

How to Forget a Secret

(Extended Abstract)

GIOVANNI DI CRESCENZO¹

NIELS FERGUSON²

RUSSELL IMPAGLIAZZO¹

MARKUS JAKOBSSON³

¹ Computer Science Department, University of California San Diego,
La Jolla, CA, 92093-0114. E-mail: giovanni.russell@cs.ucsd.edu
(Part of Giovanni's work done while at Bellcore)

² Counterpane Systems. E-mail: niels@counterpane.com
(Part of this work done while visiting UCSD)

³ Information Sciences Research Center, Bell Labs, Murray Hill, NJ 07974.
E-mail: markusj@research.bell-labs.com
(Part of this work done while at UCSD)

Abstract. We uncover a new class of attacks that can potentially affect *any* cryptographic protocol. The attack is performed by an adversary that at some point has access to the physical memory of a participant, including all its previous states.

In order to protect protocols from such attacks, we introduce a cryptographic primitive that we call *erasable memory*. Using this primitive, it is possible to implement the essential cryptographic action of *forgetting* a secret. We show how to use a small erasable memory in order to transform a large non-erasable memory into a large and erasable memory. In practice, this shows how to turn any type of storage device into a storage device that can selectively forget. Moreover, the transformation can be performed using the minimal assumption of the existence of any one-way function, and can be implemented using any block cipher, in which case it is quite efficient. We conclude by suggesting some concrete implementations of small amounts of erasable memory.

1 Introduction

All of cryptography is based on the assumption that some information can be kept private, accessible only by certain parties. At some level, physical control over the storage device containing the information must be maintained. While it is reasonable to assume that such control can be maintained for some finite amount of time, as the duration of a protocol, it is not realistic to assume it continues indefinitely. Thus, it is important to be able to *forget* information.

Practical considerations. Current computer systems treat erasing memory only from the point of view of efficiency, not security. Erasing a memory location enables the system to use that location for other purposes; it does not guarantee

that the information is destroyed. In fact, in many practical systems the old value might still be retrievable, making the assumption that data is forgotten, if not specifically remembered, a *fallacy*. For example, it has been observed that if a message is stored for a long time in some types of RAM, the chip will slowly ‘learn’ this value. Overwriting the data or powering down the chip will not erase it. When the chip is powered up again there is a good chance that the data will again appear in the RAM [4]. Some of these effects are due to aging effects in the silicon that depend on the bits stored in the RAM. Another example is the case of magnetic media. Here, the problem is more severe, since these devices are notoriously hard to wipe. It has long been known that simply overwriting the data is not sufficient; US government specifications call for overwriting the data *three times* for non-classified information, in order to minimize the chances of their retrieval by an adversary. However, the magnetic fields used to store the data have a tendency to migrate away from the read/write head and bury themselves deep down in the magnetic material, so even if these fields are no longer readable by the original read/write head, other techniques might well retrieve them. The problem gets even more complicated if the operating system supports virtual memory. A typical virtual memory system will write a piece of RAM to disk in a swap-file without telling the program using the RAM. When the program tries to access this RAM, the operating system reads the data from the swap file back into RAM for the program to use. Suppose the program overwrites the RAM in an attempt to erase the data and then exits. The old copy in the swap file is not erased by the operating system, but just marked as ‘available’. A direct inspection of the swap file will reveal the old values of the RAM.

Cryptographic protocols. In the security and cryptography literature, many protocols make the silent assumption that we can ‘forget’ information. For example, the randomly chosen ‘temporary secret key’ in DSS [15] and similar schemes [7, 18] should be forgotten by the signer after it has been used. If it can be found by an attacker, he will be able to reconstruct the secret key of the signer from the signature and the temporary secret key. More generally, participants to cryptographic protocols should forget all partial results, randomness values and temporarily stored information they use while executing such protocols. Otherwise this information could later fall in the hands of some adversary, who will be able to use it to attack the scheme. Clearly, this is an issue that can potentially arise in just any cryptographic protocol (apart possibly from very few exceptions). Known cryptographic techniques, such as ‘proactivization’ (e.g., [8, 13, 12, 17]), or ‘forward security’ (e.g., [6]) do not avoid the problem. The former technique maintains secrecy with respect to intruders that occasionally gain access to some of the storage devices, by changing the way secrets are shared between such devices. This, however, is of no help if the old shares cannot be forgotten. The latter techniques guarantee security with respect to adversaries that obtain the current state of the memory, but still knows nothing about the previous states (which we consider here).

