Guaranteeing memory integrity in secure processors with Dynamic Trees

ECE594 – Advance computer architecture, Final Report

Shashank Khanvilkar
{shashank@evl.uic.edu}

Abstract: *Due to the widespread software copyright violations (piracy, reverse engineering and tampering), significant efforts have been made to protect applications from host attacks. With the advent of open operating systems like Linux, it is has become even easier for adversaries to hack the OS and inflict such attacks. As a result, it is becoming increasingly difficult to trust OS for protecting software copyrights. Recently, an eXecution Only Memory (XOM) architecture has been proposed to support copy and tamper resistant software, where the program and data are stored in encrypted format outside the CPU boundary and decrypted just before being executed by the CPU. XOM uses a memory integrity verification scheme that can protect it against spoofing and splicing attacks but cannot protect it against replay attacks. In this report, we present an efficient memory-integrity verification scheme called Dynamic-Trees (or D-Trees), which eliminates these kinds of attacks and provides an efficient implementation. D-trees create a recursive tree structure with multiple root nodes and eliminates many short-comings (higher latency for updates and limited arity) in Merkle trees. This report also presents a brief survey of existing schemes proposed to improve Merkle trees and their relative advantages and disadvantages.*

INTRODUCTION

A well-known problem in computer security research is concerned with protecting the integrity of a benign host and its data from attacks against malicious client programs [1]. A recent surge of interest in mobile agent systems, however, have caused researchers to focus attention on a fundamentally different aspect of security where a benign client code is being threatened by a malicious host on which it has been downloaded or installed. A malicious host attack typically takes the form of intellectual property violations where software is illegally copied and resold (Software piracy), valuable piece of code is extracted from program binary and incorporated in competitor's code (Reverse Engineering) or program code is altered to reveal trade secrets (Software tampering) [2]. Software copyright protection plays an important role in assuring the software market value and a fair return on development investment. A study in 2001 done by the Business Software Alliance showed a 12 billion dollar loss in the software industry due to such copyright violations and preventing it will surely have a large impact on software economy. Thus, it is important to develop foolproof schemes that will completely disallow unauthorized execution/tampering of software.

Several techniques, at both the hardware and software level, have been proposed to provide such protection (see Table 1 for a survey). While application-level schemes like watermarking, code-obfuscation or Tamper-proofing can provide limited protection against certain kinds of attacks, operating system-level techniques (like Microsoft's Next Generation Secure Computing Base and Stanford's TERRA) require advanced mechanisms like secure-boot, trusted kernel or virtual machine monitor to offer suitable protection. As such hardware-level schemes seem to be the most appropriate at this time. In this report, we focus on enhancing the security properties of one such technique, called XOM (eXecute Only Memory) [5], where the only trusted hardware entity is the processor itself. In other words, the trusted computing base is limited to the processor boundary and all other hardware components are considered vulnerable to security attacks.

In XOM, software is stored in encrypted form in external memory using a combination of public- and symmetric-key cryptography and can be decrypted only by the processor. Since such software can run on only one processor software piracy is prevented. All secret keys are securely protected inside the processor and accessible only to certain trusted programs. All temporary data between the processor and external memory is also encrypted. Thus adversaries are prevented from examining the clear text instructions and reverse engineer the code. XOM prevents any kind of software tampering by providing basic memory authentication/verification procedures that attempt to guarantee external memory integrity (but fails).

Inside the XOM processor, processes execute within their own secure compartments: an abstract structure formed by tagging registers/caches with special tags (called XOM-ids) that identify the principal in execution. Data-flow between processes is strictly controlled across well-defined interfaces, and no process (not even the Operating System) is allowed to access the

data belonging to other principals. Special instructions allow the OS to store/restore process state (without actually reading the state) during interrupts in an internal private cache.

Table 1: Survey of techniques to prevent software copyright violations.

