An Improved Memory Integrity Protection Scheme

February 21, 2010

Abstract

Memory integrity protection is crucial for many applications that store critical data in the memory. A secure real-time protection scheme can eliminate many potential attacks and protect such critical data effectively. However, existing schemes are not fast enough to provide on-line protection. The main difficulty in enrolling real-time memory integrity protection schemes is that the cryptographic operations typically cost a few hundred cycles per memory access. In this paper, we make an effort to reduce this overhead by proposing an improved memory integrity protection scheme. In addition, we for the first time propose a provably secure scheme that takes advantage of the error inheritance property, which can eliminate the costly check process that is normally required. More specifically, we propose the parallelization of the scheme, randomization of tagged data to eliminate potential attacks on error inheritance, and hashing of seeds (instead of the data) which enables delayed AES mask computations. The security of the proposed scheme is rigorously analyzed and the performance is measured. When these techniques are implemented simultaneously, the peak performance can be improved by up to a factor of 5 over a previously proposed scheme based on Merkle Trees.

1 Introduction

Integrity is a necessary condition for an entity to be trusted. The integrity of the memory is vital to all applications that store critical data. A compromise of memory integrity can cause a wide range of problems ranging from cheating in games to financial fraud. For instance in online games, memory can be modified by the owner of the platform to gain tactical advantage over other players [13]. A much more dangerous situation is a Trojan program forging fake transfers by modifying the memory space of a web browser who is visiting the online banking site. A real-time memory integrity protection scheme can provide an effective defense against such attacks.

Modern operating systems (OSs) prevent one process from accessing the data of another process by memory isolation techniques such as the virtual memory. Ideally, this can protect against memory modification attacks implemented by malicious code. However, complex software systems such as an OS usually have numerous bugs that attackers can exploit. Despite the continuous release of bug fixes and patches, new bugs are being discovered almost on a daily basis. Furthermore, memory protection enforced by the OS does not protect against physical attacks. If the attacker can physically read and write to the memory, the OS may not even detect the attack.

Thus, we need a simple yet efficient scheme to provide memory integrity protection. In addition, the code for the scheme also needs to be protected. If the code is executed in a trusted environment and is simple enough to avoid any possible flaws, the memory will be well protected. In practice, it can either be protected directly by executing in trusted hardware such as the ARM TrustZone [15] or by the secure kernel of the OS based on trusted hardware such as the Trusted Platform Module (TPM).

Many such memory integrity protection schemes have been proposed in the past [2, 3, 4, 5, 6, 7, 14, 21]. However, the efficiency of these schemes is still not good enough for practical use. The main problem is the overhead introduced by the cryptographic functions. Though many popular cryptographic functions such as SHA-1 are fast enough for use in networking, they are much too slow to be used in memory integrity protection. Furthermore, what differentiate the memory integrity protection problem from other security applications that require integrity checks are the so-called replay attacks. Extra measures must be taken to prevent replay attacks.
The scheme we propose in this paper is based on an earlier memory authentication scheme introduced in [14]. For easy reference we name this scheme as the HHS scheme in the rest of the paper. Generally speaking, the HHS scheme efficiently combines the Merkle Tree and a family of universal hash functions called NH to provide memory integrity protection. In this paper, we proposed an improvement over the HHS scheme which reduces the overhead while retaining the security level. We also propose taking advantage of the so-called "error inheritance" property which allows our scheme to detect earlier attacks without the need to check the integrity before each individual memory access. This feature brings real-time memory integrity protection one step closer to practical reality.

The rest of the paper is organized as follows: in Section 2, we will briefly introduce our attacker model. The original HHS scheme is outlined in Section 3. Our work will be discussed in Section 4 and the complete scheme will be given in Section 5. The security of this scheme will be analyzed in Section 6 and the performance will be discussed in Section 7 which includes a performance simulation.

2 Objective

In this paper, we want to build a provably secure protocol to provide memory integrity protection service. Before introducing our scheme, we will clarify the attacker model and outline some assumptions.

The memory in our model is divided into two parts:

- **Protected memory** which is protected by higher level of trust service providers such as the ARM TrustZone or trusted OS extensions\(^1\). It is assumed to be secure in this paper, i.e. the attacker can neither view nor modify the contents of it. The protected memory is used to store the code for our scheme, the root of the Merkle Tree and the keys for the AES and the NH. It can also provide a secure cache for the scheme we propose. Our goal is to make this piece of memory as small as possible.

- **Unprotected memory** which is the vulnerable memory that we want to provide integrity protection. The tags, seeds and randomizers used in our protection scheme will also be stored here. Any ordinary process that want to use our integrity protection service must call the Read and Update (Write) functions provided by our scheme to access the memory.

The attacker in our model can perform any operations that an authorized user can and he can also view and modify the data in the memory directly at any time. Here 'directly' means that the attacker can also access the memory without the participation of the protection scheme, e.g. the inter-process memory access to a piece of protected memory. The goal of the attacker is to modify the data directly without being detected by our scheme. The attacker in our model is stronger than the attackers in practical cases. For example, in the case of the inter-process memory access, normally the attacker cannot make the victim process modify the data itself. In the case of our scheme, it means that the attacker cannot pretend to be victim process and call the Read and Update (Write) functions. However, the attacker in our model can call the functions to help achieving his goal of directly modification of the data. The depiction of our model is presented in Figure 1.

The goal of our scheme is to provide real-time integrity protection of the unprotected memory. When an attacker has once modified the data directly in the unprotected memory, the scheme should be able to detect this violation with a high probability even after a few other operations (some Read and Update calls from the victim process, etc.) Even if the attacker reverses his modification after the victim process has accessed that piece of data at least once (otherwise the attack will be meaningless), this modification should also be detectable by the scheme.

In addition, we make the following assumptions to ensure the security of the scheme. Firstly, the functions of the scheme are treated as atomic such that the attacker cannot perform attacks while the authentication scheme is executing. Secondly, if an attack is detected by the scheme, an alarm is raised and all the keys are flushed so that the attacker cannot exploit the scheme by repeating the attacks [19, 20].

