generates hashes which are 32 Bytes in size, so that can be conveniently used with caches that have cache line size as a multiple of 32 Bytes. The leaf nodes are often assumed to be 4 KBytes in size to match the operating system's memory page size (although this can change in different operating systems). There are, however, many changes to this basic scheme and many optimizations that have been presented.

3. INTEGRITY CHECKING BASED ON SKEWED MERKLE TREES

In this section we present our proposal for using skewed Merkle trees for memory integrity checking. The key difference between regular Merkle tree and a skewed Merkle tree is that the paths from different leaf nodes to the root note have different lengths, depending on type of the skewed tree. Thus, checking integrity of the leaf nodes which are on the shorter path to the root takes less time and has lower overhead in terms of memory accesses. By carefully structuring a skewed Merkle tree, we can significantly improve the performance of the memory checking schemes.

[1]Note that to ensure freshness and prevent replay attacks, nonces are often used when generating the hash values of the nodes as well, this is not show for simplicity.

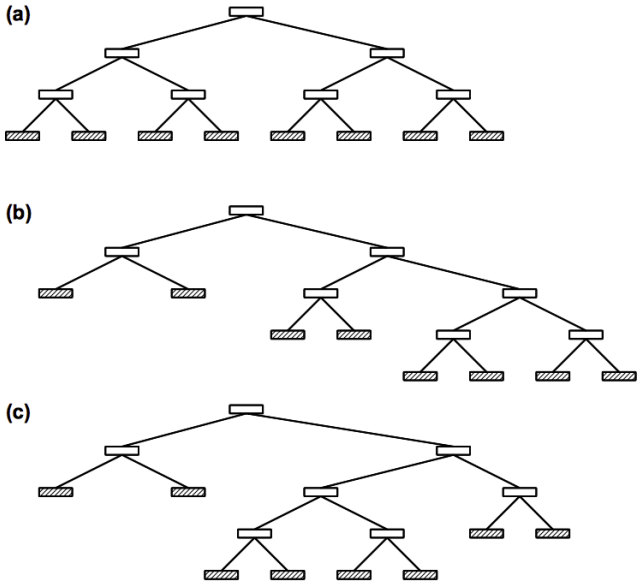

Figure 4: Sample Merkle trees showing distinction between (a) full, balanced Merkle tree, (b) right skew-by-one Merkle tree and (c) middle skewed-by-one Merkle tree.

The insight behind this work is that not all physical memory pages are made equal. Our evaluation of a modern computer system shows (c.f. Figure 1) that certain memory pages, especially those associated with the operating system, are accessed significantly more than others, on average. Also, these pages tend to be located at predictable addresses: low and high physical memory pages. Thus, by structuring the skewed Merkle tree such that these frequently accessed pages correspond to the leaf nodes with shorter path to the root, performance can be improved.

3.1 Types of Skewed Merkle Trees

Different designs for skewed Merkle trees are possible. Figure 4 (a) show a full, balanced tree, and two other designs. Figure 4 (b) shows a right skewed Merkle tree since nodes on the left side have shorter paths to root and going to the right, the paths get extended. Note, there can be a mirror, left skewed tree. Figure 4 (c) shows a middle skewed Merkle tree. In the middle skewed tree, just like in the right skewed tree, some of the nodes have shorter paths. These nodes, however, have been located on the both ends of the tree, and the longer paths are in the middle. Note that while Merkle tree has all paths from leaves to root of same length, with the right and middle skewed trees, some paths are shorter, while some are longer.

For 2-ary trees, such as those presented in Figure 4, if there are L leaves of a full tree, then for a skewed tree, $L/4$ nodes have path to the root shorter by 1 hop, $L/4$ have same path as for full tree and $L/2$ have a path that is 1 hop longer. Likewise, the shortest path can be recursively shortened by extending the long path. By careful structuring the tree to match the memory access patterns we observe in modern systems, we can gain the benefit of the shorter paths, without adversely being affected by the longer paths.