	Defense Mechanism	Piracy	Reverse-Engg.	Tamper	Remarks
Application Level	Watermarking [2]	X			Copyright notice is stealthily embedded in the source code, which helps in tracking and prosecuting pirated software. Discourages (DOES NOT PREVENT) Software Piracy attack.
	Code-Obfuscation [2]		X		Transforms the program into an equivalent code that is harder to reverse engineer. Discourages Reverse Engineering attacks, by increasing the cost to reverse engineer as compared to buying it.
	Tamper-proofing [2]			X	Embeds guards in the program that causes it to abort or malfunction if modification is detected. Prevents Software Tampering attacks
	Traitor tracing, Secret Sharing, Reference States, secure evaluations are being actively researched.				
OS Level	Microsoft's NGSCB [11]	X	X	X	Uses the TCPA specifications to create a secure/trusted platform. Both host and client are protected from each other. Vulnerable to Hardware attacks (Xbox). Uses Secure Boot to load Nexus.
	Stanford's TERRA [8]	X	X	X	Uses a Trusted Virtual Machine Monitor (TVMM) that partitions tamper resistant hardware platform into multiple, isolated virtual machines providing the appearance of multiple boxes on a single general purpose computer. Each such machine can be customized
Hardware Level	XOM [5]	X	X	X	TCB is limited to the processor. Uses Encryption/Authentication to protect the privacy of software. All off-chip data is in encrypted form.
	AEGIS [9]	X	X	X	Enhances XOM to provide tamper-evident and tamper-resistance modes.
	External Hardware [10]	X		X	Dongles, smartchips and SecureIDs. Vulnerable to a host of attacks.

An in-depth analysis of the XOM architecture, however, reveals that XOM suffers from two major drawbacks. Since every off-chip memory transaction undergoes encryption (and decryption), even with the most optimistic assumption of finishing the crypto process in 48 cycles with fully pipelined hardware, causes the processor performance to degrade by ~17% [3]. Yang et. al. [3] sufficiently solve this problem using a *one-time pad* encryption algorithm that overlaps the encryption latency with the memory access latency to post an improvement of ~35% over the previous case. A similar encryption mechanism with some basic changes has also been used in [4]. Thus, we do not address the issue of speeding up the encryption and decryption and

concentrate on the more pressing issue of preserving data integrity in external memory, as the memory-integrity verification scheme used by XOM is faulty [7]. In this report, we discuss an efficient scheme to overcome this deficiency. As discussed later, a number of schemes have been proposed in the literature (see Gassend et. al. [6] for a survey) to provide integrity verification. However each scheme has its drawbacks that are further explored in upcoming sections.

The remainder of this report is organized as follows. We first describe the XOM architecture in Section 2. Then we elaborate on how replay attacks can occur in XOM, and discuss some existing schemes to prevent replay attacks and their respective drawbacks in Section 3. Section 4 illustrates the detailed architecture design of our Dynamic Tree algorithm and proves its security properties. Section 5 provides a basic performance analysis and concludes this report.

II. XOM ARCHITECTURE OVERVIEW

XOM is a generic microprocessor that maintains separate compartments for programs. On-chip compartments are realized using architectural tags (called XOM-ID's), while encryption is used to maintain isolation when data is written to external memory. Every program is initially encrypted using a symmetric key K_s that is associated with a unique XOM-ID. When the code is about to be executed, it is read from memory, decrypted and tagged with this ID. Additionally all on-chip data or code that belongs to a program are also tagged. The hardware disallows adversarial programs from tampering with the user's data by checking its tag against the XOM-ID tag of the active program and raising an exception on a mis-match. In this way, the tags act as an identifier of the writer of the data, and thus determine who can read it. A program may also allow another program to read its data by explicitly moving that data from its private

compartment to a shared compartment. In this way, programs in separate compartments may selectively share data.

Some additional instructions are added to the instruction set of the XOM machine for users to take advantage of the XOM hardware. *Secure load* and *secure store* instructions allow programs to preserve the XOM-ID of data when storing it from registers to cache. Later, if the data is flushed to memory, it is automatically encrypted using the compartment key K_s indicated by the XOM-ID. To allow operating system to save the state of the interrupted user program during a context switch, without allowing it to read user data, XOM provides the *register save* instruction. Here the machine encrypts the register contents and stores the cipher text in the target register (along with a MAC). The operating system can then store the state of an interrupted process, but does so safely since it is only allowed to handle only the encrypted state. When the operating system wishes to restore the register contents it uses the *register restore* instruction. The XOM machine returns those values back to the registers.

To guarantee memory integrity, the XOM processor adds a hash of the data and its address when data is stored to external memory to protect against the tampering of memory values. This prevents an adversary from substituting random values in place of encrypted values (*spoofing attack*), or from copying encrypted values from one address to another (*splicing attack*). This simple mechanism, however, does not prevent replay attacks, where encrypted values at the same address are restored at a later point of time. This is further discussed in the next section.