\(^1\)Some Intel or AMD CPUs provide instructions such as GETSEC[ENTER] on Intel TXT to help performing a measured launch of the operation system with the help of the TPM. The code for the OS is authenticated before being executed. If we can use the authenticated code to build up the protected memory and the code for our scheme is also authenticated, we can achieve the required secure level for the protected memory. The authentication is only done before the loading of the OS so that it will not affect the performance. There are already some schemes using similar design such as 'Flicker'[18]
3 The HHS Scheme

The HHS scheme is a combination of the Merkle Tree and the universal hash function family. To prevent replay attacks, the HHS scheme chains the tags together using a Merkle Tree structure [1]. Merkle Trees provide an excellent authentication technique without the need for too much protected storage. Another significant advantage of Merkle Trees is that they can be updated incrementally [3], which is an important requirement for applications where online performance is critical [9].

The universal hash function family is a collection of functions that map data to digests [12]. The HHS scheme uses a universal hash function family \( NH \) which was proposed for use in the UMAC scheme [10]. The \( NH \) is computed as the following:

\[
NH_K(M) = \sum_{i=1}^{n/2} ((m_{2i-1} + k_{2i-1}) \mod 2^w) \cdot ((m_{2i} + k_{2i}) \mod 2^w) \mod 2^{2w}.
\]

It exhibits excellent performance in software [11] as it was designed to be naturally compatible with the instructions supported on common processor architectures. Suppose the \( w \) in the equation is 32 or 64, the only operations needed for computing the \( NH \) are additions and multiplications. However, despite the precise characterization of the statistical properties of universal hash functions, when employed alone they do not provide any cryptographic protection. Thus, the HHS scheme encrypts the results of the universal hash functions using random bit strings. The combination works like a one-time-pad and can provide provable security. Also, Toeplitz technique [16] [17] is used to strengthen the security.

The detail of a two level Merkle Tree with the 32-bit \( NH \) universal hash function applied twice using the Toeplitz approach as described above is outlined in Figure 2. Tags are generated by adding masks to the hash results. The masks are generated by encrypting a random seed using a block cipher. For this task the AES block cipher is employed. The seeds are randomly generated and saved in the unprotected memory in plaintext alongside the produced tags. The tags are then hashed to provide the root hash result. It is easy to extend this scheme to an l-level version by using the tags as the data for the higher level. This scheme is called the basic scheme in [14]. We will refer this scheme as the HHS Basic Scheme later. The sensitive data such as the root of the Merkle Tree, the keys is assumed to be kept in the protected memory mentioned in the last section.

As also mentioned in [14]. There are some possible ways to improve the efficiency of the basic scheme. One of them is the use of caches. In the CPU cache system, frequently referenced data is cached to reduce the frequency of the relatively costly memory accesses. In our case, the bottleneck of the basic scheme is in the encryption and decryption of the tags. Thus similarly, the caches can be used to reduce the frequency of the relatively costly AES operations. Either the masks or the unmasked hash results can be cached to
meet the needs of different applications. In addition, the basic scheme can be further improved by pre-computation and parallel computation. Basically, independent computations can be executed separately, either by pre-computation or parallel computation. In the basic scheme, the preparation of the seeds and masks is independent with calculating the hash results. Thus, pre-computation or parallel computation or both can be used to speed up that scheme. Parallel computation and pre-computation will need some extra computation units. However, they can improve the performance largely. Our scheme will also take advantage of the caches and the parallelism and improve the efficiency even greater.

![Figure 2: Outline of the HHS basic scheme in [14]](image)

## 4 Proposed Improvements

In this part, some other techniques that can help improve the efficiency of the scheme will be discussed. They can work together to reduce the overhead to a lower level.

### 4.1 Error Inheritance

One problem of the HHS scheme, as well as other similar authentication schemes is that the data need a check before every access or the system will fail to detect the unauthorized modification. For example, if the data is not checked before every access, one can modify the data and recover it after the system have read that piece of data but have not checked yet. In this way, the system will never be able to detect that it has read forged data. In a Merkle Tree structure, a check means at least a hash over an entire block. In the HHS scheme, though it is still quite fast, the check operation will become the bottleneck due to the incremental update. However, if the error inheritance property holds, we can use it to eliminate this check and can hence boost the performance. This property was firstly mentioned in [14] but not discussed in detail. In this paper, we will discuss the error inheritance property and make use of this property to improve the scheme.

With this property, any injected error that occurs within the execution will carry forward to the future values of the tag. Even if the tags are checked after a large number of cycles it is still possible to detect the injection of an error at one point in time with high probability. For example, consider the NH scheme. The hash result for a block of data where the attacker changed data blocks $M_1, M_2$ to $M'_1, M'_2$ yields

$$
\text{NH} = \text{NH}_{\text{others}} + 64 (M_1 + 32 K_1) \times 64 (M_2 + 32 K_2) \\
- 64 (M'_1 + 32 K_1) \times 64 (M'_2 + 32 K_2) + 64 (M''_1 + 32 K_1) \times 64 (M''_2 + 32 K_2)
$$
As long as the attacker cannot cancel the term calculated from $M_1'$ and $M_2'$, the attack can be detected. The SHA-1 based scheme on the other hand does not offer such a property. The tags will have to be checked in every update cycle to detect. The HHS scheme\(^2\) and the modification proposed in this paper will allow one to perform integrity checks at a much lower rate without the danger of missing an unauthorized modification, which translates to better performance.

The security of the error inheritance property stems from the fact that the attacker cannot hide the trace of a past attack. In NH, such a trace is the subtraction of the partial result which is calculated from old attacked data. However, if the attacker can control an authorized update, which may be possible, he can remove the trace by letting the system update to the attacked data. In the previous example, if the new authentic value for the attacked position is $M_1', M_2'$, the process will be as follows.

\[
\begin{align*}
\text{NH}_{\text{new}} &= \text{NH}_{\text{others}} + 64 (M_1 + 32 K_1) \times 64 (M_2 + 32 K_2) \\
&\quad - 64 (M_1' + 32 K_1) \times 64 (M_2' + 32 K_2) + 64 (M_1'' + 32 K_1) \times 64 (M_2' + 32 K_2) \\
&\quad - 64 (M'' + 32 K_1) \times 64 (M'' + 32 K_2) + 64 (M_1' + 32 K_1) \times 64 (M_2' + 32 K_2) \\
&= \text{NH}_{\text{others}} + 64 (M_1 + 32 K_1) \times 64 (M_2 + 32 K_2)
\end{align*}
\]

By this approach, the attacker not only clears the trace of past attack, but also forces the hash result to be valid for $M_1, M_2$ when the actual data should be $M_1', M_2'$. Also, even the attacker cannot control the new data. If he can predict that some value will appear before the next check, such as the $M_1', M_2'$ in the example, then he can simply modify the data and wait for the valid $M_1', M_2'$ to appear to finish the attack.