Similar skewed Merkle trees have been explored before in context of certificate revocation schemes [13, 10], for example. The key contributions of these works are on how to efficiently traverse different types of Merkle trees which are not a full, balanced tree. The insights from these works can be applied to the design of the secure processor hardware that would actually walk such a skewed tree and authenticate the memory. In this work, we focus on a simple scheme where the nodes are walked sequentially, but plan to incorporate the past work in future designs.

3.2 Tree Skewness

The skewed Merkle trees do not only have different shapes, but can have different levels of tree skewness. As shown in Figure 4, the tree in (c) is skewed-by-one compared to (a). I.e. the minimum length form leaf to root is one less than for a full tree. There are other skewness levels possible, such as skewed-by-two, etc. With each level, the shorter paths gets shorter, but the longest path also gets longer. Moreover, the are 1/2 as many nodes on the short path as there are on the long path, so skewing the tree too much can actually hurt the performance.

4. EVALUATION

The evaluation presented in this section was performed on an mobile device like configuration of ARM processor and Linux system. In particular, GEM5[2] simulator was used to obtain memory traces. Custom skewed Merkle tree simulation was written to simulate extra accessed due to the integrity checking and to generate updated memory traces. The DineroIV[3] cache simulator was then used to evaluate the overheads of the basic Merkle tree scheme and the proposed skewed Merkle tree scheme.

4.1 Simulation Configuration

Simulated Hardware Configuration: 32-bit ARM-based processor was simulated using GEM5 simulator. Details of the various configurations used are shown in Table 1. For each DRAM memory configuration, different cache configurations were simulated. (Due to limitation of the GEM5 simulator, only up to 256MB of DRAM was simulated.) Physical memory access traces were obtained from GEM5, and used to drive cache simulation.

For the cache simulations, the caches were set to use the least recently used (LRU) replacement policy. If an L2 cache was present, then it was simulated as a shared cache. Table 1 shows approximate architectures which each cache configuration represents. The cache configurations were varied slightly from the actual ones to introduce a variety of cache sizes and set sizes in the tests.

Software Configuration: The simulated system was used to boot up and run Linux 2.6.38.8, compiled for the ARM platform and the GEM5 simulator.

4.2 Impact on Memory Hierarchy

In this paper we focus on evaluating the impact of the full Merkle trees and the new skewed Merkle trees on the memory hierarchy. Since fetching of the nodes of the merle tree will pollute the caches, use of Merkle tree will degrade the performance and introduce new cache demand misses.

[2]http://www.gem5.org

[3]http://pages.cs.wisc.edu/~markhill/DineroIV/

Table 1: Simulated system configurations

Component	Details
CPU	32-bit ARMv7
DRAM	256MB
Caches	(ARM Cortex-A5 approx.) L1 Dcache: 32 KB, 32 B/line, 4-WAY L1 Icache: 32 KB, 32 B/line, 4-WAY L2 cache: none
	(ARM Cortex-A7 approx.) L1 Dcache: 64 KB, 32 B/line, 8-WAY L1 Icache: 64 KB, 32 B/line, 8-WAY L2 cache: 1 MB, 64 B/line, 16-WAY
	(ARM Cortex-A8 approx.) L1 Dcache: 32 KB, 32 B/line, 4-WAY L1 Icache: 64 KB, 32 B/line, 4-WAY L2 cache: 512 KB, 64 B/line, 8-WAY
	(ARM Cortex-A9 approx.) L1 Dcache: 32 KB, 64 B/line, 2-WAY L1 Icache: 32 KB, 64 B/line, 2-WAY L2 cache: 2 MB, 128 B/line, 8-WAY
	(ARM Cortex-A15 approx.) L1 Dcache: 32 KB, 64 B/line, 4-WAY L1 Icache: 32 KB, 64 B/line, 4-WAY L2 cache: 1 MB, 64 B/line, 16-WAY

We show, however, that with the use of skewed Merkle tree, this impact can be significantly reduced.

To evaluate the impact, we simulate boot up and runtime of the ARM system with Linux and collect physical memory access traces. It was already shown that the memory accesses tend to be quite localized, e.g. as was shown in Figure 1.