The XOM software model requires all programs to be encrypted by the vendor. Such encryption not only protects the privacy of the software but also guarantees that it can only run on a single target processor. To maximize security and performance, the software is encrypted using a combination of symmetric and asymmetric key cryptography. Every XOM chip comes

installed with a private-key $K_{private}$ (not accessible to anyone) and a public-key K_{public} (open to all). The vendor first encrypts the software using some fast symmetric key cipher with a private key K_s. To communicate K_s to the XOM processor, the vendor uses K_{public} to encrypt it and ships it along with the software. The execution of the protected software begins with computing K_s from $K_{private}$. This may take a relatively long time, but is carried out only once at the beginning of the execution. Once the symmetric key K_s is decrypted, instructions encrypted using K_s can be recovered much faster. Because of the public-key encryption, software encrypted for processor-1 cannot be run on processor-2.

The XOM architecture can be built on top of any existing architecture through modest modifications to the processor hardware, and has been shown to be realistically implementable [5]. Such modification may include, adding tags to all registers and cache lines, updating the system software to handle interrupts, introducing an encryption/decryption unit at the external memory interface etc.

III. PREVENTING REPLAY ATTACKS IN XOM

Replay attacks occur when the adversary is able to record some values and use them at a later point of time. Two scenarios have been envisioned where replay attacks can occur in XOM. These are presented below:

1. *Replay attacks due to cache invalidation [7]:* Assume a data-value *A* residing at the address *0x20*. Suppose that some function causes *A* to be written to main memory. The program then updates *0x20* with *B*, which is still in cache and not updated to memory. Under such situation, the adversary can invalidate the cache line, causing the program to read back the

old value of *A* rather than *B* thus resulting in a replay-attack. The required sequence of steps to carry out this attack is illustrated in Table 2.

Table 2: Replay Attack due to cache invalidation.

Action	Cache	Memory	Hash
User writes A to cache at virtual address 0x20	A	Ø	{Ø}
Cache is flushed to memory	Ø	Enc(A)	{Hash{A|20|1}}
User writes B to cache at virtual address 0x20	B	Enc(A)	{Hash{A|20|1}}
Adversary Invalidates the cache	Ø	Enc(A)	{Hash{A|20|1}}
Processor reads back value A, instead of B	A	Enc(A)	{Hash{A|20|1}}

2. *Replay attack due to past data-recording [4]:* Here again assume that *A* resides at virtual address *0x20*. If some function causes the causes *A* to be written to the main memory, an adversary can record this and replay it at some later time. The required sequence of steps to carry out this attack is illustrated in Table 3.

Table 3: Replay Attack due to past data-recording.

Action	Cache	Memory	Hash
User writes A to cache at virtual address 0x20	A	Ø	{Ø}
Cache is flushed to memory	Ø	Enc(A)	{Hash{A|20|1}}
Adversary stores the contents of memory and hash to some other place.			
User writes B to cache at virtual address 0x20	B	Enc(A)	{Hash{A|20|1}}
Cache is flushed to memory	Ø	Enc(B)	{Hash{B|20|1}}
Adversary replaces the new value with the old stored value	Ø	Enc(A)	{Hash{A|20|1}}
Processor reads back value A, instead of B	A	Enc(A)	{Hash{A|20|1}}

Replay attacks due to cache invalidation can be prevented by extending the instruction set of XOM processor. These additional instructions will not allow the adversary to directly invalidate

a cache line belonging to a different principal. Instead such lines will be first flushed to external memory and their XOM ID will be nullified, thus releasing it to be used by others. Although this scheme provides a simple solution to prevent replay attacks of the first-type, it fails to prevent the second. A deeper insight into the problem root reveals that any scheme that attempts to store Hashes (or MACs) alongside data chunks is susceptible to such attacks. This is becuase the adversary has complete access to both data and MAC. Thus this problem requires a completely different approach.

One approach is to completely isolate data and corresponding hashes and preventing the adversary from accessing the hashes. If all hashes corresponding to every cache chunk are stored inside the secure processor, inaccessible to the adversary, then all kind of attacks (spoofing, splicing or replay) can be easily prevented. As a simple proof to this statement consider this: if the adversary cannot change the hashes, any change to the data will always result in an integrity failure exception.