The outlined attack will compromise the inheritance property in the HHS schemes. However, if the data is selected uniformly at random, which means that the attacker can neither control nor predict the data, the inheritance property can still serve as a way to boost the performance. Normally the data in the memory is not random so that this property does not work. However, we can randomize the data by modifying the scheme. With randomization, even the check before every read operation can be eliminated. We achieve this by keeping random masks for every piece of data that needs to be randomized. We call these random masks randomizers. The randomizers can be generated either by a hardware random source or a PRNG. For each update of the Merkle Tree, instead of the data, the sum of the data and the corresponding randomizer is treated as the leaf of the tree. In addition, the randomizers will also change for each update. If the randomizers are changing randomly, the sum of the randomizer and the data should be also random. In this way, the error inheritance property works. These randomizers do not work as the masks for the hash results, which serves as one-time-pads. They are only used to randomize the data, not encrypt the data. Thus, as long as the randomizers are unpredictable, the error inheritance property will hold. Using the same example as above, the update process now will become:

\[
\begin{align*}
\text{NH} &= \text{NH}_{\text{others}} + 64 (M_1 + 32 R_1 + 32 K_1) \times 64 (M_2 + 32 R_2 + 32 K_2) \\
&\quad - 64 (M_1' + 32 R_1' + 32 K_1) \times 64 (M_2' + 32 R_2' + 32 K_2) + 64 (M_1'' + 32 R_1'' + 32 K_1) \times 64 (M_2' + 32 R_2' + 32 K_2) \\
&\quad - 64 (M'' + 32 R_1'' + 32 K_1) \times 64 (M'' + 32 R_2'' + 32 K_2) + 64 (M_1' + 32 R_1'' + 32 K_1) \times 64 (M_2' + 32 R_2'' + 32 K_2)
\end{align*}
\]

If the attacker can neither control nor predict the $R_1', R_2'$, he cannot perform the same attack over error inheritance as in the previous example. Obviously, the randomizers should also be protected from unauthorized modification. Otherwise, an attacker can compromise the system easily by forging the data and randomizers that sum to the same value. Another Merkle Tree can perfectly protect the randomizers. Since the randomizers are basically random data, they can benefit from the error inheritance property directly. Randomizing the data can enable the error inheritance and thus save the costly checks before every access. However, randomizing the data will introduce new overhead on both speed and storage space, mainly due to the generation of the randomizers and the maintaining the tree for the randomizers. The overheads can be so large that it will void the performance improvement provided by the elimination of the check before each access. Thus, we need extra optimization to the scheme to minimize the impact of the randomization process and build an even faster design. We will then introduce the basic ideas for the improvement and a scheme combining them together will be discussed in next section.

\(^2\)The error inheritance property does not hold for the HHS scheme. We will show that later.
4.2 Better Parallelism

In the proposed scheme, the operations related to the randomizers are mostly independent of the rest of the scheme. Thus, we can take advantage of the parallel computation and pre-computation to speed up the process. The generation of the randomizers and the operations to protect the integrity of the randomizers can all be done in parallel. The detail will be discussed in next section. Besides the parallelism introduced to lower the impact from the new randomizers, we also make improvements on the parallelism of the tree structure of the scheme. Computing the masks in advance or in parallel can greatly improve the performance as discussed in last section. In the HHS scheme, the mask is the only part that can be computed in parallel. In a Merkle Tree, the computation in higher level has to wait until the lower level finishing the update before it can start updating. In other words, higher levels of the tree depend on the results of lower levels. This dependence forms a slow path as the tree grows large and other costly computations are pre-computed. However, a small modification to the original HHS basic scheme can make the parallel computation for each level of the tree possible.

In our scheme, the tag for a block of data is actually composed of two parts, the tag itself and the seed. In the HHS scheme, the tags serve as the data for the higher level of the tree. The tags are computed from the data so that it will cause the problem mentioned above. However, if the data dependence is moved to the seeds instead of the tags at the higher levels of the tree, this restriction can be removed. The new seeds are selected completely at random so that they will not cause any dependence problems. Each level of the Merkle Tree can start updating without waiting for the result of the lower level. Thus parallel computation of the Merkle Tree can be employed. We will show later that the scheme remains secure with this modification.

4.3 An Alternative Caching Scheme

Using caching is an effective way to improve the performance of memory accesses. The HHS scheme also relies heavily on caching. The caches should be put in the protected memory otherwise the attacker may get the hash result in plain text and break the scheme based on it. In our new scheme, since the randomizers are protected by another Merkle Tree like system, a similar cache system can also help to speed up the operations. Furthermore, we can alter the caching scheme a bit to make it even better. In the HHS scheme, caching the masks in a secure cache can improve the efficiency of the scheme significantly without losing the security level. However, we can make it better by caching the hash result itself. In this case, no costly AES operation is needed as long as the hash result does not leave the cache. Thus, an even higher level of efficiency can be achieved.

Moreover, putting the hash results in a cache has some other useful benefits. Even the hash cache is used, the system still needs to recover the old mask when there is a miss. However, this operation can be delayed. The incremental update of the hash result and the encryption by the one-time-pad mask are both achieved by addition. Thus, one can actually update the tags without even unmasking the tags. In the HHS scheme this would not be secure since the update of hash results and changing of the masks should be done atomically or the security is compromised. However, with a hash cache, this operation becomes secure because the attacker can neither view nor change the tags in the caches. Although the final computation needed for unmasking cannot be eliminated, this property can help the scheme to achieve a good peak performance by delaying the costly AES operations.

5 Putting It All Together

By combining the methods introduced above we can achieve great performance improvements. Here we briefly outline the required modifications to the HHS scheme. First of all, the hash results need to be cached to improve the efficiency of the scheme and the new masks are pre-computed or computed in parallel. Let us assume that the masks can be prepared in parallel. The tags of the higher level of the Merkle Tree are calculated by hashing the seeds of the lower level and caches are also used to further improve the efficiency.

