An Efficient Scheme to Provide Real-time Memory Integrity Protection

by

Yin Hu

A Thesis
Submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the
Degree of Master of Science

in

Electrical and Computer Engineering

by

May 2009

APPROVED:

Professor Berk Sunar Thesis Advisor ECE Department	Professor Wenjing Lou Thesis Committee ECE Department
Professor Kathryn Fisler Thesis Committee Computer Science Department	Professor Fred J. Looft Head of Department ECE Department

Abstract

Memory integrity protection has been a longstanding issue in trusted system design. Most viruses and malware attack the system by modifying data that they are not authorized to access. With the development of the Internet, viruses and malware spread much faster than ever before. In this setting, protecting the memory becomes increasingly important. However, it is a hard problem to protect the dynamic memory. The data in the memory changes from time to time so that the schemes have to be fast enough to provide real-time protection while in the same time the schemes have to use slow crytographical functions to keep the security level.

In this thesis, we propose a new fast authentication scheme for memory. As in previous proposals the scheme uses a Merkle tree to guarantee dynamic protection of memory. We use the universal hash function family NH for speed and couple it with an AES encryption in order to achieve a high level of security. The proposed scheme is much faster compared to similar schemes achieved by cryptographic hash functions such as SHA-1 due to the finer grain incremental hashing ability provided by NH. With a modified version of the proposed scheme, the system can access the data in memory without checking the integrity all the time and still keeps the same security level. This feature is mainly due to the incremental nature of NH. Moreover, we show that combining with caches and parallelism, we can achieve fast and simple software implementation.

Acknowledgements

Foremost, I would like to thank Prof. Berk Sunar for providing me with the opportunity to be in the CRIS Lab. I am more than grateful to him for his continuous support since the very beginning, in terms of technical and personal advice,guidance and inspiration. I feel really lucky that I have the chance to work with such a great person in the future, too.

In addition, I want to thank Ghaith Hammouri for his invalunable help in writing Chapter 5.

Also, I am glad to have had Prof. Wenjing Lou and Prof. Kathryn Fisler in my thesis committee. I appreciate their advice and feedback in spite of tight schedules.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this chapter is to give a general idea of the goal of the scheme proposed in this thesis. Some previous works in this field will also be introduced.

1.1 Motivation

Memory integrity protection has been a longstanding issue in trusted system design. Most viruses and malware attack the system by modifying data that they are not authorized to access. With the development of the Internet, viruses and malware spread much faster than ever before. In this setting, protecting the memory becomes increasingly important.

Modern operating systems (OSs) achieve memory space isolation, i.e. preventing one process to access the data of another process, by using virtual memory. Ideally, this can protect the system against attacks from software. Even if perfect process memory space isolation is achieved attack techniques such as cross-site scripting — a website attacking another website running under the same memory space of the client browser — still pose a serious threat. Moreover, OSs 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.

Even further, in distributed applications the owner of the device has incentives to break the rules setup by the application server. Usually, the OS does little to limit the behavior of the computer owner. For instance, the owner can tamper with the client of an e-cash system running on his own machine. Similarly, the owner may want to cheat in an online game to gain a financial advantage. These scenarios make it clear that companies will have to find a way to protect the clients against privileged user of the OS.

The fundamental problem here the lack of a trusted root where all the protection based on. Obviously, data in the memory can never be protected by a program whose codes is stored in the same memory without further protection. Fortunately, hardware today can be made extremely hard to compromise, though also quite expensive, which can serve as a root of trust for various protection schemes. For instance, the Trusted Platform Modules (TPMs) are such chips and are installed in most new computers.

However, the secure hardware is expensive and will remain relatively high cost in the foreseeable future. Thus the secure storage space and computation power is limited. Carefully designed schemes is needed to bridge the gap between the limitation and the large open memory.

1.2 Previous Work

Many schemes have been proposed trying to provide protection to the large unsecured memory with high speed using the secure however limited space and computation power provided by hardware or maybe carefully designed software if the secure requirement is not so high.

In [5] Gassend et. al proposed a data authentication scheme based on Merkle trees [7]. To authenticate an arbitrarily large untrusted RAM, an m-array tree is used. Each node in the tree will essentially be a collision resistant hash of the following m children-nodes. The leaves at the end of the tree will contain the actual data. The root of the entire tree is securely stored in an on-chip trusted register of constant size. All other nodes are kept in main memory or cache. To authenticate a block of data (or an inner node) all hash values starting from the level following the data to the final level containing the root of the hash-tree are computed and compared to the stored values. If a mismatch is detected in any of the checked nodes the integrity check fails. To update the data, each hash value starting from the level following the data to the root of the tree has to be recalculated and restored. It is straight forward to see that this scheme is secure under the assumption that the hash function used is collision free. Any modification to the memory will eventually require a corresponding modification to the root of the tree which is kept in a secure register. This means that no un-authenticated modifications can take place without detection. The worst case time complexity of each read or write operation will be $O(\log_m(N))$ (for balanced hash-trees), where N is the size of the memory. This performance can be improved by observing that the nodes which are already in a trusted L1 cache do not need to be re-authenticated. Re-authentication is only performed on dirty cache-lines when the cache-lines are flushed to memory. With

this observation the memory overhead of the scheme will be $1/(m-1)$.

In [4] Clarke et. al proposed a scheme which keeps the hash values of the logs of all read and write operations. The memory can then be authenticated off-line by comparing the logs with the actual content of the memory. This scheme is inspired by the incremental hashing proposed by Bellare, Goldreich, and Goldwasser [8]. In their work Gassend et. al introduced the new concept of *multiset hash functions*, which are families of hash functions that map a multiset (a set with possible repetitions) to a constant size hash value. They give three constructions of multiset hash functions which are incremental — that is: given two multisets and their hash values, it is efficient to compute the hash value of the multiset-union of the two multisets. In the context of data authentication, the goal is to keep logs of all read and write operations. These log entries contain address, data, and time stamp triples. Since the logs can become arbitrarily large, only two multiset hashes are kept: one for the read operation and one for write operations. When the memory is authenticated, *all* memory locations are compared with the logs. This gives a very inefficient authentication process requiring a running time of $O(N)$. However, read and write operations require only a small constant running time (the time to add entries to the read and write logs). The only memory requirements for this scheme are the two multiset hash values of the logs. Since authentication is very expensive, it is only performed when the data is exported out of the program's execution environment.

In [6] Clarke et. al proposed a hybrid protocol with the aim of balancing the drawbacks and benefits of the two protocols described in [4] and [5]. The overhead of this scheme becomes a constant as the number of instructions between critical regions grows. To detect unauthorized modification to a memory, one may keep a hash result (a tag) of each block of data. As long as the hash function is secure, the attacker cannot forge the tag of the modified data. However, such a scheme

suffers from the so-called *replay* attacks. The attacker can overwrite the data and the corresponding tag with the ones that appeared in an earlier version of the same memory block. This way, an attacker can roll-back old data without being noticed. A common method to protect against replay attacks is to add time stamps to the tags. However, the system has to keep a timer for each block of data in a protected part of memory to determine whether the tag is valid or not. The protected memory requirement becomes excessive especially with growing number of tags. To reduce the amount of secure storage the use of Merkle trees was proposed in [7]. A Merkle tree can provide defense against replay attacks as long as the attacker cannot access the root and replay it. Any unauthorized replay of the tags (which will be a child node of the root) can be detected. Another feature of the Merkle tree structure is that it only needs a small space of secure storage, which makes it more practical than adding time stamps to the tags. However, the extra hash operations needed for the multi-level tree structure will slow down the speed of the scheme significantly. Even using the popular SHA-1, which is usually considered fast, in the Merkle tree scheme will cause large overhead.

In [1] Yan proposed a scheme for memory encryption. In their work, they used the GHASH combined with Merkle trees to provide authentication as part of the memory encryption scheme. GHASH is a kind of hash function based on Galois counter Mode of operations. It was proposed in [27] by McGrew and Viega. It runs faster than SHA-1 and can be pipelined.

The goal of the scheme proposed in this thesis is similar. The scheme can protect a large memory space with only a small amount of secure storage space and will only introduce a mild computation overhead. It is not about building a completely new protocol, but rather trying to find an efficient combination of existing methods to solve a really difficult problem.

In [11] McCune et al. proposed an architecture called "Flicker" that provides trusted computation based on TPMs. With such design, it is possible to execute secure codes even with a compromised OS. It may provide a tool to implement our scheme in software.

In [12] Handschuh proposed some attacks over UMAC which uses the universal hash function called NH. The same hash function is used in our scheme. However, the attack just offers a new way to compromise the hash function with the probability no higher than the collision probability. The security level of the hash function is not reduced and this attack can be also be avoided with a carefully designed scheme. Actually, our scheme is not affected by this new attack since the tags in our scheme are protected by the one-time-pad.

1.3 Thesis Outline

The rest of the thesis will be organized as the following.