4.2.1 L1 Caches

Running cache simulation on the raw memory accesses, gives the cache behavior described in Table 2. This and following tables list demand misses. The L1 data cache misses are broken down to read misses, write misses and "tree" misses. The reads and writes are of data, and "tree" are access to the Merkle tree nodes. In Table 2 there is no tree in use, so last column is empty.

Table 2: Cache behavior during boot up and runtime without integrity checking overheads.

Config	L1 I	L1 D read	L1 D write	L1 D tree
A5	0.0153	0.0373	0.0423	n/a
A7	0.0086	0.0296	0.0386	n/a
A8	0.0092	0.0373	0.0423	n/a
A9	0.0114	0.0309	0.0237	n/a
A15	0.0106	0.0278	0.0230	n/a

Introduction of a full, balanced Merkle tree degrades the cache performance, as shown in Table 3. Instruction caches are not affected as the tree nodes are treated as data. Depending on the cache configuration, the L1 cache miss rate increased by $6.5x$ (reads) and $3.2x$ (writes) on average of the five configurations.

The use of a right skewed-by-one tree, gives cache performance, as shown in Tables 4. The L1 cache miss rate now

Table 3: Cache behavior during boot up and runtime with a full Merkle tree in use.

Config	L1 I	L1 D read	L1 D write	L1 D tree
A5	0.0153	0.2176	0.1102	0.064
A7	0.0086	0.1523	0.0773	0.0329
A8	0.0092	0.2176	0.1102	0.064
A9	0.0114	0.2595	0.1204	0.0664
A15	0.0106	0.2095	0.0815	0.0491

increases $5.6x$ (reads) and $2.7x$ (writes) on average of the five configuration over baseline in Table 3.

Table 4: Cache behavior during boot up and runtime with a right skewed-by-one Merkle tree in use.

Config	L1 I	L1 D read	L1 D write	L1 D tree
A5	0.0153	0.1867	0.0948	0.0364
A7	0.0086	0.1162	0.0642	0.015
A8	0.0092	0.1867	0.0948	0.0364
A9	0.0114	0.2298	0.1030	0.065
A15	0.0106	0.1861	0.0719	0.0385

The use of a middle skewed-by-one tree, gives cache performance, as shown in Tables 5. The L1 cache miss rate is further improved, increasing only $4.8x$ (reads) and $2.3x$ (writes) on average over the five configurations, over the base with no Merkle tree.

Table 5: Cache behavior during boot up and runtime with a middle skewed-by-one Merkle tree in use.

Config	L1 I	L1 D read	L1 D write	L1 D tree
A5	0.0153	0.1636	0.0817	0.0217
A7	0.0086	0.0900	0.0543	0.0036
A8	0.0092	0.1636	0.0817	0.0217
A9	0.0114	0.2051	0.0894	0.0554
A15	0.0106	0.1652	0.0620	0.0286

We also characterize the extra memory fetches due to the use of the different styles of Merkle trees. Figure 5 shows the extra L1 data fetches (recall the Merkle tree nodes are treated as data accesses in our simulation). It can be seen that the right skewed Merkle tree is actually worst than the regular full tree. On the other hand, use of the middle skewed tree gives about 1.3×10^9 fewer accesses.

4.2.2 L2 Caches

Our L2 cache evaluation focuses on the extra misses induced through the use of the different types of Merkle trees. Figure 6 shows the L2 cache demand misses. The fours bars correspond to four tested configurations which had L2 cache (recall A5 does not have L2 cache). It can be noticed that the number of misses is related to the cache configuration,

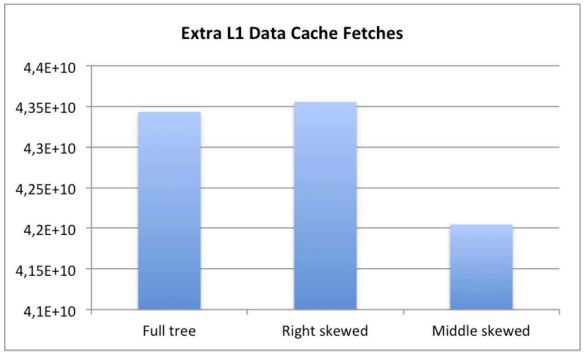

Figure 5: Extra fetches due to different Merkle tree configurations.

e.g. A8 always has the highest number of misses. Furthermore, pattern similar to one observed in L1 caches emerges. The right skewed tree actually adds more misses than the full tree. On the other hand, the middle skewed tree is better (fewer cache misses) than both the full tree and the right skewed tree.