As outlined above, the data needs to be randomized to keep the error inheritance property. However, randomizers for every single byte of data will introduce a 100 percent overhead on storing space. It is similar to the problem of the caches that expensive but fast caches are limited. Thus, similar scheme may work. The solution is only using randomizers for the frequently updated data. Each data block is associated with
a randomizer when entering the cache, the randomizer is only assigned when the tag of that block of data is loaded into the cache. Though a large amount of computation has to be done when there is a cache miss, including a check of the data and the randomizers, the average speed will still be fast with a high cache hit rate.

Now when the tag of a block of data is loaded into cache, a block of randomizers are initialized to zero. When the update happens, the data plus the randomizer is treated as the data in the HHS scheme. If the randomizers are truly random, the data will be perfectly masked. This will enable the error inheritance property. In addition, reading the data can now be treated as updating the data to the same value. Since the randomizer will change even when the data remains the same, the sum of the data and the randomizer will still remain a random value. With this randomization, the error will be inherited with high probability.

The randomizers should also be protected by hashing them and saving the hash results in the caches. The randomizers are chosen completely at random. Thus the error inheritance property will perfectly work without any further change. When the block of data is about to be preempted from the cache, a check of the randomizers and the masked data is needed. Then the block of data will be hashed in the usual way and the hash result will be masked and saved to memory just as in the HHS basic scheme. The check is necessary because when the hash is calculated as usual, the data is no longer protected with randomizers. The data will become predictable and the error inheritance property is lost. Also, the randomizers have to change 128 bits for each update to achieve a 128-bit security level.

This design needs a large amount of extra protected (for the hash result for the randomizers) and unprotected (for the randomizers) memory space and will introduce some overhead. More computation units are also needed. However, we eliminate the check process before every read access to memory, which is much slower than the update operation. In addition, an efficient hardware random source can reduce the overhead significantly. Even when such hardware random source is not available, still, the randomizers can be pre-computed to improve the performance.

The higher levels of the Merkle Tree are built similarly using the seeds as data. They do not need such randomizers to maintain the error inheritance property because the seeds are essentially random values which are unpredictable. The resulting scheme is depicted in Figure 3 and the algorithm is given in Figure 4.

Figure 3: Outline of the new scheme
Step Initialization:
1. For each block of data $M_i$, generate a random seed $S_i$ and store it in the unprotected memory.
2. Encrypt the random seed $S_i$ using AES to obtain a mask $A_i$.
3. Calculate the hash value $H_i$ of each block $M_i$, add the mask $A_i$ to $H_i$ to obtain the tag $T_i$. Save $T_i$ in the memory.
4. Compute hash value $R$ of the tags and save it in the protected memory.
5. Clear the caches. Pre-compute whatever can be pre-computed.

Step Load hash into the cache($M_i$):
1. If necessary, unload some other block.
2. Load the tag of the data $H_i$ in the cache.
3. Begin to unmask the tag. (in parallel)
4. Set the randomizers for this block to all zeros.
5. Hash the randomizers and keep the hash result in cache.
6. Update the higher levels of the tree with a new seed.

Step Parallel operations:
1. Pre compute the $S'_i$, $R'_i$ and $A'_i$.
2. If any block in cache need unmasking,
   a) Read the corresponding seed.
   b) Update the higher levels with another seed.
   c) Use the read seed to unmask the tag.
3. If the higher levels of the tree need updating, update them.

Step Unload hash in the cache($M_i$):
1. Check whether the data meets the cached hash or not.
2. Hash the data without randomizers.
3. Take a new pair of pre-computed $S'_i$ and $A'_i$.
   add the mask $A'_i$ to $H'_i$ to obtain the new tag $T'_i$. Save $T'_i$ and $S'_i$.
4. Update the higher levels for the new seed.

Step Update($M_i$):
1. Visit the cache to find the hash $H_i$ corresponding to the block $M_i$.
2. If there is a miss, load the hash $H_i$ for this block.
3. Prepare a new randomizer $R'_i$.
4. Update the hash result incrementally.
5. Upgrade the hash result for randomizer $H_{Ri}$ incrementally.

Step Read($M_i$):
1. Just as update the data to the same value

Step Check($M_i$):
1. Check whether the block is in cache.
2. If there is a miss. Check as in the Basic Scheme. Except that the seeds other than the tags are checked.
3. If there is a hit.
   Hash the data with the randomizers and compare to the cached hash result.
   Hash the randomizers and compare to the cached hash result.
4. If any pair of them does not match, fail the check.
5. start checking higher levels as the basic scheme with caches.

Figure 4: Proposed Scheme

6 Security Analysis

The security analysis of the HHS basic scheme is provided in [14]. Here we will discuss the security implications of the proposed improvements.
6.1 Analysis for The Tree of Seeds

In the HHS scheme the Merkle Tree is formed by hashing tags. In the proposed modification the Merkle Tree is formed by seeds. We show that the security level is retained by giving a proof similar to that of the HHS scheme as follows.

**Theorem 1** Let $A$ be any adversary which has access to $q$ tuples $(M_i^1, M_i^2, R_i, \tau_i)$, where $q$ is a polynomial in $w$. Here we have $\tau_i = NH_{K_1}(M_i^1) | NH_{K_2}(M_i^2) + AES_{K_e}(R_i)$ where the addition is computed modulo $2^{2w}$ separately on each of the two parts of the AES$_{K_e}(R_i)$, $|$ is the string concatenation operator, and $K_1, K_2, M_i^1, M_i^2 \in \{0,1\}^{2w}$ such that $K_2$ is the shifted version of $K_1$. $K_e, R_i, \tau_i \in \{0,1\}^k$ where we have set $k = 4w$. For such an adversary we define $P_A$ to be the probability of $A$ finding $M_1^1, M_2^1$ and $\tau$ where $M_1^1 \neq M_2^1$ and $M_1^2 \neq M_2^2$ such that $\tau = NH_{K_1}(M_1^1) | NH_{K_2}(M_2^2) + AES_{K_e}(R)$ and $R = R_j$ for some $j \in [1, \ldots, q]$. Given the above, $P_A$ is bounded by