In Chapter 2, there is brief introduction about the Merkle tree, the universal hash function and some other concepts will be referred in the following part of the thesis. These ideas are the building block of the proposed scheme.

In Chapter 3, the basic idea of the scheme will be described following making some important assumption about the security settings.

In Chapter 4, some possible improvements on the basic idea are introduced. With these ideas, the speed of the scheme can be further improved.

In Chapter 5, the proof of the security level of the proposed scheme is provided. We can see that the scheme is secure enough for practical use.

In Chapter 6, some performance analysis of the scheme will be provided followed by some test results of the scheme. A comparison with the popular SHA-1 Merkle

tree scheme will also be provided.

In Chapter 7, a summary of the works is given with some discussion about the possible future research topics.

Chapter 2

Background

In this chapter, the building blocks of the scheme will be introduced. It will help the readers to better understand the scheme in the following chapters. Firstly we will introduce the concept of the universal hash function and the family of universal hash functions we will used in our scheme to provide the message authentication codes. Secondly we will introduce the Toeplitz approach which will be used to improve the security of our scheme. Finally a brief introduction about the Merkle tree will be given.

2.1 Universal Hash Function

Roughly speaking, universal hash functions are collections of hash functions that map messages into short output strings such that the collision probability of any given pair of messages is small. A universal hash function can be used to build an unconditionally secure MAC. To achieve this, the communicating parties share a secret encryption key, and a random hash function secretly chosen from the universal hash function family. A message is authenticated by hashing it with the shared hash function and then encrypting the resulting hash using the shared key. Carter and

Wegman [25] showed that when the hash function family is strongly universal, i.e. messages are mapped into their images in a pair wise independent fashion, and the encryption is realized by a one-time pad, the adversary (even with unbounded computational power) cannot forge the message with probability better than that obtained by choosing a random string for the MAC.

A universal hash function, as proposed by Carter and Wegman [24], is a mapping from the finite set A with size a to the finite set B with size b. For a given hash function $h \in H$ and for a message pair (M, M') where $M \neq M'$ the following function is defined: $\delta_h(M, M') = 1$ if $h(M) = h(M')$, and 0 otherwise, that is, the function δ yields 1 when the input message pair collide. For a given finite set of hash functions $\delta_H(M, M')$ is defined as $\sum_{h \in H} \delta_h(M, M')$, which tells us the number of functions in H for which M and M' collide. When h is randomly chosen from H and two distinct messages M and M' are given as input, the collision probability is equal to $\delta_H(M, M')/|H|$.

We next quote a number of definitions concerning the universal hash function family. The following two definitions were introduced in [21].

Definition 1 *The set of hash functions $H = h : A \rightarrow B$ is said to be* ***universal*** *if for every $M, M' \in A$ where $M \neq M'$,*

$$|h \in H : h(M) = h(M')| \ = \delta_H(M, M') = \frac{|H|}{b} \ .$$

Definition 2 *The set of hash functions $H = h : A \rightarrow B$ is said to be ε-**almost universal** if for every $M, M' \in A$ where $M \neq M'$,*

$$|h \in H : h(M) = h(M')| \ = \delta_H(M, M') \leq \varepsilon |H| \ .$$

In the past many universal and almost universal hash families were proposed [22, 16, 23, 19, 9, 15]. In [9] Black et al introduced an almost universal hash function family called NH. The definition of NH is given below.

Definition 3 *([9]) Given $M = (m_1, \cdots, m_n)$ and $K = (k_1, \cdots, k_n)$, where m_i and $k_i \in U_w$, and for any even $n \geq 2$, NH is computed as follows:*

$$\begin{aligned} \mathsf{NH}_K(M) &= \left[\sum_{i=1}^{n/2} ((m_{2i-1} + k_{2i-1}) \bmod 2^w) \cdot \right. \\ &\qquad \left. ((m_{2i} + k_{2i}) \bmod 2^w)\right] \bmod 2^{2w} \ . \end{aligned}$$

In the same paper NH was shown to be 2^{-w}-almost universal which results in a collision probability of 2^{-w}. It can be further improved by the Toeplitz approach, which will be introduced in next section.

2.2 Toeplitz Approach

The technique of multi-hashing was introduced by Rogaway in [23] to increase the security level of a given hash function without changing the size of the hash value at the expense of more key material. The technique concatenates the hash results of the same data hashed with different keys. While the collision probability of the resulting scheme is immediately improved, i.e. if the collision probability of a universal hash is 2^{-w}, concatenating t such results obtained under t uniformly random keys gives a collision probability of 2^{-wt} at the expense of t times more key material. Later on, Krawczyk showed that under certain conditions [18] the key requirement may be reduced by shifting the keys in each multi-hash iteration. In Krawczyk's construction the hash is obtained via a binary matrix-vector product. Therefore the shifted keys yield a Toeplitz matrix. In [13] Kaps et al. observed that

the multi-hashing idea together with the Toeplitz technique may be used to gain computational efficiency. They reduce the word-size of the hash function t-times and concatenate t hashes together to achieve the same collision probability as of the original hash function. Hence, the security level is kept intact at the cost of a slight increase in the key-size. In this work, we make use of the same ideas. That is, we use multi-hashing to amplify the security level while performing all operations with 32-bit words which map more naturally and efficiently to existing processor instructions.

2.3 Merkle Trees

A Merkle Tree is a tree structure that uses hash functions to protect data. It was first proposed by Merkle in [2]. As shown in 2.1, the leaf nodes of the tree is the data and the branch nodes of the tree is the hash results of blocks of data or hash results of lower levels.

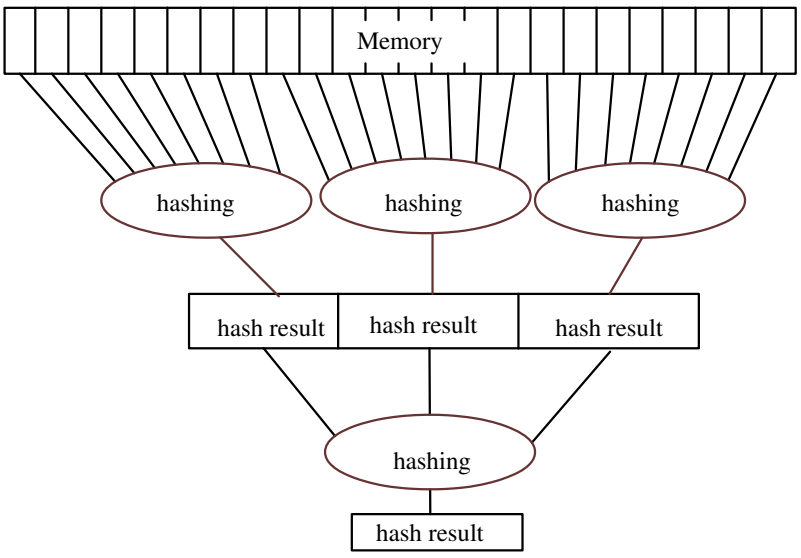

Figure 2.1: The Merkle Tree

It is easy to see that if it is hard for the attacker to find a collision of the used hash function, which can be achieved by using some proved secure hash functions, the only way to perform unauthorized modification to the data is to change the branch nodes of the tree as well. However, if the attacker changes the branch nodes, he will have to change the higher levels of the tree as well to hide his attack. Ultimately, the attacker will have to change the root of the tree to finish a successful attack. If the root of the tree is kept in some safe place, provided by hardware for instance, the data is well protected by the Merkle Tree.

In the proposed scheme, the NH mentioned before will be used to form a Merkle Tree other than using other popular hash functions, such as SHA-1.

Chapter 3

Basic Scheme

The basic scheme will be described in this chapter. We will outline the assumptions and define the attacker model to establish the security level of the scheme.

3.1 Attacker Model and Assumptions

As discussed in the previous chapters, the computer memory is vulnerable. The attacker may gain great benefits from unauthorized access. For instance, while a trusted process is accessing a bank account, the malicious code may interfere. Normally, the operating system does not allow any other process including malicious ones to access the memory space of the first process. However, the critical process and the malicious process actually share the same physical memory. There are many ways to by pass the restrictions set by the OS, such as stack overflow. By doing so, the malicious process may be able to change the behavior of the first process. In addition, if the attacker is powerful enough, he may be able to directly change the contents of the memory via physical attack. In such a case, existing OS level protection techniques will not suffice. Thus, we need some extra protection.

Our goal is to provide real time integrity protection for a part of memory which

will be referred to as the unprotected memory. In other words, we want to detect any unauthorized modification to the unprotected memory, which is the ordinary memory in the example above. We assume that the unprotected memory is considered cheap such that we can save a relatively large amount of data without incurring a high cost. Data in the unprotected memory, regardless of whether it is data we are protecting or the tags we generate in our scheme, are all susceptible to unauthorized accesses and modifications.

