

Efficient authentication of large, dynamic data sets using Galois/Counter Mode (GCM)

David McGrew
Cisco Systems, Inc.
510 McCarthy Blvd..
Milpitas, CA 95035
mcgrew@cisco.com

Abstract—The Galois/Counter Mode (GCM) of operation can be used as an incremental message authentication code (MAC); in this respect, it is unique among the crypto algorithms used in practice. We show that it has this property, and show how to use it as an incremental MAC. These MACs have great utility for protecting data at rest. In particular, they can be used to protect a large, dynamic data set using only a small, constant amount of memory.

I. INTRODUCTION

The Galois/Counter Mode (GCM) of Operation for block ciphers is an authenticated encryption algorithm that has been developed to protect high-speed packet flows [1], [2]. It has been adopted by industry and appears in several standards. It uses a Carter-Wegman style message authentication code [3], in which a universal hash function is applied to the data, and the resulting value is encrypted. When the authentication component of GCM is used without encryption, it is called Galois Message Authentication Code (GMAC). This algorithm is *incremental*: after computing the MAC of message M , the computational cost of computing a message M' that is close to M is proportional to the hamming weight between those messages [4]. This property of GMAC follows from the algebraic properties of its hash function; it is extremely useful for protecting data at rest. In contrast, conventional message authentication codes (MACs) such as HMAC [5] and CBC-MAC [6] provide no such shortcuts. Other MACs that use similar hash functions to GMAC also share its incremental properties, though these properties have not been previously noted. The idea of an incremental MAC is well understood by the theoretical community, and practical designs have appeared [7], though the authors are unaware of any such

MACs in real-world use. Our contribution is to show how a standardized MAC can be used in this manner.

One of the outstanding problems is the case in which a client with a small, constant amount of storage must authenticate a large, dynamic data set stored on an untrusted server. Considerable work has been done showing how Merkle hash trees can be used to this end. However, the computation, communication, and storage of the elements of those trees adds complexity and cost. An incremental MAC provides a simpler alternative.

In the following, we review the definition of GCM and GMAC (Section II), show how GMAC is incremental (Section III), and illustrate how it can be applied to authenticate a large data set partitioned into short blocks (Section IV). We describe several authenticated storage applications (Sections V and VI), mention several important security aspects (Section VII), and conclude with a comparison between the MACs mentioned above (Section VIII).

II. GCM DEFINITION

We briefly review the definition of GCM, closely following its specification [1], but considering a block cipher with a width of $w \geq 64$ bits, instead of focusing on the 128-bit wide Advanced Encryption Standard (AES) [8]. We assume that w is even. The two main functions that GCM uses are block cipher encryption and multiplication over the field $GF(2^{128})$; it defines a particular field, but its details are irrelevant to our analysis. The block cipher encryption of the value $X \in \{0, 1\}^w$ with the

key K is denoted as $E(K, X)$. The multiplication of two elements $X, Y \in GF(2^{128})$ is denoted as $X \cdot Y$, and the addition of X and Y is denoted as $X \oplus Y$. The function $\text{len}(S)$ takes a bit string S with a length between zero and $2^{w/2} - 1$, inclusive, and returns a $w/2$ -bit string containing the nonnegative integer describing the number of bits in its argument, with the least significant bit on the right. The expression 0^l denotes a string of l zero bits, and $A\|B$ denotes the concatenation of two bit strings A and B . The function $\text{MSB}_t(S)$ takes a bit string S and returns the bit string containing only the leftmost t bits of S , and the symbol $\{\}$ denotes the bit string with zero length.

The authenticated encryption operation takes as inputs a secret key K , initialization vector IV , a plaintext P , and additional authenticated data A , and gives as its outputs a ciphertext C and an authentication tag T . These values are bit strings with lengths given as follows:

$$\begin{aligned} 0 \leq \text{len}(P) &\leq (2^{32} - 2)w \\ 0 \leq \text{len}(A) &\leq 2^{w/2} \\ 0 < \text{len}(IV) &\leq 2^{w/2} \\ \text{len}(C) &= \text{len}(P) \\ \text{len}(T) &= t \leq w, \end{aligned} \quad (1)$$

where the parameter t is fixed for each instance of the key. The secret key has a length appropriate to the block cipher, and is only used as an input to that cipher. For each fixed value of K , each value of the IV must be distinct, but those values need not have equal lengths. The authenticated decryption operation has five inputs: K, IV, C, A , and T , as defined above. It has only one output, either the plaintext value P or the special symbol **FAIL** that indicates that its inputs are not authentic.