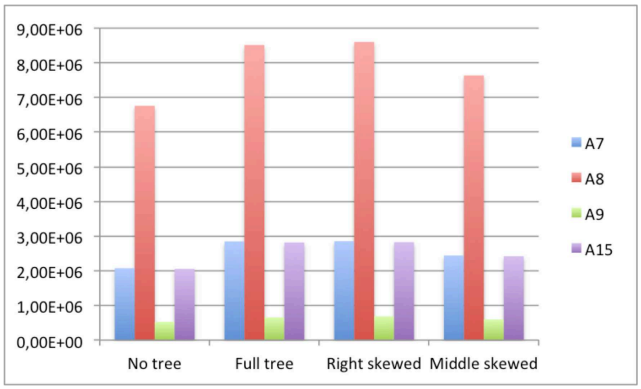

Figure 6: L2 misses for different simulated configurations of Merkle tree.

4.3 Evaluation Summary

Based on our results, it can be seen that the type of the skewed Merkle tree has large impact on the memory hierarchy performance. The middle skewed tree beats both the full tree and the right skewed tree. In all cases of using a Merkle tree there is a significant overhead. We are able to, however, cut down the overhead from $6.5x$ to $4.8x$ which is a 27% improvement. Nevertheless, further work is needed, e.g. combine middle skewed Merkle trees with cached hash trees proposals, to reduce the number of misses and miss rates even further.

5. RESEARCH CHALLENGES AND OPPORTUNITIES

This work is only a first step in what we have will be believe to be a new and potentially very fruitful research area on efficient hardware memory integrity checking using skewed trees. There are many open challenges, and opportunities which they bring.

5.1 Skewed Tree Structure

The exact structure of a skewed tree is an important research problem. We have presented two possible structures, the right skewed tree and the middle skewed tree. Within these structures, we presented the skewness level (i.e. skewed-by-one, skewed-by-two, etc.). Other structures, however, are also possible. In the extreme, there could be trees which have different path lengths from the root to different leaf nodes, not necessarily related to the position of the leaf node in the tree. For example, a random-like tree could be constructed.

The structure of the tree depends on in-depth knowledge of the physical memory access patterns. More experiments (and on different types of systems) are needed to understand how different memory regions are accessed. Such understanding will allow one to generate an optimal tree for a specific setup.

5.2 Tree Traversal

Once a tree design is selected, then one needs to consider the tree traversal problem. There is a choice of how and where to store the tree nodes in memory. Naive selection can lead to poor performance as accessing the tree nodes will cause cache misses which could be avoided with better tree layout in memory.

5.3 Dynamic Skewed Trees

So far we have focused on static tree design. However, the tree could be re-structured just as the system runs and memory access statistics are gathered. Similar idea could be used with the skewed tree. Rather than re-balance the tree, however, the goal is to move the tree nodes around so the the currently frequently accessed memory pages (leaf nodes) are on the shortest path. This will significantly increase hardware complexity, however, may be worth the performance improvements. The dynamic trees should be studied to understand their tradeoffs.

5.4 Extending Previous Schemes

A large number of memory checking schemes have been proposed. Many have also included encryption to provide both confidentiality and integrity protections. All these schemes, however, are based on a full, balanced trees. Updating these schemes with a skewed Merkle tree design would give best of both worlds. Previous optimizations, such as the Bonsai tree [16] or the cached trees [11] could further be improved if the number of memory accesses to the tree nodes is reduced through the use of a skewed tree. Also, an interesting question is the tree structure and how the skewness would differ when used in plain hash tree versus use within one of these previous optimizations.