Separating the hashes from data, however leads to additional concerns about addressing the hashes (*how to find the right hash during integrity verification?*) and securing the hashes (*how to efficiently hide the hashes from the adversary?*). Although our simple scheme of storing all hashes inside the processor will provide the most efficient integrity verification (boils down to reading only the concerned data line), it is highly idealistic and very costly in terms of real-estate on the processor. For example, if each data chunk is 128B, and a 1MB address space needs to be protected, 8K hashes will need to be stored inside the processor. If each hash is 16B in size the total secure memory required will be 128K. The size of this hash table increase linearly with the protected address space. A number of other schemes (Merkle trees and its variants) take this

approach, while balancing space (to store the hashes) and time (to verify the integrity). These are discussed next.

Merkle trees capitalize on space efficiency for storing hashes at the expense of time to access them. Figure 1 illustrates a binary Merkle tree. Data (nodes V1 , V2 …) is located at the leaves of the tree. Each internal node contains the hash of the concatenation of` its children. The root of the tree is always stored in secure memory where it cannot be tampered with.

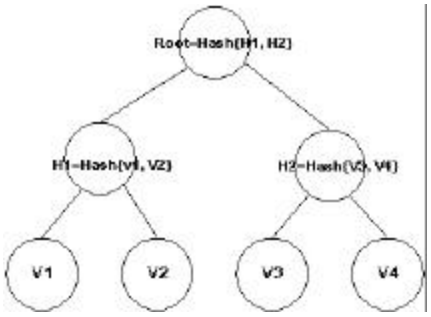

Figure 1: Binary Merkle Trees

To check the integrity of a node in the tree, the processor:

(i) Reads the node and its siblings from the memory,

(ii) Concatenates their data together,

(iii) Computes the hash of the concatenated data, and

(iv) Checks that the resultant hash matches the hash in the parent.

These steps need to be repeated all the way to the root of the tree. To update a node, the processor checks its integrity as described above while it

(i) Modifies the node, and

(ii) Re-computes and updates the parent to be the hash of the concatenation of the node and its siblings.

These steps are again repeated to update the whole path from the node to the root, including the root. With a balanced m-ary tree, the number of chunks to check on each memory access is $\log_m(N)$, where N is the number of chunks in the protected memory space. Continuing with our previous example, the number of hashes needed to be stored inside the secure processor for this binary Merkle tree will be just 1 (only root node needs to be stored). However every memory access (read or write) will require verification/update of up to 13 nodes ($\log_2 2^{13}$) till the root, which increases the latency of hash verification by 13 times to the previous case. Note that increasing the arity of the Merkle tree does not help in this case, as increased arity leads to larger data being required to be fetched for integrity verification, which again increases latency.

Cached Hash Trees improve the performance of Merkle trees by caching the internal hashes on-chip with regular data. The processor trusts data stored in the cache, and can access them directly without any checking. Therefore, instead of checking the entire path from the chunk to the root of the tree, the processor checks the path from the chunk to the first hash it finds in the cache. This hash is trusted and the processor can stop checking further. When a chunk is evicted from the cache, the processor brings its parent into the cache (if it is not already there), and updates the parent in the cache. Cached hash trees reduce the verification latency for reads, but writes require all nodes upto the root to be updated.

Log-hash trees is a new approach of verifying memory integrity with low run-time overhead. Here the processor maintains a read log and a write log of all of its operations to off-chip memory. At runtime, the processor updates logs with minimal overhead so that it can verify the integrity of a sequence of operations at a later time. To maintain the logs in a small fixed amount

of trusted on-chip storage, the processor uses incremental multiset hash functions. When the processor needs to check its operations, it performs a separate integrity-check operation using these logs. The drawback of log-hashes is that memory integrity is verified only at regular intervals. Hence breach in the integrity of data is reported only at the end of verification cycle. This may not be acceptable in many situations. Moreover extra space is required inside the processor to maintain the memory access logs. At this point, it is not clear about what should be the duty-cycle for log-hash trees to perform efficiently.

Table 4 tabulates the advantages and disadvantages of all the schemes that we have discussed above.

Table 4: Memory Integrity Verification Techniques.