The attacker in our model can view and modify any data stored in the unprotected memory as needed. He can also call all the functions provided by the scheme, such as update and check, with any parameters. For instance, he can legally update the data in any position to any value to collect different tags in our scheme. The goal of the attacker is to perform unauthorized modification to the data and let the system read the wrong data at least one time without being detected by the subsequent check.

The attacker may also record a large collection of data to analysis the system. However, he can only collect data without triggering an alarm. If any attack is detected, the system will halt and all the keys will be changed after the system is reset. In addition, the amount of data that the attacker can collect is still limited in a small scale comparing to the security level of our scheme. This can be achieved by changing the keys after every certain number of operations.

We need some place to store critical data such as keys and temporary data generated and used by the proposed scheme. Therefore, we assume that a relatively small amount of protected memory is reserved for these purposes. This memory can be protected by hardware or by the secure extensions of the OS. We assume the protected memory is neither readable nor writable by the adversaries. Since this part of the memory will be expensive, we will try to minimize the size of the

protected memory used by our scheme. Finally, similar to the protected memory, we assume that all the operations, such as hash and encryption are well protected and thus are safe from attacks.

For a software implementation we assume that the OS can perform the needed operations safely and the protected memory can be provided by some secure extensions such as TPMs. This is a reasonable assumption since attacking the encryption operation is much harder than simply forging data in the memory.

3.2 Overview of The Scheme

Merkle trees provide an excellent authentication technique without the need for too much protected storage. However, when used together with cryptographic hash functions, the combination becomes less practical. Cryptographic hash functions such as SHA-1 and MD5 provide excellent performance for many applications. However, as far as memory integrity is concerned, the frequency of memory accesses in the execution of a process will be significantly high that using cryptographic hash functions becomes impractical. Typically, several hundred clock cycles will be required to hash a single block. Coupled with the time needed to traverse the Merkle tree, such a scheme will simply bear too high of an overhead to be considered practical.

As described earlier a universal hash function family is a collection of functions that map data to digests. In our scheme, we use the universal hash family NH which was proposed for use in the UMAC scheme [9]. NH exhibits excellent performance in software as it was designed to be naturally compatible with the instructions supported on common processor architectures. That said, despite the precise characterization of the statistical properties of universal hash functions, when employed alone they do not provide any cryptographic protection. However, when the output of the

universal hash functions is encrypted using a random bit string, the combination can provide provable security.

In the scheme we propose, we take advantage of an important property of NH in order to improve its online performance. That is, if the universal hash function is proved to be additive-universal, then adding a random mask to the hash result will produce a provably secure MAC. This property will allow us to compute the encryption masks off line. Hence, even if a 128-bit cryptographic block cipher such as AES is used the addition of the masks will only take one cycle assuming the mask has been pre-computed and stored in a cache. Such topics will be discussed in following chapters.

The universal hash function we used in our scheme and in our simulation is computed by fixing NH to a word-size of 32-bits as follows

$$\begin{aligned} \mathsf{NH}_K(M) &= (M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2) +_{64} \cdots \\ &\quad +_{64}(M_{l-1} +_{32} K_{l-1}) \times_{64} (M_l +_{32} K_l) \end{aligned}$$

where M_i denotes the i-th message word and K_i stands for the i-th key word. The 32-bit addition $+_{32}$ and the 64-bit multiplication $\times_{64}$ are implemented using the 32-bit addition and multiplication instructions provided by the CPU. Thus the hash function can be realized efficiently in software.

The output size of this particular instance of NH is 64-bits which gives a collision probability of 2^{-32}. This does not provide a sufficient security margin. To overcome this problem we employ the multi-hashing technique introduced in [23]. The technique was proposed to increase the security level of a given hash function without changing the size of the hash value at the expense of more key material. We reverse this procedure to preserve the security level while reducing the size of the

hash value. Furthermore, we use the Toeplitz approach [18] to reduce the amount of key material needed. We achieve this by employing the hash function two times with different keys and get two 64-bit hash results. Note that due to the Toeplitz approach the second key can be taken as the one word shifted version of the first key. By computing two separate hash results and concatenating them we obtain a hash result of 128 bits which does provide adequate collision probability.

Another important advantage of this family of universal hash functions is its additive structure. This feature means that the hash value can be computed incrementally. One does not need to go through an entire block of data (i.e. l-words) to calculate the new tag. Hence, computing new tags for modified data can be carried out extremely fast. It can be shown as follows

$$\begin{aligned} NH_k(M) &= \sum_{i=1;i\neq m}^{l/2} (M_{2i-1} +_{32} K_{2i-1}) \times_{64} (M_{2i} +_{32} K_{2i}) \\ &\quad +_{64}(M_{2m-1} +_{32} K_{2m-1}) \times_{64} (M_{2m} +_{32} K_{2m}) \end{aligned}$$

$$\begin{aligned} NH_k(M') &= \sum_{i=1;i\neq m}^{l/2} (M_{2i-1} +_{32} K_{2i-1}) \times_{64} (M_{2i} +_{32} K_{2i}) \\ &\quad +_{64}(M'_{2m-1} +_{32} K_{2m-1}) \times_{64} (M'_{2m} +_{32} K_{2m}) \end{aligned}$$

$$\begin{aligned} NH_k(M') &= NH_k(M) -_{64} (M_{2m-1} +_{32} K_{2m-1}) \times_{64} (M_{2m} +_{32} K_{2m}) \\ &\quad +_{64}(M'_{2m-1} +_{32} K_{2m-1}) \times_{64} (M'_{2m} +_{32} K_{2m}) \end{aligned}$$

As opposed to previous proposals which do provide coarse grain incremental hashing through the use of Merkle trees, our scheme allows for even finer grain

verification at the hash level. This was not possible in earlier schemes which employed traditional hash functions such as SHA-1. Effectively, the usage of NH with the Merkle tree structure yields an efficient integrity protection scheme. However, as we mentioned earlier the universal hash function is not secure. If we leave the data and the corresponding tags as plaintext stored in memory like, the attacker can compromise the hash function and therefore find collisions. To solve this problem we encrypt the tags stored in unsecured memory. The encryption process has to be fast so that it does not become the bottleneck of the scheme. The simplest yet most effective candidate for such an encryption process is a stream cipher. To mimic a stream cipher type of encryption we generate different masks for each tag using AES. These masks can be pre-computed or computed in parallel in order to hide the latency of generating the masks behind the memory access latency. For a software implementation the encryption process is achieved by adding masks to the hash results produced by NH.

3.3 The Basic Protocol

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 will be introduced. It is easy to extend this scheme to an l-level version. The generated tags are then encrypted by a simple addition of the mask. The masks are generated by encrypting a random seed using a block cipher. For this task we use AES. The seeds are randomly generated and saved in the unprotected memory in plaintext alongside the produced tags. The basic scheme is presented in Figure 3.2. This process is also outlined in Figure 3.1.

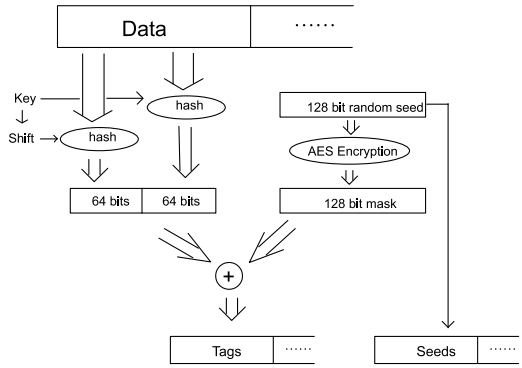

Figure 3.1: Outline of the basic scheme

3.4 Implementation Concerns

The NH we introduced above works well for software implementation. Only multiplication and addition are needed for the NH. Moreover, some variances of NH can take advantage of the SIMD instructions to make the hash operation even faster([9]). These variances of NH still perfectly keep the mathematical properties of the NH. Thus they can be used to implement the scheme when needed without any problem.

For hardware implementation, some variances of the NH have been introduced in [28]. These versions of universal hash functions have better performance with hardware implementation in speed, power consumption, and area while still keeping the same security level. Also, since they are basically NH of polynomials over $GF(2)$ instead of integers, all the useful properties of the NH such as incremental update are still kept.

Actually, the proposed scheme is more suitable for hardware since parallel computation, which is hard to achieve via software, can improve the performance greatly. Details will be discussed in the following chapter.

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.
Step	**Update(M_i):**
1	Find the random seed S_i corresponding to the data block M_i and compute the mask A_i.
2	Subtract A_i from T_i to obtain the stored H_i.
3	Update the H_i incrementally to H_i'.
4	Generate a new pair of 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'.
5	Update the root of the Merkle tree R_i incrementally to obtain the new root R_i'.
Step	**Check(M_i):**
1	Find the random seed S_i corresponding to the data block M_i and compute the mask A_i.
2	Subtract A_i from T_i to obtain the stored H_i.
3	Compute the hash value H_i' of the block M_i'. If $H_i \neq H_i'$, fail the integrity check.
4	Compute the hash value R_i' of the tags. Compare it with the R_i stored.
5	If $R_i \neq R_i'$, fail the integrity check.