$$P_A \leq q(2^{-2w} + \text{Adv}_A^{AES}(4w))$$

**Proof** The adversary $A$ needs to find new messages which map to a tag with a same seed while it has no access to the hashing function nor to the AES encryption function. Therefore, $A$ needs to take advantage of the given $q$ tags. So let $A$ choose some tuple say $(M_i^1, M_i^2, R_i, \tau_i)$ where $t \in [1, \ldots, q]$ and $M_1^1 \neq M_2^1$ and $M_2^1 \neq M_2^1$. $A$ can now either fix $\tau_i$, or fix the messages $M_1^1, M_2^2$. Let us start by fixing $\tau_i$, in such a case $A$ needs to find an $M_1^1, M_2^1$ such that the tuple $(M_i^1, M_i^2, R_i, \tau_i)$ is valid. For this case $A$ needs to have

$$NH_{K_1}(M_1^1) | NH_{K_2}(M_2^2) + AES_{K_e}(R_i) = \tau_i$$

$$NH_{K_1}(M_1^1) | NH_{K_2}(M_2^2) + \tau_i - NH_{K_1}(M_1^1) | NH_{K_2}(M_2^2) = \tau_i$$

$$NH_{K_1}(M_1^1) | NH_{K_2}(M_2^2) - NH_{K_1}(M_2^1) | NH_{K_2}(M_2^2) = a$$

where $a = \tau_j - \tau_i$. The probability of finding such $M_1^1$ and $M_2^2$ is the $\varepsilon$-A$\Delta$U of the proposed scheme which happens to be $2^{-2w}$.

Alternatively, the adversary $A$ could also choose to fix the messages $M_1^1, M_2^1$ and find an appropriate $\tau$ such that the tuple $(M_1^1, M_1^2, R_j, \tau)$ is valid. For such a case $A$ needs

$$NH_{K_1}(M_1^1) | NH_{K_2}(M_2^2) + AES_{K_e}(R_j) = \tau$$

$$\tau_i - AES_{K_e}(R_i) + AES_{K_e}(R_j) = \tau$$

$$\tau - \tau_i = b$$

where $b = AES_{K_e}(R_j) - AES_{K_e}(R_i)$. The probability of finding such $\tau$ is equal to $\text{Adv}_A^{AES}(4w)$ the advantage of the adversary $A$ against the AES algorithm.

With $q$ possible pairs the probability of the adversary succeeding will at most increase $q$-fold. Therefore we still have

$$P_A \leq q(2^{-2w} + \text{Adv}_A^{AES}(4w))$$

The above theorem essentially reduces the security of the proposed scheme to that of the AES algorithm. □

6.2 Analysis of Error Inheritance

Adding a completely random mask to the data provides a one-time-pad like protection to the data. It is quite obvious that it should be secure. However, a formal proof is still given to strengthen the security claim.

The ability to incrementally update NH yields the error inheritance property which can roughly be stated as: wrong data-tag pairs will be updated to other wrong data-tag pairs during memory updates. If the data is changing randomly, it is safe to take advantage of such property to avoid checking the integrity of the data before each access. The random masks have nothing to do with this attack since they will cancel each other if the attacker did not modify them. If the attacker targets the masks, the security level will rely on the strength of AES as shown in the previous section. Alternatively, the attacker may attack the incremental hash process which is what we will focus on in this section. Remember that the incremental update process can be written as

$$\text{NH} = \text{NH}_{\text{others}} + 64(M_1 + 32 K_1) \times 64(M_2 + 32 K_2)$$

$$-64(M_1' + 32 K_1) \times 64(M_2' + 32 K_2) + 64(M_3'' + 32 K_1) \times 64(M_4'' + 32 K_2)$$

9
For an attacker to successfully compromise the incremental update, he needs to find some valid $M_1''', M_2'''$ so that

$$NH_{others} + \text{hash}(M_1''' + 32 K_1) \times \text{hash}(M_2''' + 32 K_2) = NH_{others} + \text{hash}(M_1 + 32 K_1) \times \text{hash}(M_2 + 32 K_2)$$

or

$$- \text{hash}(M_1 + 32 K_1) \times \text{hash}(M_2 + 32 K_2) + \text{hash}(M_1''' + 32 K_1) \times \text{hash}(M_2''' + 32 K_2)$$

where $M_1' \neq M_1, M_2' \neq M_2$ and the attacker can select $M_1, M_2, M_1', M_2', M_1''', M_2'''$ and $M_1'', M_2''$ are generated randomly. However, the attacker can only change $M$ and $M'$ before $M''$ is generated. In this way, the attacker can clear the trace of unauthorized modification, the $M'$. The equation can also be written as

$$NH(M''') - NH(M) = NH(M') - NH(M')$$

A proof with one level Toeplitz approach will be presented. It is easy to extend it to more levels of Toeplitz approach.

**Theorem 2** Let $A$ be any attacker to the incremental update. With a successful attack $A$ will have:

$$NH_{K_1}(M_1') | NH_{K_2}(M_2') - NH_{K_1}(M_1) | NH_{K_2}(M_2) = NH_{K_1}(R_1') | NH_{K_2}(R_2) - NH_{K_1}(M_1') | NH_{K_2}(M_2')$$

where the addition is done modulo $2^{2w}$ separately on each of the two parts, $|$ is the string concatenator operation, and $K_1, K_2, M_1', M_2', R_1, R_2 \in \{0, 1\}^{2w}$ such that $K_2$ is the shifted version of $K_1$. $R_1', R_2$ are random numbers being generated after $M_1', M_2', M_1, M_2$ are chosen. For such an adversary we define $P_A$ to be the probability of $A$ finding $M_1', M_2', M_1, M_2$ where $M_1' \neq M_1$ and $M_2' \neq M_2$ such that the above equation is satisfied. Given the above, $P_A$ is bounded by $P_A \leq 2^{-2w}$.