During the encryption and decryption processes, the bit strings P, C , and A are broken up into w -bit blocks. We let n and u denote the unique pair of positive integers such that the total number of bits in the plaintext is $(n - 1)w + u$, where $1 \leq u \leq w$, when $\text{len}(P) > 0$; otherwise $n = u = 0$. The plaintext consists of a sequence of n bit strings, in which the bit length of the last bit string is u , and the bit length of the other bit strings is w . The sequence is denoted $P_1, P_2, \dots, P_{n-1}, P_n^*$, and the bit strings are called data blocks, although the last bit string, P_n^* , may not be a

complete block. Similarly, the ciphertext is denoted as $C_1, C_2, \dots, C_{n-1}, C_n^*$, where the number of bits in the final block C_n^* is u . The additional authenticated data A is denoted as $A_1, A_2, \dots, A_{m-1}, A_m^*$, where the last bit string A_m^* may be a partial block of length v , and m and v denote the unique pair of positive integers such that the total number of bits in A is $(m - 1)w + v$ and $1 \leq v \leq w$, when $\text{len}(A) > 0$; otherwise $m = v = 0$. The authenticated encryption operation is defined by the following equations:

$$\begin{aligned} H &= E(K, 0^w) \\ Y_0 &= \begin{cases} IV\|0^{31}1 & \text{if } \text{len}(IV) = w - 32 \\ \text{GHASH}(H, \{\}, IV) & \text{otherwise.} \end{cases} \\ Y_i &= \text{incr}(Y_{i-1}) \text{ for } i = 1, \dots, n \\ C_i &= P_i \oplus E(K, Y_i) \text{ for } i = 1, \dots, n - 1 \\ C_n^* &= P_n^* \oplus \text{MSB}_u(E(K, Y_n)) \\ T &= \text{MSB}_t(\text{GHASH}(H, A, C) \oplus E(K, Y_0)) \end{aligned} \quad (2)$$

Successive counter values are generated using the function $\text{incr}()$, which treats the rightmost 32 bits of its argument as a nonnegative integer with the least significant bit on the right, and increments this value modulo 2^{32} . More formally, the value of $\text{incr}(F\|I)$ is $F\|(I + 1 \bmod 2^{32})$. The function GHASH is defined by $\text{GHASH}(H, A, C) = X_{m+n+1}$, where the inputs A and C are formatted as described above, and the variables X_i for $i = 0, \dots, m + n + 1$ are defined as

$$X_i = \begin{cases} 0 & \text{for } i = 0 \\ (X_{i-1} \oplus A_i) \cdot H & \text{for } i = 1, \dots, m - 1 \\ (X_{m-1} \oplus (A_m^* \| 0^{w-v})) \cdot H & \text{for } i = m \\ (X_{i-1} \oplus C_{i-m}) \cdot H & \text{for } i = m + 1, \dots, m + n - 1 \\ (X_{m+n-1} \oplus (C_n^* \| 0^{w-u})) \cdot H & \text{for } i = m + n \\ (X_{m+n} \oplus (\text{len}(A) \| \text{len}(C))) \cdot H & \text{for } i = m + n + 1. \end{cases} \quad (3)$$

The function GMAC is a special case of GCM in which $C = \{\}$. We denote the application of GMAC with key K , initialization vector IV , and message M as $\text{GMAC}(K, IV, M)$.

In the following, for notational simplicity, we use H to denote the hash key that is derived from the GMAC key K as in Equation 2.

III. INCREMENTAL AUTHENTICATION

Given a message M and a corresponding tag $T = \text{GMAC}(K, IV, M)$, we can efficiently compute the tag T' for a new message M' , with computational effort proportional to the hamming distance between M and M' (that is, the number of nonzero bits in $M \oplus M'$). GMAC inherits this attribute from GHASH. Before we analyze that function, we first show how GMAC can be decomposed into two lower-level functions:

$$\text{GMAC}(K, IV, M) = \text{GPRF}(K, IV) \oplus \text{GHASH}(H, M, \{\}). \quad (4)$$

The function GPRF is the pseudorandom function used to encrypt the output of the hash function; its definition is obvious and we leave it implicit.