Figure 3.2: The Basic Integrity Checking Scheme

Chapter 4

Possible Improvements

In this chapter, we will introduce some possible improvements to the basic scheme proposed in last chapter. These improvements will need some extra secure storage space and computation units. However, they can largely improve the performance of the scheme and make the scheme closer to real-time authentications.

4.1 Using Caches

Making use of caches is a widely used method to improve the efficiency of storage systems. Since our scheme is designed to protect a storage system, caches can also be employed to improve the performance.

In the CPU cache system, frequently referenced data is cached to reduced the frequency of the relatively costly memory access. The bottleneck of the proposed scheme is 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 hashed to meet the needs of different applications.

If the mask is hashed, the latency introduced by recovering the current mask

when the system is going to read the hash result can be saved. However, the time needed for preparing the new mask cannot be saved because a new mask is needed for each update to maintain the one-time-pad security level. This latency can be hidden through pre-computation or parallel computation as will be discussed in the next part. However, if the frequency of operations becomes higher, the bottleneck still exists.

Alternatively, the hash result itself can be cached. 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. However, with a hash cache, the tags are only updated when the hash results have to leave the cache. Therefore, the tags stored in the memory may not represent the latest data. In other words, the tags may not match the current data. Although it will not affect the security level when the whole system works well, it may cause some troubles in some special cases. For instance, if the caches become unavailable suddenly, the scheme with hash caches cannot continue to work while the one with mask caches can still work.

No matter it is the mask or the hash result that is going to be cached, the caches need to be safe, otherwise the attacker may get the hash result in plain text and break the scheme based on it. Thus, basically the idea of caches is exchanging secure storage space for speed and is useful in the case when speed is more important.

Moreover, putting the hash results in a secure cache has some other useful properties. Even the hash results are cached, the system still needs to recover the old mask when the hash result is loaded into the cache for the first time. However, this operation can be delayed. The incremental update of the hash result and the encryption by the one=time-pad like mask are both achieved by addition. Thus, one can actually update the tags without even unmasking the tags and change the masks later. Normally, it is unsafe since the update of hash results should happen

in the same time as changing the mask to a new one or the one-time-pad security level is lost. However, with a hash cache, this operation becomes safe because the attacker can neither view or change the possible unsafe tags in the caches. Although finally the computation needed for unmasking can never be saved, this property can help the scheme to achieve a good peak performance by delaying the costly AES operations.

4.2 Pre-computation and Parallelism

The 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. Note that in the proposed 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 the scheme.

Furthermore, in a Merkle tree, the higher level has to wait until the lower level finishing the update and then starts updating. In other words, higher levels of the tree depend on 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 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 basic 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 it is the seeds instead of the tags to be treated as data for higher levels, 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 for the Merkle tree can be employed and it is still safe as will be shown in next chapter.

Parallel computation and pre-computation will cost some extra computation units. However, they can improve the performance largely.

4.3 Error Inheritance

Another benefit of the proposed scheme is what we refer to as *error inheritance*. Due to the incremental property of the universal hash 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. The SHA-1 based scheme on the other hand does not offer such a property. The tags will have to be checked every update cycle in order to detect the errors. This advantage of our proposed schemes seems to allow a lower rate of integrity checks, which translates to a faster performance of the application.

The security level of the error inheritance property relies on that the attacker cannot hide the trace of the past attack. In the NH, such trace is the subtraction of the sub result which is calculated from old attacked data. However, if the attacker can control the legal update, which is highly possible, it can cancel the trace by letting the system update to the attacked data:

The hash result for a block of data where the attacker changed the data M_1 to M_1' may looks like this:

$$
\begin{aligned}
NH &= NH_{others} +_{64} (M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2) \\
&\quad -_{64}(M_1' +_{32} K_1) \times_{64} (M_2' +_{32} K_2) \\
&\quad +_{64}(M_1'' +_{32} K_1) \times_{64} (M_2'' +_{32} K_2)
\end{aligned}
$$

If the new legal value for the attacked position is M_1', M_2', the process will be:

$$
\begin{aligned}
NH_{new} &= NH_{others} +_{64} (M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2) \\
&\quad -_{64}(M_1' +_{32} K_1) \times_{64} (M_2' +_{32} K_2) \\
&\quad +_{64}(M_1'' +_{32} K_1) \times_{64} (M_2'' +_{32} K_2) \\
&\quad -_{64}(M_1'' +_{32} K_1) \times_{64} (M_2'' +_{32} K_2) \\
&\quad +_{64}(M_1' +_{32} K_1) \times_{64} (M_2' +_{32} K_2) \\
&= NH_{others} +_{64} (M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2)
\end{aligned}
$$

In this way, the attacker not only clears the trace of past attack, but also causes the hash result to be valid for M_1, M_2 when the legal 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, he can simply modify the data and wait the value that can help him to appear.

Such problems compromise the property completely, however, if the data is selected completely at random, which means that the attacker can neither control nor predict the data, this 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 with carefully designed schemes. In the next part, a scheme that makes the data random and hence keeps the error inheritance property will be discussed. With that design, even the check before every read operation can be saved.

4.4 Putting It All Together

In this section, a protocol taking advantage of all mentioned improvements will be discussed. In practice, one may only use part of the possible improvements to find a trade off in efficiency and cost.

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 mentioned above, the data needs to be random to keep the error inheritance property. Thus for each single byte of the data, a random number called randomizer is assigned when the tag of that block of data is loaded into the cache.

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 original 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 now can be treated as updating the data to the same value. Because 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, a check of the randomizers and the masked data is needed. Then the block of data will be hashed as usual way and the hash result will be masked then saved to the memory just as in the basic scheme. The check is necessary because when the hash is calculated in as usual, the data is no longer protected with randomizers so that the data will become predictable and the error inheritance property is lost. Also, the randomizers have to change 128 bits one time to achieve a 128-bit security level.

This design needs a large amount of extra secure and unsecured memory space and will introduce some overhead. More computation units are also needed. However, we eliminate the check process, which is much slower than the update operation, before every access of the memory. 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 do not need such randomizers to maintain the error inheritance property because the seeds and the tags are random values.

Step	**Initialization:**
1	Same as in the Basic Scheme.
2	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.
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.
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.

Figure 4.1: Combination Scheme Part I

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	check whether the block is marked as checked.
6	If it is true, return true.
7	If the block is not checked before, start checking higher levels as the basic scheme.

Figure 4.2: Combination Scheme Part II

Chapter 5

Security Analysis

The security analysis of the schemes proposed in previous chapters will be present in this chapter. The schemes can be proved to be secure enough for practical usage. This part is written together with Ghaith Hammouri.

5.1 Analysis for the Basic Scheme

The main purpose of this scheme is to protect data from unauthorized modifications. Thus the security level is described by how hard it is for an attacker can forge the data without being detected. As mentioned above, we assume that the root of the hash tree, the keys used and the operations themselves are assumed to be secure. Any modification to the data will have to provide a valid tag corresponding to the new data block. One way to find such a tag is to replay old data along with their tags. This is where the Merkle tree comes into play. Although one can replay some of the leaves in the tree, it is not possible to replay the root of the tree which is kept in a secure location. One can also hope to find a new tag which does not affect the second level hash value which is kept in secure memory. The probability of such an attack succeeding is equal to the collision probability of the hash function used.

In our case this probability is 2^{-2w}. Thus the only way to compromise our scheme is to find a data-block M' which happens to have the same tag as that of a valid data-block M. As we will show in this section the probability of such an attack succeeding is bounded by the probability of finding the pre-image of a randomly chosen AES cipher text (without knowledge of the key).

The universal hash function we used in our scheme is proved to be 2^{-32}-almost universal [9] which corresponds to a collision probability of 2^{-32}. Since we used the Toeplitz method in our scheme, the probability of a collision is reduced to 2^{-64}. The proof for this probability can be found in [9]. For our purposes we will be interested in a slightly different probability, namely **ε-almost-Δ-universal (ε-AΔU)** [16], which is defined as follows.

Definition 4 *The set of hash functions $H = h : A \rightarrow B$ is said to be* ***ε-almost-Δ-universal*** *if for every $M, M' \in A$ where $M \neq M'$ and for all $a \in B$,*

$$|h \in H : h(M) - h(M') = a| \ = \delta_H(M, M') \leq \varepsilon |H| \ .$$

For NH the subtraction in the above definition is done modulo 2^{2w}. Another simple way to view the above definition is by saying that h is ε-AΔU if $\Pr[h(M) - h(M') = a] \leq \varepsilon$. For a 2-level Toeplitz construction like the one we use this probability is 2^{-2w}.