**Proof** To make the error checking equation work, the attacker may fix the left part and try to find the proper values in the right or vice versa. If the attacker fixes the right part, the problem is reduced to trying to find $M_1', M_2'$ such that

$$NH_{K_1}(M_1') | NH_{K_2}(M_2') - NH_{K_1}(M_1) | NH_{K_2}(M_2) = a$$

where $a$ is some constant. If $M_1' \neq M_2'$ and $M_2' \neq M_2$, since the NH is $2^{-2w}$-almost-$\Delta$-universal as mentioned above, the probability to find the proper $M_1', M_2'$ is $2^{-2w}$. The attacker may alternatively choose $M_1' = M_2'$ and $M_2' = M_2'$ and try to make the $a$ to be zero. This is a special case of fixing the left part and trying to find the proper right part. If the attacker choose to do so, the problem will become trying to find $M_1', M_2'$ such that

$$NH_{K_1}(R_1') | NH_{K_2}(R_2) - NH_{K_1}(M_1') | NH_{K_2}(M_2') = a$$

where $a$ is some constant. Similarly, the attacker may either let $a$ be zero and try to make $M_1' = R_1$ and $M_2' = R_2$ or try to find some $M_1', M_2' \neq R_1$ and $M_2' \neq R_2$ to meet the equation where $a$ can be any value. If the attacker tries to force $M_1' = R_1$ and $M_2' = R_2$, since $R_1, R_2$ are random numbers generated after $M_1', M_2'$ are chosen, the probability for that $R_1 = M_1', R_2 = M_2'$ is $2^{-2w}$. If the attacker chooses the other way, the problem is again reduced to the $2^{-2w}$-almost-$\Delta$-universal property of NH. Therefore, the error inheritance property will be compromised with probability at most $P_A \leq 2^{-2w}$. 

\[\square\]

### 6.3 Birthday Attacks

In the traditional applications in the message integrity protection codes. We expect the security level to be half of the tag size due to a birthday attack. That is, 32-bit security level for a 64-bit tag. However, the birthday attack will not reduce the security level in our scheme.

The tag $\tau$ in our scheme is computed by adding the mask to the hash result:

$$NH_{K_1}(M^1) | NH_{K_2}(M^2) + AES_{K_e}(R) = \tau$$

To perform a birthday attack on our scheme. The attacker may either try to find any two messages that map to the same hash result $NH_{K_1}(M^1) | NH_{K_2}(M^2)$ or try to find two messages that map to the same tag $\tau$. If the attacker chooses to attack the hash result, since the hash result is protected by one-time-pad, the attacker cannot even check for a collision. In the case that the attacker want to find a collision over $\tau$, the birthday attack applies. If the word size used is $w$, the size of the tag $\tau$ will be $2w$. The probability of a collision will become $2^{-w}$ instead of $2^{-2w}$ under a birthday attack. However, the collision probability of the hash function is also $2^{-w}$ in the beginning. Thus the birthday attack will not provide a more efficient way to compromise the scheme than trying to find collisions of the NH directly.
7 Performance

In this section, we will briefly analyze the performance of the proposed scheme. We will discuss the overhead introduced by the scheme and the protected and unprotected storage space demand. In addition, some test results collected from a software implementation of the scheme is presented to evaluate the performance in practice.

7.1 Performance Analysis

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M$</td>
<td>4G</td>
<td>size of memory to protect in bytes</td>
</tr>
<tr>
<td>$S$</td>
<td>32</td>
<td>size of seed in bytes</td>
</tr>
<tr>
<td>$T$</td>
<td>32</td>
<td>size of tag in bytes</td>
</tr>
<tr>
<td>$B$</td>
<td>changing</td>
<td>block size in bytes</td>
</tr>
<tr>
<td>$R$</td>
<td>1</td>
<td>number of roots, or number of trees used</td>
</tr>
<tr>
<td>$l$</td>
<td>changing</td>
<td>number of levels</td>
</tr>
<tr>
<td>$t_{AES}$</td>
<td>30</td>
<td>time for AES encryption in cycles per byte</td>
</tr>
<tr>
<td>$t_h$</td>
<td>2</td>
<td>time for hashing in cycles per byte</td>
</tr>
<tr>
<td>$t_u$</td>
<td>100</td>
<td>time for incremental hashing in cycles/two words</td>
</tr>
<tr>
<td>$t_r$</td>
<td>1.5</td>
<td>time to generate a random seed in cycles/byte</td>
</tr>
<tr>
<td>$H$</td>
<td>0.8</td>
<td>hit rate of the mask cache</td>
</tr>
<tr>
<td>$C$</td>
<td>0.2</td>
<td>percentage of masks (or hash result) cached</td>
</tr>
</tbody>
</table>

Table 1: Variables used in the analysis

In this section the memory and time requirements for the proposed schemes are estimated. The parameters used in the analysis are defined in Table 1. They are the same as that used in [14] so that we can compare the performance with the HHS scheme. The scheme with this parameter choice will have a $2^{-128}$ collision probability. Note in the table that due to the structure of NH, an update will involve at least two words. Even only one word is updated, 100 cycles are needed instead of 50 cycles. However, if two continuous words are going to change together, such as the update of the higher levels of the tree, 100 cycles are enough per two words.

From the derivations in [14] we can derive that the time needed per update $t_{up}$ for the scheme with mask caches and the time needed per check $t_{check}$ will be

$$t_{up} = t_u + (l-1)[\frac{t_u T}{8} + (1-H)t_{AES}T + t_{AES}T + t_r S]$$ \hspace{1cm} (1)

$$t_{check} = t_h B + (1-H)\frac{1-(1-H)^{l-1}}{H}(t_h B + t_{AES} T)$$ \hspace{1cm} (2)

If the hash result is cached directly, there is no need for a new mask. Following a similar procedures, the time needed will be

$$t'_{up} = t_u + (1-H)\frac{1-(1-H)^{l-1}}{H}\left[\frac{t_u T}{8} + 2t_{AES} T + t_r S\right]$$ \hspace{1cm} (3)

$$t'_{check} = t_h B + (1-H)\frac{1-(1-H)^{l-1}}{H}(t_h B + t_{AES} T)$$ \hspace{1cm} (4)

However, the data needs to be checked before each update. The seeds are random and the tags are protected by the one-time-pad. Thus, the higher levels of the tree can benefit from the error inheritance property and do not have to be checked all the time as explained before. Also, since the check and update process have some shared operations, such as unmasking, the time needed for an update following a check will be

$$t_{check\&up} = t_u + t_h B + (l-1)[\frac{t_u T}{8} + (1-H)(t_{AES} T + t_h B) + t_{AES} T + t_r S]$$ \hspace{1cm} (5)
Also from [14], we obtain the size of the unprotected memory as

\[ t'_{\text{check\&up}} = t_u + t_h B + (1 - H) \frac{1 - (1 - H)^{l-1}}{H} \left( \frac{t_u T}{8} + t_h B + 2 t_{AES} T + t_r S \right) . \]  

(6)

Also from [14], we obtain the size of the unprotected memory as

\[ M_u = M'_u = \frac{B}{T} \left( \frac{1 - \left( \frac{H}{B} \right)^{l-1}}{1 - \left( \frac{H}{B} \right)} \right) (T + S) R . \]  