A. Algebraic Properties of GHASH

The function $\text{GHASH}(H, A, C)$ has the property that it is linear in terms of its arguments A and C .

Theorem 1 (linearity): For any $H \in \{0, 1\}^w$ and any bit strings A, A', C and C' such that $\text{len}(A) = \text{len}(A')$ and $\text{len}(C) = \text{len}(C')$,

$$\text{GHASH}(H, A, C) \oplus \text{GHASH}(H, A', C') = \text{GHASH}(H, A \oplus A', C \oplus C'). \quad (5)$$

Proof: We consider the evaluation of $\text{GHASH}(H, A, C)$ and $\text{GHASH}(H, A', C')$, when the lengths are restricted as in the theorem. Let X_i be as defined in Equation 3, and let X'_i be defined similarly, but with $X'_i, A'_i,$ and C'_i replacing $X_i, A_i,$ and C_i , respectively. Then define δX_i to be $X_i \oplus X'_i$, and define δA_i and δC_i equivalently. Using the fact that each case in Equation 3 is linear,

$$\delta X_i = \begin{cases} 0 & \text{for } i=0 \\ (\delta X_{i-1} \oplus \delta A_i) \cdot H & \text{for } i=1, \dots, m-1 \\ (\delta X_{m-1} \oplus (\delta A_m^* \| 0^{w-v})) \cdot H & \text{for } i=m \\ (\delta X_{i-1} \oplus \delta C_{i-m}) \cdot H & \text{for } i=m+1, \dots, m+n-1 \\ (\delta X_{m+n-1} \oplus (\delta C_n^* \| 0^{w-u})) \cdot H & \text{for } i=m+n \\ (\delta X_{m+n} \oplus (\text{len}(A) \| \text{len}(C))) \cdot H & \text{for } i=m+n+1. \end{cases} \quad (6)$$

Thus $\text{GHASH}(H, A, C) \oplus \text{GHASH}(H, A', C') = \delta X_{m+n+1} = \text{GHASH}(H, A \oplus A', C \oplus C')$, since the variables δA_i and δC_i are equivalent to the block decomposition of $A \oplus A'$ and $C \oplus C'$, respectively. ■

It is also possible to efficiently compute the value of GHASH applied to one string appended to another string, given the GHASH values of those strings, provided that some alignment restrictions are met. This property is captured in Lemma 2.

Lemma 2 (appending and prepending): For any $H \in \{0, 1\}^w$, any bitstring A with $\text{len}(A) < 2^{64}$, and any P such that $\text{len}(P) = lw$ for some value of l , the value of GHASH applied to $P\|A$ can be computed as

$$\begin{aligned} \text{GHASH}(H, P\|A, \{\}) &= \text{GHASH}(H, A, \{\}) \oplus \\ &H^a \cdot (\text{GHASH}(H, P, \{\}) \oplus H \cdot (\text{len}(P) \| 0^{64})) \oplus \\ &H \cdot ((\text{len}(A) \oplus \text{len}(P\|A)) \| 0^{64}) \end{aligned}$$

where $a = \lceil \text{len}(A)/w \rceil$ denotes the number of w -bit blocks in the block decomposition of A .

IV. BLOCK STORAGE

One important case is when the data A that is authenticated is formatted as a sequence of fixed-length blocks B_1, B_2, \dots, B_l , where the length of each block is wq for some value of q . In this case, GHASH can be computed as

$$\begin{aligned} \text{GHASH}(H, B_1\|B_2\|\dots\|B_l, \{\}) &= \\ H \cdot \left((\text{len}(A) \| 0^{w/2}) \oplus \bigoplus_{i=0, l-1} H^{iq} \cdot h(B_i) \right) & \quad (7) \end{aligned}$$

where the function h is a degree q polynomial in H .

When the j^{th} block has been changed from B_j to B'_j , the new value of the entire hash can be computed by adding $H^{qj} \cdot h(B_j \oplus B'_j)$ to that value. The value of H^{qj} can be computed efficiently by using the repeated square and multiply algorithm:

$$\begin{aligned} X &\leftarrow H^q, Y \leftarrow 1 \\ \text{for } i \text{ from } 0 \text{ to } \lceil j \rceil \text{ do} \\ &\quad \text{if the } i^{\text{th}} \text{ bit of } j \text{ is one then} \\ &\quad \quad Y \leftarrow Y \cdot X \end{aligned}$$

```

    X ← X · X
  end if
end for
return Y

```

This algorithm requires no more than $\lceil j \rceil$ squarings and multiplies (in $GF(2^w)$). For example, with 256 gigabytes data broken into 512-byte blocks, no more than 31 squares and multiplies are needed. In this case the computational cost for ‘seeking’ into the j^{th} block is comparable to the cost of computing the hash $h(B_j)$. Various time/memory tradeoffs can reduce this cost further.