Note that the simple collision definition will not be sufficient for our purposes. This is the case since we utilize a pseudo-random mask and add it to the hash result in order to provide strong cryptographic protection to the hash function. Recall that the mask is computed by encrypting a random seed using a block cipher (AES). This motivates the following definition which describes the probability of an adversary attacking AES.

Definition 5 *Let A be an adversary who is given two random bit-strings $C, R_1 \in \{0,1\}^k$. Then we define the advantage of the adversary A by*

$$\mathbf{Adv}_A^{AES}(k) = \Pr[A(C, R_1) = R_2 | C = \mathrm{AES}_K(R_2) - \mathrm{AES}_K(R_1)].$$

Where $\mathrm{AES}_K(r)$ denotes the AES encryption of some string $r \in \{0,1\}^k$ using a randomly chosen key $K \in \{0,1\}^k$ which is hidden from the adversary A. The subtraction is done modulo $2^{k/2}$ separately on each of the two $k/2$ bits of the cipher text, and the probability is taken over all cipher texts C, all keys K and random strings R_1.

With this definition we can now bound the probability of an adversary succeeding in attacking our scheme.

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 $\tau_i = \mathsf{NH}_{K_1}(M_i^1)|\mathsf{NH}_{K_2}(M_i^2) + \mathrm{AES}_{Ke}(R_i)$ where the addition is done modulo 2^{2w} separately on each of the two parts of the $\mathrm{AES}_{Ke}(R_i)$, $|$ 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. $Ke, R_i, \tau_i \in \{0,1\}^k$ where we have set $k = 4w$. For such an adversary we define P_A to be the probability of A finding M^1, M^2 and R where $M^1 \neq M_j^1$ and $M^2 \neq M_j^2$ such that $\tau = \mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) + \mathrm{AES}_{Ke}(R)$ and $\tau = \tau_j$ for some $j \in [1, \ldots, q]$. Given the above, P_A is bounded by*

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

Proof The adversary A needs to find new messages which map to the same tag 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 $(M_t^1, M_t^2, R_t, \tau_t)$ where $t \in [1, \ldots, q]$ and $M_t^1 \neq M_j^1$ and $M_t^2 \neq M_j^2$. A can now either fix R_t, or fix the messages M_t^1, M_t^2. Let us start by fixing R_t, in such a case A needs to find an M^1, M^2 such that the tuple (M^1, M^2, R_t, τ_j) is valid. For this case A needs to have

$$\begin{aligned}
\mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) + \mathrm{AES}_{Ke}(R_t) &= \tau_j \\
\mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) + \tau_t - \mathsf{NH}_{K_1}(M_t^1)|\mathsf{NH}_{K_2}(M_t^2) &= \tau_j \\
\mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) - \mathsf{NH}_{K_1}(M_t^1)|\mathsf{NH}_{K_2}(M_t^2) &= a \quad ,
\end{aligned}$$

where $a = \tau_j - \tau_t$. The probability of finding such M^1 and M^2 is the ε-AΔU of the proposed scheme which happens to be 2^{-2w}.

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

$$\begin{aligned}
\mathsf{NH}_{K_1}(M_t^1)|\mathsf{NH}_{K_2}(M_t^2) + \mathrm{AES}_{Ke}(R) &= \tau_j \\
\tau_t - \mathrm{AES}_{Ke}(R_t) + \mathrm{AES}_{Ke}(R) &= \tau_j \\
\mathrm{AES}_{Ke}(R) - \mathrm{AES}_{Ke}(R_t) &= b \quad ,
\end{aligned}$$

where $b = \tau_j - \tau_t$. The probability of finding such R is equal to $\mathbf{Adv}_A^{AES}(4w)$ 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 have

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

The above theorem essentially reduces the security of the proposed scheme to that of the AES algorithm. As the cryptographic features of AES are will studied it is quite safe to have the security of our scheme reduce to that of AES.

□

The random number generator we used in our scheme will not have a large impact on the security level. Note that the seeds generated will be stored in unprotected memory. If the seeds do not repeat, the block cipher can make sure the masks are almost uniform and therefore serve as good "pseudo-one-time-pads". Even a timer can serve as the random number generator here, as long as the attacker cannot force it to produce the same seed twice.

As mentioned in the instruction part, in [12] the authors proposed some possible attacks to NH. However, such attacks just provide a way to recover the keys with a probability of 2^w, which does not reduce the strength of the NH. In addition, the proposed attack needs observing whether the hash results is changing or not, which is impossible in our scheme since the tags are protected by one-time-pad. Therefore, the proposed attack methods in this paper do not affect the secure level of our schemes.

Birthday Attack Normally, with birthday attack, the attacker can cut the security level by half, which means from 64-bits to 32-bits, etc. However, the birthday attack will not reduce the security level in our scheme.

To perform a birthday attack to our scheme. The attacker may either try to find any two of the messages that maps to the same hash result (unmasked) or try to find two messages that map to the same tag (masked). Because the hash result is protected by one-time-pad, the attacker cannot know even he get the same result. Thus the attacker may only choose the second way to initial the attack. However, the tag size is actually double the size of the collision probability. Therefore, even

the security level is reduced by the birthday attack, it still keeps the same level as the case that attacking the scheme normally as mentioned above.

5.2 Analysis for Improvements

The improvements for the scheme bring small changes to the security model, thus further analysis is needed.

For the caches. If the masks are cached, nothing related to security is actually changed. As long as the caches are safe, the security level remains the same. If the hash values of the data are cached instead, it is obvious that the only way to compromise the scheme is to find collisions because the attacker cannot even read the cached hash results. That will keep the scheme in a 2^{-2w} security level.

Pre-computation In the proposed scheme and the improved version introduced later, the seed-mask pairs and the randomizers can be pre-computed. Both of them should be kept as secret before they are referenced. With this setting, the security level will be the same as these values are generated online. Thus pre-computation will not affect the security of the scheme.

Error Inheritance

The incremental updating gives the NH an useful property we called "error inheritance" that a wrong pair of data and tag will be upgraded to another wrong data-tag pair. If the data is changing randomly, it is safe to take advantage of such property to avoid to check the integrity of the data before each access. The random masks has nothing to do with this attack method since they will cancel each other if the attacker do not touch them at all. If the attacker attacks the masks, the security level will rely on the AES strength as shown before. Alternatively, the attacker may attack the incremental hash process.

Remember that the incremental update process can be written as

$$\begin{aligned} NH &= NH_{others} +_{64} (M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2) \\ &\quad -_{64}(M_1' +_{32} K_1) \times_{64} (M_2' +_{32} K_2) \\ &\quad +_{64}(M_1'' +_{32} K_1) \times_{64} (M_2'' +_{32} K_2) \end{aligned}$$

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

$$\begin{aligned} NH_{others} &+ {}_{64}(M_1''' +_{32} K_1) \times_{64} (M_2''' +_{32} K_2) \\ = NH_{others} &+ {}_{64}(M_1 +_{32} K_1) \times_{64} (M_2 +_{32} K_2) \\ &- {}_{64}(M_1' +_{32} K_1) \times_{64} (M_2' +_{32} K_2) \\ &+ {}_{64}(M_1'' +_{32} K_1) \times_{64} (M_2'' +_{32} K_2) \end{aligned}$$

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 adversary who is to attack the incremental update. With a successful attack A will have that $\mathsf{NH}_{K_1}(M_1^1)|\mathsf{NH}_{K_2}(M_1^2) - \mathsf{NH}_{K_1}(M_2^1)|\mathsf{NH}_{K_21}(M_2^2) = \mathsf{NH}_{K_1}(R^1)|\mathsf{NH}_{K_2}(R^2) - \mathsf{NH}_{K_1}(M_3^1)|\mathsf{NH}_{K_2}(M_3^2)$ where the addition is done modulo 2^{2w} separately on each of the two parts, $|$ is the string concatenation operator, and $K_1, K_2, M_i^1, M_i^2, R^1, R^2 \in \{0,1\}^{2w}$ such that K_2 is the shifted version of K_1. R^1, R^2 are random numbers being generated after $M_2^1, M_2^2, M_3^1, M_3^2$ are chosen. For such an adversary we define P_A to be the probability of A finding $M_1^1, M_1^2, M_2^1, M_2^2, M_3^1, M_3^2$ where $M_2^1 \neq M_3^1$ and $M_2^2 \neq M_3^2$ such that the above equation is satisfied. Given the above, P_A is bounded by*

$$P_A \leq 2^{-2w}$$

Proof To make the equation works, 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_1^1, M_1^2 such that

$$\mathsf{NH}_{K_1}(M_1^1)|\mathsf{NH}_{K_2}(M_1^2) - \mathsf{NH}_{K_1}(M_2^1)|\mathsf{NH}_{K_2}(M_2^2) = a$$

where a is some constant. If $M_1^1 \neq M_2^1$ and $M_1^2 \neq M_2^2$, since the NH is 2^{-2w}-almost-Δ-universal as mentioned above, the probability to find the proper M_3^1, M_3^2 is 2^{-2w}.