Hash Hiding technique	Advantages	Disadvantages
Hiding all the hashes inside the secure processor	Simplicity.	Un-scalable
	Takes constant time to verify both memory reads/writes.	Very costly in terms of additional cache required for storing the hashes.
	Highly idealistic	Very unrealistic.
Using Merkle Trees or Cached Hash trees.	Uses Tree Structure to reduce the space requirement inside the secure processor.	All children need to be read for integrity verification. This limits the value of "m". Higher "m" leads to greater latency to verify the integrity of a single node.
	Memory integrity verification for read/write takes $O(\log_m N)$: m is the arity.	Memory writes require all internal nodes up to the root node to be updated.
	Cached hashed trees reduce the time for memory read's by caching internal nodes of the merkle tree.	The entire tree structure needs to be in place before execution.
Using Log Hash trees	Uses two incremental hashes (ReadHash and WriteHash) that are stored inside the secure processor.	Some mechanism to record the memory access patterns needs to be in place.
	Sequence of memory operations are verified instead of verifying each individual cache line.	Memory integrity holes cannot be reported until the end of memory verification cycle.
		Memory verification cycle introduces additional latency, which may not acceptable in many situations.

IV. DYNAMIC TREES

To balance the demands on the requirement for hash size and time for hash verification, we take a conservative approach in Dynamic Trees (or D-Trees). D-Trees take advantage of the fact that hashes for READ_ONLY nodes (like instructions) or WRITE_ONLY nodes (like program output) need not be kept secret or hidden from the adversary. A simple addressed MAC (MAC calculated over the concatenation of the node virtual address and data) added to the node should suffice and provide the necessary security. This is due to the fact that READ_ONLY or WRITE_ONLY nodes cannot suffer from replay attacks and a simple addressed MAC can prevent all other types of attacks. For example, even if the adversary has access to both data and MAC, he cannot replace valid data with junk values (as the calculated MAC will be different) or replace both data and MAC with valid cipher-texts from other location (since the MAC uses the virtual address for verification). The adversary cannot manipulate the MAC either without getting noticed, because MACs involve a secret key not known to the adversary. If a 16B MAC is added to every 128B node, this would increase the memory requirement by 12.5%.

For dynamic data created during the program execution (stack or heap), D-trees are created on the fly. A recursive data structure is used where the parent nodes are created only when the child nodes get evicted to external memory. All parent nodes contain hashes of X leaf data nodes belonging to contiguous memory chunks. As long as the parents remain inside the secure processor, they play the role of the root node in a Merkle tree and are treated in the same was as the leaf data nodes when they themselves are evicted. In this way a sparse forest is created, with multiple root nodes. The following pseudo-code explains the working of the algorithm.

```
1.  Struct node {
2.     int address;
3.     int value;
4.  }Node;

5.  Void CalculateHash(Node Child){
6.      Node parent;
7.      if (Child evicted to external memory){
8.            parent = new(Node);
9.            parent.value = hash(Child.value);
10.           parent.address = Offset  + Child.address*sizeof(Hash)/sizeof(Child);
11. //where offset = integral multiple of the sizeof(node);
12.     }
13. }

14. Void VerifyIntegrity(Node Child){
15.     Node parent;
16.     parent.address = Offset + Child.address*sizeof(Hash)/sizeof(Child);
17.     If (parent is present in the cache){
18.         if (parent.value == hash(Child.value)){
19.             integrity is verified;
20.             destroy(parent.value);
21.         }else{
22.             integrity breach;
23.             throw an exception;
24.         }
25.      }else{ //parent not present in cache
26.         VerifyIntegrity(parent);
27.      }
28. }
```

Figure 2:Pseudo-code for calculating/verifying integrity in D-trees

In the above pseudo-code, `calculateHash()` creates a new parent node when the child is evicted. Here the child node can be either a leaf data node or an inner parent node. The address of this new node is calculated in such a way that each parent node can accommodate the hash values for "`X=sizeof(Hash)/sizeof(leaf data node)`" number of children. For example, if an L2_cache chunk is treated as a leaf data_node with a size of 128B, and the hash has a size of 16B then each parent will contain the hash for 8 leaf data nodes and the integrity of all 8 can be verified by this single parent. `VerifyIntegrity()` recursively verifies the integrity of each node up to a point where the parent is found in the cache. By having a special cache replacement policy where child nodes are preferentially evicted before parents, a forest of Merkle trees with multiple roots is created. If a situation arises where all root nodes need to be flushed to memory,

a special routine is employed to collect all existing root nodes and replace them with a single hash that is guaranteed to be stored inside the secure processor.