(7)

The cached masks (or hash results), the roots of the trees and the keys for the universal hash (the key for AES and the extra hash key introduced by Toeplitz approach are omitted since it is too short) will all need to be stored in protected memory. Thus the protected memory needed will be

\[ M_p = M'_p = \left[ \frac{B}{T} \left( \frac{1 - \left( \frac{H}{B} \right)^{l-1}}{1 - \left( \frac{H}{B} \right)} \right) TC + T \right] R + B . \]  

(8)

For the scheme with randomizers, the time needed for generating a seed will be longer because a secure random number is needed. The speed of hash functions with randomizers will not change significantly since it does not introduce much extra computation. Thus the time needed for generating random numbers will be approximately \( t'_r = t_{AES} \) which is 30 cycles per byte. With such settings, these values will be

\[ t''_u = \frac{t_u T}{8} + \frac{t'_r T}{2} + (1 - H)(3 t_h B + 2 t_{AES} T + t'_r S + \frac{t_u T}{4}) + 2(1 - H)^2 \frac{1 - (1 - H)^{l-2}}{H} \left( \frac{t_u T}{8} + t_h B + (2 t_{AES} T + t'_r S) \right) . \]  

(9)

\[ t''_{\text{check}} = (1 + H) t_h B + (1 - H) \frac{1 - (1 - H)^{l-1}}{H} (t_h B + t_{AES} T) . \]  

(10)

\[ M''_u = \frac{B}{T} \left( \frac{1 - \left( \frac{H}{B} \right)^{l-1}}{1 - \left( \frac{H}{B} \right)} \right) (T + S) R + MC . \]  

(11)

\[ M''_p = \left[ 2 \frac{B}{T} \left( \frac{1 - \left( \frac{H}{B} \right)^{l-1}}{1 - \left( \frac{H}{B} \right)} \right) TC + T \right] R + B . \]  

(12)

where \( l \geq 2 \) since less than two levels will be meaningless in the scheme.

The scheme with randomizers does show some advantages over the ones without them. However, when the parallelism comes into effect, the result can be even better. If we assume that the pre-computation and parallelism work perfectly, which means that the system can get the required values which can be separately prepared at once, such as the random numbers, the schemes can be speed up dramatically. Suppose that the caches are hit, all the pre-computed values are always ready to serve and the parallel computation units are always available, the peak "check and update" time for the schemes will become

\[ t_{\text{check\&up}} = \max[t_h B, t_u + (l - 1) \frac{t_u T}{8}] . \]  

(13)

with mask caches and \( t'_{\text{check\&up}} = \max[t_h B, t_u] \) with hash result caches. The update time needed for the scheme with randomizers will become \( t''_u = \frac{t_u T}{4} \).

These results are affected by the block size. If the scheme covers the whole memory, then the block size, the number of levels and the size of the memory must satisfy the following equation:

\[ M = R \left( \frac{B}{T} \right)^{l-1} B . \]  

(14)

With this equation, we can then calculate the proper block size for different number of levels. Then using the block size, we can calculate the speed and the memory consumption of the Merkle Tree structure for different choice of levels.
The speed of the outlined schemes when caches are employed are calculated and listed in Table 2. We can see clearly from the table a significant peak performance improvement in the scheme with randomizers. When parallelism is employed, this improvement will become even larger. The average performance for the schemes without randomizers will never be better than $t_{LU}$ only when there is a cache miss.

As shown in Table 4, the unprotected memory needed for the scheme with randomizers is huge due to the extra demand for storing randomizers. However, the cache will be much smaller in practice (consider 2MB CPU L2 cache for GBs of memory) so that this value will decrease a lot. If only one percent of the hash results will be cached instead of twenty percents in the original settings, the unprotected memory demand will drop significantly to about 240 MB.

### Table 2: Speed Comparison (cycles)

<table>
<thead>
<tr>
<th>$l$</th>
<th>$t_{check_{up}}$</th>
<th>$t_{check}$</th>
<th>$t_{peak_{up}}$</th>
<th>$t_{check_{up}}$</th>
<th>$t_{check}$</th>
<th>$t_{peak_{up}}$</th>
<th>$t_{check_{up}}$</th>
<th>$t_{check}$</th>
<th>$t_{peak_{up}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>743155</td>
<td>889938</td>
<td>741455</td>
<td>890320</td>
<td>889939</td>
<td>741455</td>
<td>446489</td>
<td>1483102</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>36068</td>
<td>40862</td>
<td>32768</td>
<td>41301</td>
<td>40862</td>
<td>32768</td>
<td>23584</td>
<td>67077</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>11789</td>
<td>8835</td>
<td>6889</td>
<td>9284</td>
<td>8835</td>
<td>6889</td>
<td>6530</td>
<td>14346</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>9202</td>
<td>3616</td>
<td>2702</td>
<td>4068</td>
<td>3616</td>
<td>2702</td>
<td>3712</td>
<td>5778</td>
<td>400</td>
</tr>
</tbody>
</table>

### Table 3: Speed Comparison II (cycles)

<table>
<thead>
<tr>
<th>$l$</th>
<th>$t_{check_{up}}$</th>
<th>$t_{check}$</th>
<th>$t_{peak_{up}}$</th>
<th>$t_{check_{up}}$</th>
<th>$t_{check}$</th>
<th>$t_{peak_{up}}$</th>
<th>$t_{check_{up}}$</th>
<th>$t_{check}$</th>
<th>$t_{peak_{up}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7034100</td>
<td>7034100</td>
<td>7034100</td>
<td>1485378</td>
<td>1483870</td>
<td>1485378</td>
<td>446489</td>
<td>1483102</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>431200</td>
<td>431200</td>
<td>431200</td>
<td>103140</td>
<td>100224</td>
<td>103140</td>
<td>23584</td>
<td>67077</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>116200</td>
<td>116200</td>
<td>116200</td>
<td>34758</td>
<td>30434</td>
<td>34758</td>
<td>6530</td>
<td>14346</td>
<td>400</td>
</tr>
<tr>
<td>5</td>
<td>55700</td>
<td>55700</td>
<td>55700</td>
<td>23084</td>
<td>17352</td>
<td>23084</td>
<td>3712</td>
<td>5778</td>
<td>400</td>
</tr>
</tbody>
</table>