5.5 Hardware-Software Co-Design

Conversely to matching the tree structure to the memory access pattern, software could optimize memory accesses given a fixed tree structure. If the software and hardware are designed together, the software could be informed how to allocate the memory so that the frequently accessed pages are on the short paths of the tree. Given that it is very difficult to change hardware after manufacturing, and that software changes more rapidly, this may be a good design direction. Just as software can optimize memory accesses for cache sizes, the software could optimize them for the particular memory checking scheme currently implemented.

6. RELATED WORK

Many designs for memory integrity checking, with various optimizations have been previously proposed by researchers. There are a number of surveys, e.g. [7], which detail existing hardware memory authentication techniques. Below is only a brief summary for some of these works.

6.1 Memory Checking with Hash Trees

Hash trees were originally presented in a patent by R. C. Merkle published in 1982 [14], after whom they are named. The original use of hash trees was for checking signatures in public key crypto systems. In 1994, Blum [2] presented a paper that describes on-line and off-line memory checkers. The work also established a bound of $O(log(N))$ of private secure memory needed to check a string of size N.

Works on memory integrity checking in computer systems, such as in [11], presented numerous designs for hash tree based schemes for memory integrity verification: chash, mhash or ihash; in addition to using a tree structure, these designs also store hashes in the processor cache to improve performance. Researchers have explored secure and fast architectures for authenticating memory when memory pages are accessed, or shared, by multiple processing elements [18]. Integrity checking tree designs which allow for parallelization of both the checking and update procedures have been presented as well [12].

To reduce the hash tree size, [4] introduces reduced address space (RAS) which is a virtual address space but excludes pages not used by the program – the tree is built over the RAS and thus has a smaller size. In [17], on the other hand, a one-level scheme for memory protection in distributed shared memory (DSM) systems is presented; in the scheme typical counters are used to protect the memory. The counters are securely shared between processors so they can be sent to requester ahead and so data from home node's memory can be sent directly to requester (no re-encryption)

Energy efficiency considerations have also been proposed, and researchers have looked at augmenting the tree structure with timestamps and timestamp cache to speed up the checking procedure [15]. Previous work on hash trees focuses on using full, balanced trees. Publications, e.g. [21], show how to select optimal block size and depth for Merkle tree. However, non-full, balanced trees have not been previously proposed for memory integrity checking.

6.2 Integrity Checking Combined with Encryption

Numerous works have combined hash checking with encryption to provide both integrity and confidentiality protections. For example, [19] presented an incremental hash of logs of memory operations combined with an encryption scheme based on one-time-pads and time stamps. In [9], authors present a parallelized encryption and integrity checking architecture. In [8], a Tamper Evident Counter Tree (TEC-Tree) is presented which uses encryption and nonces to protect integrity of the system's memory.

Meanwhile, [22] gives a good introduction to latency and throughput considerations and to the combination of encryption and hashing. For example, using Galois/Counter Mode (GCM) mode for integrity and confidentiality can im-

prove the performance. Also, address independent seed encryption (AISE) [16] has similarly been proposed which uses symmetric encryption algorithm (such as AES) in counter mode to protect memory. In the work [16], a Merkle tree based integrity verification that extends the tree to protect off-chip data and Bonsai Merkle Trees (BMT) are proposed.

6.3 Use of Integrity Checking in Secure Architectures

Secure architectures such as AEGIS [20] and Bastion [5] have incorporated hardware memory integrity verification. Thanks to use of the hash trees (and encryption) such architectures can reduce the trusted hardware to only the microprocessor chip.

6.4 Skewed Merkle Trees

The idea of skewed Merkle trees has been proposed before in other scenarios. Motivated by the fact that memory layout of full, balanced Merkle trees can hinder the tree traversal (e.g. cache misses due to accessing sibling nodes and parent node), researchers have explored skewed Merkle trees, and how their use can optimize binary search tree performance [3]. With careful layout, we also benefit from this fact as we use a skewed tree. Skewed Merkle tree traversal or search has also been explored in context of certificate revocation schemes [13, 10]; in certificate revocation, efficient traversal of hash tree is important and the tree may not be a full, balanced tree depending on which certificates have been revoked and placed in the tree.