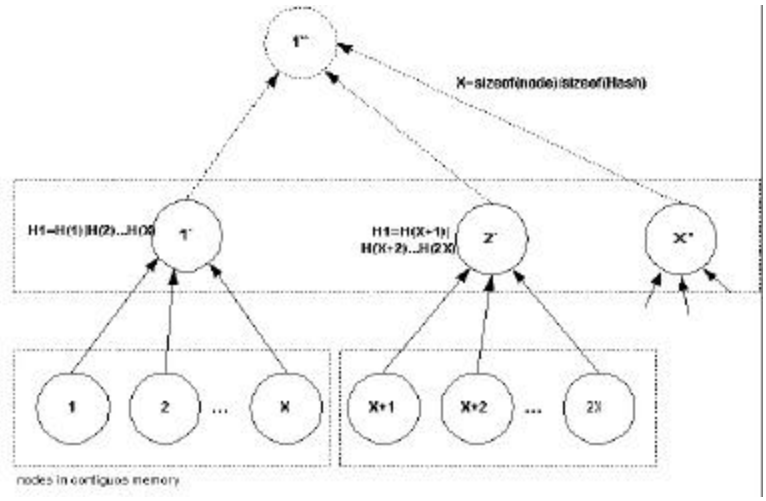

Figure 3: D-Trees

Figure 3, provides further details about how the root nodes are created. Here root node 1` is created when any of the child nodes {1, 2 .. X} get evicted to memory and contains the hashes for all of them. Root node 1`` is created when any of the previous roots {1`, 2` .. X`} are evicted. Also the cache replacement policy will make sure that the root node 1'' will never get evicted until one of its children are present in the cache.

Figure 4, illustrates a simple hardware enhancement to the basic processor to enable it to handle D-Tree based integrity verification. While the rest of the components are familiar, *H_Func* block needs more explanation. This block calculates the address where the hash for the node might have been stored using the virtual address. *H_Func* interfaces with L2_Cache to determine if the parent node is present. If necessary, *H_Func* will recursively get the parent nodes from memory till a root is found in the cache. During this time the secure processor can

speculatively execute the program. Since the hashes are important only at the level 2 cache, an optional cache may be added to *H_Func* to exclusively store the hash nodes. In this case, a more intelligent *H_Func* may be designed that will take responsibility of keeping track of hashes for each process. H_Func may use steal the memory bus during idle time.

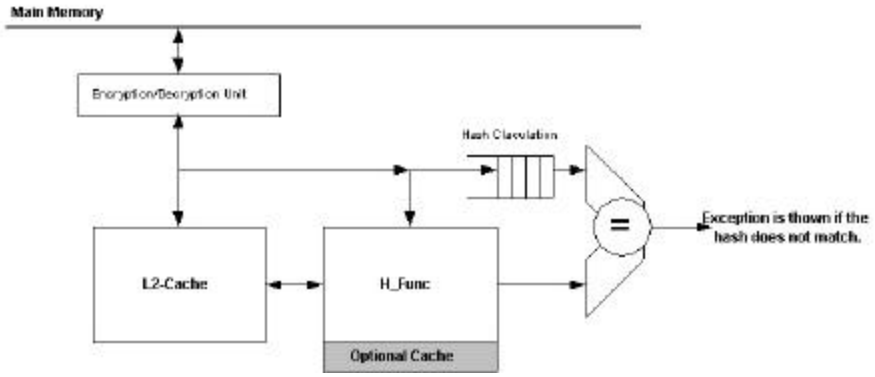

Figure 4: Sample Hardware for D-Trees.

V. PERFORMANCE ANALYSIS AND CONCLUSION

Because of time-limitations in this semester, a detailed simulation of the above scheme could not be performed. However a rough analysis of the expected performance gains follows. The arity of D-Trees is directly proportional to the node size or inversely proportional to the hash size. The node size is limited by the amount of data (or chunk) that can be simultaneously read from external memory. The width of the memory bus determines this factor. In modern architectures a 128B wide memory bus is quite common and we can assume the node size to be the same.

The hash size, however, depends on the security strength expected by application. For highly secure applications, hash produced by algorithms like MD5 (16B) (or SHA1 (20B)) can be used.