The attacker may alternatively choose $M_1^1 = M_2^1$ and $M_1^2 = M_2^2$ 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_3^1, M_3^2 such that

$$\mathsf{NH}_{K_1}(R^1)|\mathsf{NH}_{K_2}(R^2) - \mathsf{NH}_{K_1}(M_3^1)|\mathsf{NH}_{K_2}(M_3^2) = a$$

where a is some constant.

Similarly, the attacker may either let a be zero and try to make $M_3^1 = R^1$ and $M_3^2 = R^2$ or try to find some $M_3^1 \neq R^1$ and $M_3^2 \neq R^2$ to meet the equation where a can be any value.

If the attacker tries to make $M_3^1 = R^1$ and $M_3^2 = R^2$, since R^1, R^2 are random numbers generated after M_3^1, M_3^2 are chosen, the probability for that $R^1 = M_3^1, R^2 = M_3^2$ is 2^{-2w}. If the attacker chooses the other way, the problem is again reduced to the 2^{-2w}-almost-Δ-universal property of NH.

Therefore, the probability to compromise the error inheritance is at most

$$P_A \leq 2^{-2w}$$

□

Tree of Seeds

If the Merkle tree is not formed by tags which serve as the hash results in original Merkle design, it can be formed by seeds instead and remain the same security level. It can be proof similarly.

Theorem 3 *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 $\tau_i = \mathsf{NH}_{K_1}(M_i^1)|\mathsf{NH}_{K_2}(M_i^2)+\mathrm{AES}_{Ke}(R_i)$ where the addition is done modulo 2^{2w} separately on each of the two parts of the $\mathrm{AES}_{Ke}(R_i)$, $|$ 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. $Ke, R_i, \tau_i \in \{0,1\}^k$ where we have set $k = 4w$. For such an adversary we define P_A to be the probability of A finding M^1, M^2 and τ where $M^1 \neq M_j^1$ and $M^2 \neq M_j^2$ such that $\tau = \mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) +$*

$\mathrm{AES}_{Ke}(R)$ *and* $R = R_j$ *for some* $j \in [1, \ldots, q]$. *Given the above,* P_A *is bounded by*

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

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 $(M_t^1, M_t^2, R_t, \tau_t)$ where $t \in [1, \ldots, q]$ and $M_t^1 \neq M_j^1$ and $M_t^2 \neq M_j^2$. A can now either fix τ_t, or fix the messages M_t^1, M_t^2. Let us start by fixing τ_t, in such a case A needs to find an M^1, M^2 such that the tuple (M^1, M^2, R_j, τ_t) is valid. For this case A needs to have

$$\begin{aligned}
\mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) + \mathrm{AES}_{Ke}(R_j) &= \tau_t \\
\mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) + \tau_t - \mathsf{NH}_{K_1}(M_t^1)|\mathsf{NH}_{K_2}(M_t^2) &= \tau_t \\
\mathsf{NH}_{K_1}(M^1)|\mathsf{NH}_{K_2}(M^2) - \mathsf{NH}_{K_1}(M_t^1)|\mathsf{NH}_{K_2}(M_t^2) &= a \quad ,
\end{aligned}$$

where $a = \tau_j - \tau_t$. The probability of finding such M^1 and M^2 is the ε-AΔU of the proposed scheme which happens to be 2^{-2w}.

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

$$\begin{aligned}
\mathsf{NH}_{K_1}(M_t^1)|\mathsf{NH}_{K_2}(M_t^2) + \mathrm{AES}_{Ke}(R_j) &= \tau \\
\tau_t - \mathrm{AES}_{Ke}(R_t) + \mathrm{AES}_{Ke}(R_j) &= \tau \\
\tau - \tau_t = b \quad ,
\end{aligned}$$

where $b = \mathrm{AES}_{Ke}(R_j) - \mathrm{AES}_{Ke}(R_t)$. The probability of finding such τ is equal to

$\mathbf{Adv}_A^{AES}(4w)$ 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

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

The above theorem essentially reduces the security of the proposed scheme to that of the AES algorithm. As the cryptographic features of AES are will studied it is quite safe to have the security of our scheme reduce to that of AES.

□

In the basic scheme, the seeds can be generated from some unsecured source, such as counters. However, for a tree of seeds, if one want to take advantage the error inheritance property, the seeds have to be unpredictable. Thus some secure random source has to be used to get the seeds.

Chapter 6

Performance

This chapter focus on the performance of the proposed scheme. The theoretical analysis and the test result will show that the new scheme is quite promising.

6.1 Analysis

In this section the memory and time requirements for the proposed schemes are estimated. The most extreme case is that the checking process will hash an entire block. It is obvious that a smaller block size would speed up the process. However, a smaller block size will mean the usage of more tags and therefore more levels, which translates into a larger space requirement. Now assume that the same block size for both the data and the tags is used, an estimate for the speed and the space requirements can be worked out. Firstly the pre-computation phase is not taken into account to show the efficiency of the worst case. Then some estimation for the case that pre-computation and parallelism are perfectly working will be given to show the peak performance. Let us start by defining the variables used in the analysis as shown in Table 6.1. The scheme with such values will have a 2^{-128} secure level as the one will be used in performance test. Note in the table that due to the

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

Table 6.1: Variables used in analysis

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 one time, such as the updating of the higher levels of the tree, 100 cycles are enough per two words.

The ratio $\frac{B}{T}$ will represent the fan out in each node of the Merkle tree. With l-levels the number of nodes in the last level (the actual data) will be $\left(\frac{B}{T}\right)^{l-1}$. Therefore, to protect a memory of size M, the following relation has to satisfied

$$M = R\left(\frac{B}{T}\right)^{l-1} B \quad . \tag{6.1}$$

The time needed per update t_{up} for the scheme with mask caches will essentially depend on the number of levels in addition to the cache hit rate. When a data block is updated, the hash value will change for each of the l-nodes starting from the modified block up to the root of the Merkle tree. Moreover, with the exception of the root each one of these nodes will have to be un-masked before the hash value

can be updated. This is where the hit rate comes into play. When the mask has not been cached the delay incurred will be the time for retrieval which is considered negligible compared to the time needed to calculate the mask. With this, the time delay for an update will be

$$t_{up} = t_u + (l-1)[\frac{t_u T}{8} + (1-H)t_{AES}T + t_{AES}T + t_r S] \quad . \tag{6.2}$$

Similar to the update process, the check process will need a hash operation for each level and an encryption for each level of tags in the unprotected memory if a miss occurs. However, if there is a hit in the cache, there is no need to check the higher levels since it is hard for the attacker the data tag pair without access to the cached mask. The time needed per check t_{check} will therefore be

$$t_{check} = t_h B + (1-H)\frac{1-(1-H)^{l-1}}{H}(t_h B + t_{AES}T) \quad . \tag{6.3}$$

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

$$t'_{up} = t_u + (1-H)\frac{1-(1-H)^{l-1}}{H}(\frac{t_u T}{8} + 2t_{AES}T + t_r S) \tag{6.4}$$

.

$$t'_{check} = t_h B + (1-H)\frac{1-(1-H)^{l-1}}{H}(t_h B + t_{AES}T) \quad . \tag{6.5}$$

Similarly, no checks for higher levels are needed if there is a hit in the cache.

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

$$t_{check\&up} = t_u + t_h B + (l-1)[\frac{t_u T}{8} + (1-H)(t_{AES}T + t_h B) + t_{AES}T + t_r S] \tag{6.6}$$

$$t'_{check\&up} = t_u + t_h B + (1-H)\frac{1-(1-H)^{l-1}}{H}(\frac{t_u T}{8} + t_h B + 2t_{AES}T + t_r S) \tag{6.7}$$

.

The size of the unprotected memory will mainly depend on the number of nodes in the Merkle tree with the exception of the root which is stored in protected memory. The number of unprotected nodes will be

$$\sum_{i=1}^{l} \left(\frac{B}{T}\right)^i = \frac{B}{T}\left(\frac{1-\left(\frac{B}{T}\right)^{l-1}}{1-\left(\frac{B}{T}\right)}\right) \quad . \tag{6.8}$$

The size of the unprotected memory will be

$$M_u = M'_u = \frac{B}{T}\left(\frac{1-\left(\frac{B}{T}\right)^{l-1}}{1-\left(\frac{B}{T}\right)}\right)(T+S)R \quad . \tag{6.9}$$

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

$$M_p = M'_p = \left[\frac{B}{T}\left(\frac{1-\left(\frac{B}{T}\right)^{l-1}}{1-\left(\frac{B}{T}\right)}\right)TC + T\right]R + B \quad . \tag{6.10}$$

These values for the basic scheme can also be calculated just as the scheme with mask caches with H and C being zero.