A. Operations

GMAC supports incremental tag generation for several different types of message edits: changes within a fixed-length message, appending or prepending data to a message, and truncating data from the start or end of a message. Theorem 1 can be used to reduce the amount of computation that needs to be done to compute $\text{GMAC}(K, IV, M')$, if the value $T = \text{GMAC}(K, IV, M)$ is known, $\text{len}(M) = \text{len}(M')$ and M' is close to M in a hamming sense. Then $T' = \text{GMAC}(K, IV', M')$ can be computed as

$$T' = T \oplus \text{GPRF}(K, IV) \oplus \text{GPRF}(K, IV') \oplus \text{GHASH}(H, M \oplus M', \{ \}). \quad (8)$$

From Equation 6 we can see that several operations can be skipped for each i such that $\delta A_i = 0$. We show this in detail in the next section.

Lemma 2 can be used to reduce the work needed to compute the tag T' for the message $M' = P||M$ if the value $T = \text{GMAC}(K, IV, M)$ is known. Then $T' = \text{GMAC}(K, IV', M')$ can be computed as

$$T' = \text{GPRF}(K, IV') \oplus (\text{GPRF}(K, IV) \oplus T) \oplus H^a \cdot (\text{GHASH}(H, P, \{ \}) \oplus H \cdot (\text{len}(P) \parallel 0^{64})) \oplus H \cdot ((\text{len}(A) \oplus \text{len}(P||A)) \parallel 0^{64}) \quad (9)$$

where $a = \lceil \text{len}(M)/w \rceil$. Appending a string to a message can be handled similarly.

When the tag T for a message $M = P||M'$ is known, the tag for the message M' with the prefix P truncated away can be computed efficiently, using a method like that of Equation 9. That method also allows to efficiently compute the tag for a message after its suffix has been truncated.

V. AUTHENTICATED STORAGE

One important application for incremental MACs is the authentication of a large, dynamic data set stored on an untrusted server, using only a small, constant amount of local state within the security boundary. The data to be authenticated is divided up into parts, and each part is associated with a secret key K , an authentication tag T , and a initialization vector IV . Multiple data parts may use the same secret key, but when they do so, the system must ensure that the initialization vectors used by all of the data parts are distinct. This can be done by reserving some of the bits of IV to serve as an identifier of the data part to which it corresponds. Since each data part is otherwise treated independently, we assume without loss of generality that there is only one data part, which we call ‘the data’.

An authentication tag for the data is generated before the data is written into the untrusted store. The values of K , IV , and T are kept within the security boundary. The values of IV and T need not be kept secret, but keeping them inside the boundary ensures that an attacker cannot present an earlier instance of the data, IV , and tag, and thus undetectably ‘turn back the clock’. When the data is read from the remote store, and the authenticity of the data is checked as usual, using the values stored within the security boundary. When data is to be written into the trusted store, the authentication tag is recomputed as described in Section IV-A, using the value of T stored inside the security boundary. The need to use the current tag value to compute new tag values is the reason why we store it within the boundary.

A. Initialization

Before it is used, the authenticated data store must be initialized by bringing it into a consistent state. The

simple way to do this is to compute the MAC over the block device, thus establishing the initial state of T . However, a more efficient method is possible, based on a property of GHASH. When that function is applied to data that contains only zero bits, the result can be computed easily, and is merely a function of the length of message:

$$\text{GHASH}(H, A, \{\}) = H \cdot (\text{len}(A) \parallel 0^{64}). \quad (10)$$

A secure block device can be efficiently initialized by setting the entire block device to all zeroes, and then computing T using Equation 10. In some data storage systems, data blocks that have been written can be distinguished from data block that have not yet been written. In these systems, the zeroization step could be skipped entirely, by using the assumption that not-yet-written blocks are all zero.