These algorithms are known to produce collision resistant hashes and can be trusted to provide the required level of security. (Many security papers advice that the size of the hash must be at least equal to the size of the secret key used for encryption [12]). In this case, if the node size is 128B, the D-tree will have an arity of 8 (or 6). MD5 algorithm can be implemented in hardware using a modest 50K gates.

If the security requirement, however, is relaxed, a lower end hashing algorithm like CRC (4B) or curtailed MD5 (8B) can be used. Curtailed MD5 is the same as conventional MD5, with the difference that only a subset bytes from the MD5 output is used (either the 1st or last *x* bytes). These hashes are not as collision resistant as conventional MD5, but their security properties may suit most real-life applications. In this case the arity of D-tree increases to 32 (in case of CRC) or 16 (for curtailed MD5).

Unlike Merkle trees D-trees do not create a static tree with a single root node, but creates a dynamic forest of trees with multiple roots. Since a Merkle tree concatenates all its children to calculate the hash, its arity cannot be increased without subsequently increasing the associated latency. The increased arity in D-trees allows more neighbor nodes to be verified by a single parent taking advantage of spatial locality in programs. Since multiple root nodes are present the worst case performance of this scheme can be expected to be $O(\log_m N)$ or same as the average performance for Merkle trees.

D-Trees, however, also suffer from the following drawbacks:

(i) A chunk is evicted from cache only because there is no place in the cache for putting a new chunk. D-Trees, however, create a new parent node, which requires an additional cache line. How can one handle this situation? Our solution is to reserve a part of the cache or introducing a special additional cache to handle such situations. The optional

cache added to *H_Func* block in Figure 4 may be used to serve this purpose. However it should be noted that a parent contains *X* child nodes. Hence potentially *X* children may be evicted from the cache and replaced by a single parent.

(ii) There may arise a situation where the cache needs to be flushed. How can D-trees keep track of the multiple root nodes that are dynamically created? An extra bit-map structure or a linked list may be used for this purpose. However this idea is just preliminary and more thought needs to be put into it.

(iii) How can one guarantee that child nodes are preferentially evicted to parents? A new cache replacement schemes needs to be investigated that will determine if any children of the parent are present in the cache and selectively evict them before evicting the parent.

REFERENCES

[1] Cowan, C., "Software Security for Open-Source Systems", IEEE security and Privacy, Jan 2003.

[2] Collberg, C.S., "Watermarking, Tamper-Proofing, and Obfuscation Tools for Software Protection", IEEE transactions on Software Engg., vol. 28(8), Aug 2002.

[3] Yang, J., Zhang, Y. and Gao, L.,"Fast Secure Processor for Inhibiting Software Piracy and Tampering", Intl. Symp. on Microarchitecture (MICRO-36), 2003.

[4] Suh G.E., Clarke, D., Gassend, B., Dijk, M. and Devadas, S., "Efficient Memory Integrity Verification and Encryption for Secure Processors", Intl. Symp. on Microarchitecture (MICRO-36), 2003.

[5] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J. and Horowitz, M.,"Architectural Support for Copy and Tamper Resistant Software", In Architectural Support for Programming Languages and Operating Systems, pages 168--177, Cambridge, MA, November 2000.

[6] Gassend, B., Suh G.E., Clarke, D., Dijk, M. and Devadas, S., "Caches and Hash Trees for Efficient Memory Integrity Verification", Proc. of the 9th Intl. Symp. on High-Performance Computer Architecture (HPCA'03), 2003

[7] Lie, D., Mitchell, J. and Horowitz, M.,"Specifying and Verifying Hardware for Tamper-Resistant Software", IEEE Symp. on Security and Privacy, May 2003.

[8] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M. and Boneh, D., "Terra: A Virtual Machine-Based Platform for Trusted Computing", Proc. of the 9th ACM symp. on Operating system principles, 2003.

[9] Suh G.E., Clarke, D., Gassend, B., Dijk, M. and Devadas, S., "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing", Proceedings of the 17th annual international conference on Supercomputing, 2003.

[10] Naccache, D. and M'Raihi, D., "Cryptographic Smart Cards", IEEE Micro, 1996.

[11] England, P., Lampson, B., Manferdelli, J., Peinado, M. and Willman, B., "A Trusted Open Platform", Computer, vol: 36(7), pp: 55-62, July 2003.

[12] Kessler, G., “An Overview of Cryptography”, http://www.garykessler.net/library/crypto.html, May 1998.