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

$$\begin{aligned} t''_{up} &= \frac{t_u T}{8} + \frac{t'_r T}{2} + (1-H)(3t_h B + 2t_{AES} T + t'_r S + \frac{t_u T}{4}) \\ &\quad +2(1-H)^2 \frac{1-(1-H)^{l-2}}{H}[\frac{t_u T}{8} + t_h B + (2t_{AES} T + t'_r S)] \quad . \end{aligned} \tag{6.11}$$

$$t''_{check} = (1+H)t_h B + (1-H)\frac{1-(1-H)^{l-1}}{H}(t_h B + t_{AES} T) \quad . \tag{6.12}$$

$$M''_u = \frac{B}{T}\left(\frac{1-\left(\frac{B}{T}\right)^{l-1}}{1-\left(\frac{B}{T}\right)}\right)(T+S)R + MC \quad . \tag{6.13}$$

$$M''_p = \left[2\frac{B}{T}\left(\frac{1-\left(\frac{B}{T}\right)^{l-1}}{1-\left(\frac{B}{T}\right)}\right)TC + T\right]R + B \quad . \tag{6.14}$$

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

$$t_{check\&up} = max[t_h B, t_u + (l-1)\frac{t_u T}{8}] \tag{6.15}$$

with mask caches and

$$t'_{check\&up} = max[t_h B, t_u] \quad . \tag{6.16}$$

with hash result caches. The update time needed for the scheme with randomizers will become

$$t''_{up} \quad = \quad \frac{t_u T}{8}. \tag{6.17}$$

The average case will be hard to express if the parallel computation is introduced. However, it is easy to get the estimation that the average time needed for the schemes without randomizer will never be less than $t_h B$ while the scheme with randomizers will consume more time than $\frac{t_u T}{8}$ only when the cache misses.

We can see clearly from equation 6.1 the relation between the block size and the number of levels as well as the size of memory to protect.

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

We may also find out from other equations that the block size and the number of levels play important roles in the speed performance and space demands. Generally speaking, fewer levels and smaller block size will result in better performance

and need less storage space. However, the block size and the number of levels are restricted by equation 6.1. A smaller block will result in more levels and vise versa. We may have to find a balance between them for different practical applications. The numeric results in table 6.2, 6.3, 6.4 and 6.5 may provide a clearer view of this.

We also notice from equation 6.1 that the block size and number of levels will only grow with the memory size in a very low speed, which means that the performance in speed and space will not drop a lot for a larger memory. This is mainly due to the tree structure we used for the tags.

l	t_{check}	t'_{check}	t''_{check}
2	889938	889938	1483102
3	40862	40863	67077
4	8835	8835	14346
5	3616	3616	5778

Table 6.2: Check speed (cycles)

l	$t_{check\&Up}$	$t'_{check\&up}$	t''_{up}
2	743155	890320	446489
3	36068	41301	23584
4	11789	9284	6530
5	9202	4068	3712

Table 6.3: Update speed (cycles)

l	$t_{check\&up}$	$t'_{check\&up}$	t''_{up}
2	741455	741455	400
3	32768	32768	400
4	6889	6889	400
5	2702	2702	400

Table 6.4: Peak Update speed (cycles)

As shown in table 6.5, the unprotected memory needed for the scheme with randomizers is huge due to the extra demands in storage space for randomizers.

l	M_p	M'_p	M''_p	M_u	M'_u	M''_u
2	0.42	0.42	0.50	0.71	0.71	819.91
3	1.62	1.62	3.22	16.03	16.03	835.23
4	7.69	7.69	15.37	76.82	76.82	896.02
5	19.87	19.87	39.74	198.72	198.72	1017.92

Table 6.5: Space Demand (MB)

However, the caches 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.

The estimation for the basic scheme and SHA-1 based Merkle Tree are also present in table 6.6 and 6.7 as a comparison. The SHA-1 speed is set to 12 cycles per byte, which is collected from some optimized codes. 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, the speed is very low compared to our proposed schemes.

l	$t_{check\&up}$	t_{check}	M_u	M_p
2	1485378	1483870	0.71MB	353.58KB
3	103140	100224	16.03MB	15.66KB
4	34758	30434	76.82MB	3.32KB
5	23084	17352	198.72MB	1.32KB

Table 6.6: Estimation Basic scheme

As mentioned in the previous work part, GHASH performs well for memory au-

l	$t_{check\&up}$	t_{check}	M_u	M_p
2	7034100	7034100	0.3MB	20B
3	431200	431200	7.2MB	20B
4	116200	116200	35.8MB	20B
5	55700	55700	94.6MB	20B

Table 6.7: Estimation for SHA-1 based scheme

thentication. According to [27], GHASH can achieve 13.1 Cycles Per Byte (CPB) using a Motorola G4 CPU with polished design (pre-computation, optimized algorithm etc.). However, the universal hash we use can run at 1.58 CPB using a Motorola PowerPC 604e and can even run at 0.66 CPB using AltiVec SIMD instructions with PowerPC 7400, which is also a G4 CPU. Though the CPUs are different, they all belong to the same PowerPC family. This quick comparison demonstrates the speed-advantage of NH.

We note here that GHASH is claimed to have the ability to be computed incrementally, which makes GHASH a candidate for being used in our scheme. However, the incremental updating of GHASH requires power operations in GF. These operations are quite expensive especially when compared to the addition and multiplication operations needed by the incremental updates of our universal hash. Thus, we can claim that universal hashing is more suitable for our scheme in terms of overall speed and better incremental performance.

Finally, we note that the hardware implementation of GHASH works better than software. Therefore, it is quite interesting to carry out a detailed comparison between the GHASH scheme and the universal hashing schemes in terms of hardware performance. We suggest such a comparison for future work as it falls outside the scope of this paper.

6.2 Test Result

To evaluate the performance of the proposed protocol we implemented the scheme in software. We develop a simple two level Merkle Tree with different combination 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 the AES are computed in parallel. The simulations were carried out on an Intel Core 2 machine with a processor speed of 1.67 GHz.

We tested the performance of the proposed schemes along with the SHA-1 based scheme. The SHA-1 based Merkle Tree will produce 160-bit tags which is shorter than that in our schemes. However, the proposed schemes still have a large advantage over it. Firstly, we tested the speed of different operations of the proposed schemes when the cache is always hit. This is achieved by letting the application access the same location of the memory so that the cache will always hit. As comparison, the speed when the cache will always miss is also tested. The basic scheme and the SHA-1 based scheme do not use caches, thus the speed of them does not change whether the cache hits or not. Then we tested the average performance for different cache hit rate. This is achieved by letting the application access memory locations every several words. With the growth of the step, the cache performance will get worse.

As shown in 6.1, even the basic scheme is much faster than SHA-1 based Merkle Tree. The scheme with hash cache can even reduce the update time to less than 100 cycles. In the case that the capability of the attacker is limited so that the attacker cannot control nor predict the data, such speed is quite promising. However, in general case where a check is needed before each access, the scheme with randomizers performances better since it can save the costly check operation.

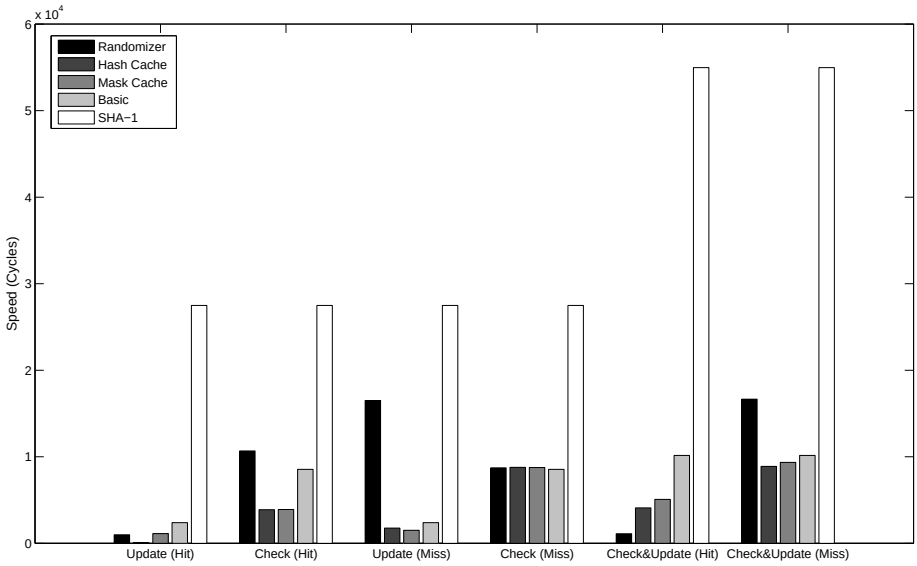

Figure 6.1: Performance Comparison

We can see clearly from the 6.2 that the performance of the improved schemes depends heavily on the cache hit rate. Among them the scheme with randomizers suffers most from low cache hit rates. This is because when the tags are going to leave the cache in the scheme with randomizers, a check is still needed and the new randomizers need to be prepared. Thus, it is not a good choice to implement the scheme with randomizers when the cache hit rate is low.