VI. APPLICATIONS

Example applications for authenticated storage include tape drives, disk drives, disk partitions, file systems, logging systems, and sets of files. We outline these applications below.

A. Disks and Tapes

For example, a disk that used GMAC authentication can compute the correct value of the tag, updating the IV and tag each time that a new disk block is written. The authentication module would read the old value of the block before it is written, then use that value to compute the new value of the tag. Tag verification would take place prior to mounting the disk, or perhaps after the disk has been mounted. One option would be to mount the disk in read-only mode while the disk is being verified, then to re-mount it after it has successfully passed its authentication check. Because verification is relatively costly, it would be beneficial to authenticate different partitions separately. If the authentication check fails, the mount operation can fail, or an appropriate alert can be generated.

Protection for disks and tapes is especially important when they leave physically secured locations. Relocation of data often takes place for archival purposes, or

because of the need for repair. In some cases, a data store may need to operate in an untrusted location; a kiosk or public workstation is a good example of this case. In this situation, a trusted user may periodically access the data store, perhaps over a network connection; this user can verify the authenticity of the store.

Several disk-level encryption systems have been described [9], [10]. However, these systems do not provide any strong data authentication. Authenticating an encrypted disk partition provides significant benefits. Many encryption systems are vulnerable to chosen-ciphertext attacks, in which an attacker who cleverly manipulates ciphertext data prior to its decryption can cause a system to unintentionally reveal information about the plaintext. Adding an authentication check protects against chosen-ciphertext attacks by detecting the ciphertext manipulations. Thus adding authentication to an encrypted file system may improve its ability to provide confidentiality as well as its ability to provide authentication.

B. Files

The incremental operations that GMAC supports provide a good match to the POSIX file operations programming interface of `open`, `read`, `write`, and `lseek` [11]. The write operation corresponds either to changing a current data block, or to appending additional data. The seek operation corresponds, roughly, to the exponentiation used to compute the appropriate power of the hash key H . Tag verification can be done during the open operation. In a journaled file system, the file system stores the information needed to retrieve an older version of a file; this capability can be used to return a file to the most recent version that passes the authentication check.

Some of the GMAC operations have alignment restrictions. Block storage devices already conform to these requirements, but the POSIX file operations do not. These facts suggest that the appropriate layer at which to add authentication is inside of the file system itself, which is aware of the block nature of the underlying storage. Alternatively, authentication can be provided at the application layer, though possibly with degraded efficiency.

File system monitoring tools such as tripwire [12] detect when files have changed by computing a ‘fingerprint’ of these files. The fingerprint of a file is checked against a reference value before that file is used. These tools are applied to protect system files and other security-critical data. These fingerprints can be computed using an incremental MAC, which allows the tool to efficiently compute new reference values when system files are changed by a legitimate user.

There are encrypted file systems that lack authentication. Adding authentication to such a system can provide it with better protection against chosen ciphertext attacks, as described above for encrypted disks.

C. Key Management

In order to transition from one key K to another key K' , it is necessary to compute the authentication tag with the new key. The entire secured data store must be processed in this computation; there is no ‘shortcut’ available.

One practical way to transition to a new key is to combine the validation step with the computation of the new key. As the data is streamed through the crypto processor, the tag corresponding to the new key can be computed in parallel with the tag corresponding to the old key. After verifying the validity of the latter tag, the key can be switched over to the new one.

VII. SECURITY

It is essential for GMAC’s security that each distinct authentication tag is generated using a distinct initialization vector. GMAC provides excellent security when its requirement for distinct IVs is respected, but provides no security when this requirement is not met. Throughout this paper, we have shown how to compute a second tag T' from a first tag T ; we emphasize that the second tag *must* use an initialization vector IV' that is distinct from the initial one IV . As long as this basic requirement is respected, the methods that we have presented for computing the second tag T' conform to the GMAC specification; thus there are no security implications in the use of these methods.

When using GMAC to authenticate large messages, it is essential to use large authentication tags. This is because the strength of the MAC degrades slightly as a function of the length of the messages [13], [14]. For this reason, we recommend that the GMAC be used with tag size that is as large as possible. For AES GMAC, the largest possible tag is 128 bits.

VIII. CONCLUSIONS

Authenticated data stores provide useful security in several scenarios, and incremental MACs make it feasible to apply authentication to disks, partitions, tapes, and file systems. GMAC provides the incremental operations needed for these applications.