7. CONCLUSION

We presented a new solution to hardware memory integrity checking based on skewed Merkel trees. Because not all memory locations are accessed equally frequently in a modern OS, our skewed Merkle three design offers better performance as the frequently accessed memory locations can be located on the leaves of the skewed Merkle tree that have shorter path to the root. The tree can be designed such that fewer nodes of the tree have to be accessed during integrity checks of the frequently accessed pages. Skewed Merkle trees offer better system performance, do not impact caches as much as full Merkle trees, and do not require more storage than full, balanced Merkle trees.

7.1 Ongoing and Future Work

Our ongoing and future work focuses on integration of the skewed merle trees with some of the other proposals (such as the cached trees or parallelzable trees). We believe the new type of hash tree can be substituted for the full, balanced trees previously used in the other schemes, and thus give even further performance improvements.

8. REFERENCES

[1] Inception, a FireWire physical memory manipulation and hacking tool. `http://www.breaknenter.org/projects/inception/`.

[2] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2-3):225–244, 1994.

[3] G. S. Brodal and G. Moruz. Skewed binary search trees. In Y. Azar and T. Erlebach, editors, *Algorithms – ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 708–719. Springer Berlin Heidelberg, 2006.

[4] D. Champagne, R. Elbaz, and R. Lee. The reduced address space (ras) for application memory authentication. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *Information Security*, volume 5222 of *Lecture Notes in Computer Science*, pages 47–63. Springer Berlin Heidelberg, 2008.

[5] D. Champagne and R. Lee. Scalable architectural support for trusted software. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Jan. 2010.

[6] J. Corbet, A. Rubini, and G. Kroah-Hartman. Linux device drivers, chapter 15, memory mapping and dma. `http://lwn.net/images/pdf/LDD3/ch15.pdf`.

[7] R. Elbaz, D. Champagne, C. Gebotys, R. Lee, N. Potlapally, and L. Torres. Hardware mechanisms for memory authentication: A survey of existing techniques and engines. In M. Gavrilova, C. Tan, and E. Moreno, editors, *Transactions on Computational Science IV*, volume 5430 of *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg, 2009.

[8] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin. Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 289–302. Springer Berlin Heidelberg, 2007.

[9] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, pages 506–509, 2006.

[10] F. F. Elwailly, C. Gentry, and Z. Ramzan. Quasimodo: Efficient certificate validation and revocation. In F. Bao, R. Deng, and J. Zhou, editors, *Public Key Cryptography (PKC)*, volume 2947 of *Lecture Notes in Computer Science*, pages 375–388. Springer Berlin Heidelberg, 2004.

[11] B. Gassend, G. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 295–306, Feb. 2003.

[12] W. Hall and C. Jutla. Parallelizable authentication trees. In B. Preneel and S. Tavares, editors, *Selected Areas in Cryptography*, volume 3897 of *Lecture Notes in Computer Science*, pages 95–109. Springer Berlin Heidelberg, 2006.

[13] M. Karpinski and Y. Nekrich. A Note on Traversing Skew Merkle Trees, 2004. `eccc.hpi-web.de/report/2004/118/`.

[14] R. C. Merkle. Method of providing digital signatures, 1982. U.S. Patent 4309569.

[15] S. Nimgaonkar, M. Gomathisankaran, and S. P. Mohanty. Tsv: A novel energy efficient memory

integrity verification scheme for embedded systems. *Journal of Systems Architecture*, 59(7):400–411, 2013.

[16] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 183–196, 2007.

[17] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin. Single-level integrity and confidentiality protection for distributed shared memory multiprocessors. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 161–172, Feb. 2008.

[18] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 123–134, 2004.

[19] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 339–350, 2003.

[20] 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 Annual International Conference on Supercomputing (ICS)*, pages 160–171, 2003.

[21] D. Williams and E. G. Sirer. Optimal parameter selection for efficient memory integrity verification using merkle hash trees. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA)*, pages 383–388, Aug. 2004.

[22] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 179–190, 2006.