Overall, we can conclude that the new proposed schemes have a significant improvement in performance compared to the SHA-1 based scheme. When the caches work well, the improved schemes can achieve even higher speed.

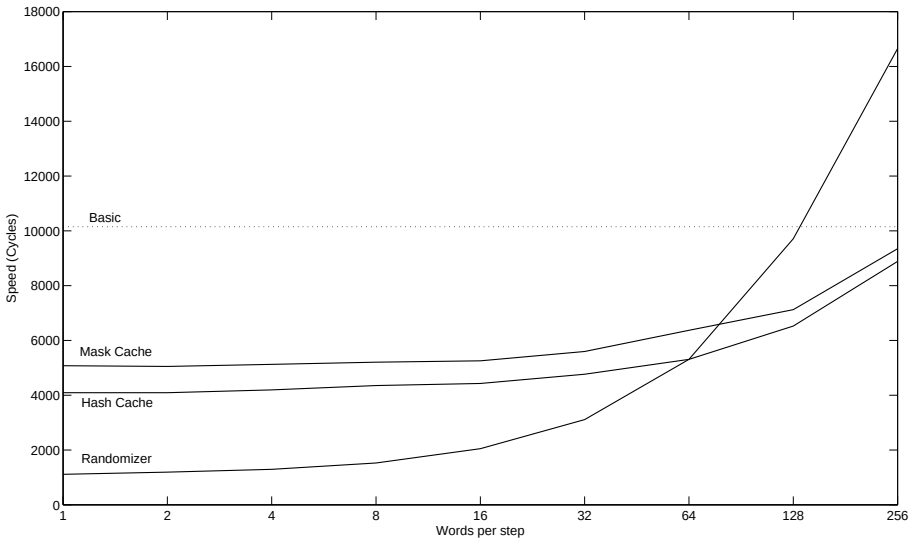

Figure 6.2: Cache Performance

Chapter 7

Conclusions

In this thesis, a new scheme to provide dynamic memory protection is proposed. The new scheme is based on universal hash functions and makes use of Merkle trees as in previous proposals to prevent replay attacks. The security of the proposed protocol is based on well established provable collision resistance properties of universal hash functions.

The proposed memory integrity protection schemes enjoy two properties of the NH family of universal hash functions: the extreme compression speed, and the ability to incrementally update hashes. As opposed to previous proposals which achieve coarse grain updates to the authentication tags via the Merkle tree construction, the universal hashing scheme allows much finer grain incremental update operations. This gives a tremendous saving in the overhead normally incurred in authenticated memory updates. Our simulation results showed more than an order of magnitude speedup over SHA-1 based schemes.

The proposed scheme also enjoys a significant speedup in the memory integrity check time, mainly due to the performance advantage of the universal hashing scheme over traditional hashing schemes. For the same setting as above the checking

times were more than 4 times better than that of the SHA-1 based scheme.

Furthermore, several possible ways to improve the performance of the basic scheme are proposed. A combination of those techniques in the simulations improved the basic scheme by about a factor of five. Clearly, more protected memory needs to be used to maintain the caches, and the employment of the improvements will depend on the requirements of the application and on the availability of resources.

7.1 Further Research

In this thesis, the proposed scheme is implemented and tested by software and it performs well. However, the efficiency could be even higher if the scheme is implemented by hardware due to the better supports for parallelism. So it is quite meaningful to implement this scheme by hardware and trying to make the efficiency closer to practical demands.

Also, in this thesis, the secure storage space is provided by assumption. However, there is some ways to provide such space with some extra overhead using current technology. It is also advisable to implement the scheme with these methods to test the real overhead of this scheme with software implementation.

Bibliography

[1] Jun Yang, Lan Gao, Youtao Zhang. Improving Memory Encryption Performance in Secure Processors. *IEEE Transactions on Computers*, VOL. 54, NO. 5, MAY 2005

[2] Ralph C. Merkle. Protocols for Public Key Cryptosystems. *Proceedings of the 1980 IEEE Symposium on Security and Privacy*, 1980.

[3] Chenyu Yan, Brian Rogers, Daniel Englender, Yan Solihin, Milos Prvulovic. Improving Cost, Performance, and Security of Memory Encryption and Authentication. *ISCA '06*.

[4] D. Clarke, S. Devadas, B. Gassend, M. van Dijk, and G. E. Suh, Incremental Multiset Hash Functions and Their Application to Memory Integrity Checking, Proceedings of the 2003 Asiacrypt Conference, November 2003.

[5] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas, Caches and Hash Trees for Efficient Memory Integrity Verification, Proc. Ninth Int'l Symp. High Performance Computer Architecture (HPCA9), Feb. 2003.

[6] D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk and S. Devadas, Toward Constant Bandwidth Overhead Memory Integrity Verification, Proceedings of the IEEE Symposium on Security and Privacy, May 2005.

[7] Ralph C. Merkle Secrecy, authentication, and public key systems Ph.D. thesis, Electrical Engineering, Stanford, 1979.

[8] Mihir Bellare, Oded Goldreich, Shafi Goldwasser Incremental Cryptography: The Case of Hashing and Signing In *Advances in Cryptology - CRYPTO'94*, Lecture Notes in Computer Science, No. 839, pages 216–233. Springer-Verlag, 1994.

[9] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, P. Rogaway. UMAC: Fast and Secure Message Authentication. In *Advances in Cryptology - CRYPTO '99*

[10] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, M. Bardouillet, C. Buatois, J. B. Rigaud. Hardware Engines for Bus Encryption: a Survey of Existing Techniques. *DATE '05*

[11] Jonathan M. McCuney Bryan Parnoy Adrian Perrigy Michael K. Reiteryz Hiroshi Isozakiyx. Flicker: An Execution Infrastructure for TCB Minimization. *EuroSys'08*

[12] Helena Handschuh and Bart Preneel. Key-Recovery Attacks on Universal Hash Function Based MAC Algorithms. *CRYPTO 2008*

[13] Jens-Peter Kaps, Kaan Yuksel, and Berk Sunar. Energy Scalable Universal Hashing. IEEE Transactions on Computers, volume 54, number 12, pages 1484-1495, December, 2005.

[14] G. Brassard. On computationally secure authentication tags requiring short secret shared keys. In D. Chaum, R. L. Rivest, and A. T. Sherman, editors, *Advances in Cryptology - CRYPTO '82*, Lecture Notes in Computer Science, pages 79–86, New York, 1983. Springer-Verlag.

[15] M. Etzel, S. Patel, and Z. Ramzan. SQUARE HASH: Fast message authentication via optimized universal hash functions. In M. Wiener, editor, *Advances in Cryptology - CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 234–251, New York, 1999. Springer-Verlag.

[16] S. Halevi and H. Krawczyk. MMH: Software message authentication in the gbit/second rates. In *4th Workshop on Fast Software Encryption*, volume 1267 of *Lecture Notes in Computer Science*, pages 172–189. Springer, 1997.

[17] J. M. Kahn, Katz R. H., and K. S. J. Pister. Next century challenges: mobile networking for "smart dust". In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278. ACM, 1999.

[18] H. Krawczyk. LFSR-based hashing and authentication. In *Advances in Cryptology - CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 129–139. Springer-Verlag, 1994.

[19] H. Krawczyk. New hash functions for message authentication. In *EUROCRYPT'95*, volume 921 of *Lecture Notes in Computer Science*, pages 301–310. Springer-Verlag, 1995.

[20] Y. Mansour, N. Nissan, and P. Tiwari. The computational complexity of universal hashing. In *22nd Annual ACM Symposium on Theory of Computing*, pages 235–243. ACM Press, 1990.

[21] W. Nevelsteen and B. Preneel. Software performance of universal hash functions. In *EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 24–41, Berlin, 1999. Springer-Verlag.

[22] V. Shoup. On fast and provably secure message authentication based on universal hashing. In *Advances in Cryptology - CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 74–85, New York, 1996. Springer-Verlag.

[23] P. Rogaway. Bucket hashing and its application to fast message authetication. In D. Coppersmith, editor, *Proceedings Crypto '95*, volume 963 of *LNCS*, pages 29–42. Springer-Verlag, 1995.

[24] J. L. Carter and M. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1978.

[25] J. L. Carter and M. Wegman. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

[26] G. Hoglund and G. McGraw. Exploiting Online Games: Cheating Massively Distributed Systems. *Addison-Wesley Professional*, 2007.

[27] David A. McGrew and John Viega The Galois/Counter Mode of Operation (GCM) *Submission to NIST Modes of Operation Process, January*, volume 15, 2004.

[28] Kaan Yüksel, Jens-Peter Kaps, and Berk Sunar Universal Hash Functions for Emerging Ultra-Low-Power Networks *Proc. of CNDS*, 2004.