### Table 4: Space Demand (MB if not specified)

<table>
<thead>
<tr>
<th>$l$</th>
<th>$M_p$</th>
<th>$M_u$</th>
<th>$M_p$</th>
<th>$M_u$</th>
<th>$M_p$</th>
<th>$M_u$</th>
<th>$M_p$</th>
<th>$M_u$</th>
<th>$M_p$</th>
<th>$M_u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20 B</td>
<td>0.3</td>
<td>35.4 KB</td>
<td>0.71</td>
<td>0.42</td>
<td>0.71</td>
<td>0.42</td>
<td>0.71</td>
<td>0.50</td>
<td>819.9</td>
</tr>
<tr>
<td>3</td>
<td>20 B</td>
<td>7.2</td>
<td>15.7 KB</td>
<td>16.0</td>
<td>1.62</td>
<td>16.0</td>
<td>1.62</td>
<td>16.0</td>
<td>3.22</td>
<td>835.2</td>
</tr>
<tr>
<td>4</td>
<td>20 B</td>
<td>35.8</td>
<td>3.32 KB</td>
<td>76.8</td>
<td>7.69</td>
<td>76.8</td>
<td>7.69</td>
<td>76.8</td>
<td>15.4</td>
<td>896.0</td>
</tr>
<tr>
<td>5</td>
<td>20 B</td>
<td>94.6</td>
<td>1.32 KB</td>
<td>198.7</td>
<td>19.9</td>
<td>198.7</td>
<td>19.9</td>
<td>198.7</td>
<td>39.7</td>
<td>1017.9</td>
</tr>
</tbody>
</table>

The estimation for the basic scheme and SHA-1 based Merkle Tree are also present in Tables 3 and 4 for comparison. The SHA-1 speed is set to 12 cycles per byte, which is collected from an optimized code. $T$ is set to 20 since the output size of SHA-1 is 160 bits which is 20 bytes. The unit for $t_{up}$ and $t_{check}$ is cycles. Since the basic scheme and SHA-1 based Merkle Tree scheme do not involve any cache or parallelism, the peak performance will be just the same as the average case. Note from the tables that on one hand the SHA-1 based scheme has a low unprotected memory requirement and an almost negligible protected memory requirement. On the other hand, it is much slower than our scheme.

### 7.2 Implementation Results

To evaluate the performance we implemented the proposed schemes on an Intel Core 2 machine with a processor speed of 1.67 G Hz. We developed a simple two level Merkle Tree with combinations of the improvements we proposed. The size of memory to be protected was 32KB. In our experiments, we used a block size of 1KB and a tag size of 256-bit (32B). The cache size is four tags and AES are computed in
parallel. We also simulated the HHS Basic Scheme and a SHA-1 based Merkle Tree for comparison. The SHA-1 based Merkle Tree will produce 160-bit tags which is shorter than that of our scheme. However, it will also provide a lower resistance against the birthday attack, that is, 80-bit compared to the 128-bit.

To evaluate the performance of the schemes in different situations, we measure the speed of our scheme with different cache hit rates. First we tested the speed of the Update and Check functions of our scheme when the cache access is always a hit. This is achieved by letting the application access the same location of the memory so that there will always be a hit. For comparison, the speed when the cache access is always a miss is also determined. We also measured the performance of the HHS basic scheme and a SHA-1 based Merkle Tree. Since they do not use caches, their speeds do not change whether the cache access is a hit or miss.

![Performance Comparison](image)

Figure 5: Performance Comparison

Figure 5 shows the overhead in cycles of the final scheme with the randomizers, the scheme with a hash cache, the scheme with a mask cache, the HHS basic scheme and the SHA-1 based Merkle Tree. For each scheme, the speed of an Update, a Check and a Check followed by an Update when the cache access is a hit or miss is provided. As shown in the figure, the scheme with the hash cache can even reduce the peak (cache hits) update time to less than 100 cycles. When the attacker can neither control nor predict the data, such speed is quite promising. However, in the general case, an additional check is needed before each access. the scheme with randomizers performs better in these cases since the error inheritance property can save from the costly check operation before every access. When the cache hit rate is high, our new scheme shows a great advantage.

We also measure the average case performance for different cache hit rates. This is achieved by letting the application access memory locations every several words. When this stride, i.e. the number of words, grows the cache efficiency will get worse. In the worst case, when the stride is larger than a cache line, each access will always skip the data loaded into the cache due to a previous access, which results in a cache miss. Figure 6 shows the performance of each scheme vs. this stride. We can see clearly from the figure that the performance of the new schemes depends heavily on the cache hit rate. The scheme with randomizers performs best with high hit rates while suffering most from low cache hit rates. This is because our scheme can benefit from the error inheritance property when there is a cache hit. However, in our scheme when the tags are going to leave the cache, a check is still needed and the new randomizers need to be prepared.
which will cause more overhead than the HHS basic scheme. Thus, it is not a good choice to implement the scheme with randomizers when the cache hit rate is low. However, in most practical cases such as the RAM integrated in a PC, the hit rate should be quite satisfactory considering the CPU cache hit rate.

In general, we can conclude that in most practical cases, where the HHS scheme needs checking before each access, the new proposed schemes have a significant improvement in performance compared to the schemes proposed in [14]. When the cache efficiency is high, the improved scheme shows great advantages due to the error inheritance property. With the settings used in the evaluation, the peak performance can be about 5 times better than the HHS scheme and about 55 times faster than the SHA-1 based scheme.

8 Conclusion and Future Work

In this work we proposed a number of improvements for the dynamic memory protection scheme proposed in [14]. The new scheme can achieve better performance via more efficient caches and better use of parallelism as well as the error inheritance property. The error inheritance property was first informally introduced in [14]. However, we presented an attack that shows that the scheme in [14] lacks this property. In our paper, we prove that the new scheme does carry the error inheritance property. Thus, costly check operations are no longer needed before each memory access. When the proposed techniques are simultaneously implemented, the overall performance can be improved significantly. Our implementations show that the peak speed is increased by up to a factor of 5 over the scheme proposed in [14], at the expense of more protected and unprotected memory used to maintain the caches.

We set the integration of the proposed memory integrity protection scheme in a real life application as future work. This will allow us to get a better sense of the memory access and cache performance and to resolve the bottlenecks more effectively.

References