A. Comparison

A detailed comparison between MACs is warranted. GMAC provides efficient ways to authenticate a large set of data, by enabling new authentication tags to be computed after a small change is made to a data location, or data is appended, prepended, or truncated, with some minor alignment restrictions on the latter operations. This efficiency is in contrast to the situation with HMAC and CBC-MAC. Because these MACs use internal chaining with a nonlinear update function, the only incremental operation that they can support is to append data, or to truncate data from the end of a message. However, to support this operation with HMAC or CBC-MAC requires that the internal state of the MAC prior to the tag-generation be stored, since it is impossible to compute this internal state from the authentication tag (except for CBC-MAC, when it is used without truncating the authentication tag).

GMAC is somewhat less efficient than XORMAC [7] when generating tags for very small edits, because GMAC requires an exponentiation in order to compute the power of H used to process the edit. XORMAC has no such step; it uses only a number of operations that is proportional to the number of w -bit blocks that have changed. However, XORMAC’s advantage over GMAC is slight when both MACs are applied to block-structured data such as a disk with 512-byte blocks. Ad-

ditionally, XORMAC requires a block cipher invocation to process each w -bit block, while GMAC uses only a multiplication over $GF(2^w)$, making GMAC more efficient for larger edits. XORMAC does not support the prepending of data, nor the truncation of data from the start of a message. Like GMAC, XORMAC has alignment restrictions.

B. Incremental Verification

GMAC easily supports incremental computation of tags, as shown above. However, does not directly support incremental *verification* of tags. That is, it does not provide a way to verify just a single block out of a large set of blocks; it is defined only as a function of the entire message. The verification of a single data block can be done with a *memory checker* [15], [16]. These functions are potentially useful. GMAC cannot directly be used as a memory checker; it may be possible to define a memory checker based on it, but that work is outside the scope of this paper.

REFERENCES

- [1] D. McGrew and J. Viega, "The Galois/Counter Mode of Operation (GCM)," *Submission to NIST Modes of Operation Process*, January, 2004.
- [2] D. McGrew and J. Viega, "The Security and Performance of the Galois/Counter Mode (GCM) of Operation," *INDOCRYPT '04*, LNCS, Springer-Verlag, 2004.
- [3] M. Wegman and L. Carter, "New hash functions and their use in authentication and set equality," *Journal of Computer and System Sciences*, 22:265279, 1981.
- [4] M. Bellare, O. Goldreich, and S. Goldwasser, "Incremental cryptography: the case of hashing and signing", *CRYPTO '94*, LNCS, Springer-Verlag, Aug. 1994.
- [5] M. Bellare, R. Canetti, and H. Krawczyk, Keying hash functions for message authentication, *Crypto '96* LNCS 1109, Springer-Verlag, 1996.
- [6] M. Dworkin, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, NIST Special Publication 800-38B, 2005.
- [7] M. Bellare and P. Rogaway, XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions, *CRYPTO '95*, LNCS 963, Springer-Verlag, 1995.
- [8] U.S. National Institute of Standards and Technology. The Advanced Encryption Standard. *Federal Information Processing Standard (FIPS) 197*, 2002.
- [9] Draft Proposal for Tweakable Narrow-block Encryption, IEEE P1619. Work in progress. August, 2004.
- [10] J. Black, P. Rogaway, A Block Cipher Mode of Operation for Parallelizable Message Authentication, *Eurocrypt '02*, LNCS, Springer-Verlag, 2002.
- [11] POSIX.1, IEEE Standard 1003.1, 1988.
- [12] G. Kim and E. Spafford", The Design and Implementation of Tripwire: A File System Integrity Checker, *ACM Conference on Computer and Communications Security*, pg. 18-29, 1994.
- [13] N. Ferguson, Authentication weaknesses in GCM. Comments to the NIST Modes of Operation process. 2005.
- [14] D. McGrew and J. Viega, GCM Update. Comments to the NIST Modes of Operation process. 2005.
- [15] Manuel Blum and William S. Evans and Peter Gemmell and Sampath Kannan and Moni Naor, "Checking the Correctness of Memories", *IEEE Symposium on Foundations of Computer Science*, pg. 90-99, 1991.
- [16] M. Fischlin, Incremental Cryptography and Memory Checkers, *Eurocrypt '97*, LNCS 1233, Springer-Verlag, 1997.