Our results. We investigate methods for storing data in such a way that information can be securely forgotten. To this purpose, we put forward the new notion

of *erasable memory*, a memory which allows to definitely and reliably erase values, by keeping only the most recently stored ones. We then formalize the type of attack that an adversary can mount on any cryptographic protocol which involves storage of (secret) values on any non-erasable memory. This leads to the formal definition of a secure erasable memory implementation as a method using a small piece of erasable memory in order to transform a large piece of non-erasable memory into large *and* erasable memory. Then, our main result consists in exhibiting such a secure erasable memory implementation under the assumption of existence of any pseudo-random permutation, which is known to exist under the existence of any one-way function (using [11, 9, 14]). We can show that such assumption is minimal too. In practice, pseudo-random permutations can be implemented using a block cipher such as some composed version of DES [1] (e.g., Triple-DES). In this case, our method also seems practical and efficient: the amount of erasable memory needed is only the amount to store the key for and compute one Triple-DES application; the storage overhead of our solution is only linear, and the computational overhead per memory access is a logarithmic number (in the size of the memory) of encryptions (this can be amortized to constant in some typical cases). While one would probably not want to use our method for all data, it would be quite reasonable for the small amount of data that is important for security purposes.

Outline of the paper: We introduce our terminology and model in Section 2, we present our construction of large erasable memory in Section 3, and discuss three possible concrete realization of small erasable memory in Section 4.

2 Definitions and Model

In this section we recall the notion of pseudo-random permutations [9, 14], we introduce the model for secure erasable memory implementation and briefly discuss a first (inefficient) solution to our problem.

2.1 Pseudo-random permutations.

Pseudo-random functions have been introduced in [9]. In this paper we will use the formalization of finite pseudorandom functions given in [2] (a concrete version of the formalization in [9, 14]).

Let \mathcal{P} be the family of all permutations, and let $G = \{E(k, \cdot), D(k, \cdot)\}_k \subseteq \mathcal{P}$ be a family of permutations, where $E(k, \cdot)$ denotes a finite permutation and $D(k, \cdot)$ its inverse. Each element from G is specified by a K -bit key k ; therefore, uniformly choosing an element from G is equivalent to uniformly choosing $k \in \{0, 1\}^K$ and returning $(E(k, \cdot), D(k, \cdot))$. Let A be an algorithm with oracle access to an element from G . We let $\mathbf{E}[A^G] = \Pr[(E(k, \cdot), D(k, \cdot)) \stackrel{R}{\leftarrow} G : A^{E(k, \cdot)} = 1]$, i.e., the probability that A outputs 1 when A 's oracle is selected to be a random permutation from G . If G, G' are two families of finite permutations, we let $\text{Adv}_A(G, G') = \mathbf{E}[A^G] - \mathbf{E}[A^{G'}]$ denote the *advantage* of A in distinguishing G from G' . Here, following [9], we are considering the following game, or statistical

test. Algorithm A is given as oracle a permutation g chosen at random from either G or G' , the choice being made randomly according to a bit b . Then A will try to predict b . The advantage is $\Pr[A^g = b] - 1/2$, i.e., the amount that the probability that A 's output is correct minus the probability that a random guess for b is correct. We say that the family of permutations G is (t, q, ϵ) -*pseudorandom* if there is no A which, running in time t and making q oracle queries, is able to obtain that $\text{Adv}_A(G, \mathcal{P}) \geq \epsilon$.

2.2 Erasable Memory implementation.

By the term *memory* we will denote any type of storage device. We will then consider two main types of memory: *persistent memory* and *erasable memory*. Persistent memory doesn't reliably forget former values (i.e., it allows the retrieval of formerly stored values). In fact, in order to prove the strongest result, we make the pessimistic assumption that all values that were ever written to the persistent memory can be retrieved by an attacker. In contrast, erasable memory reliably forgets old values (i.e., at any location, only the most recently stored value can be retrieved). Due to the practical considerations previously discussed, we should think of a typical computer storage device as being entirely persistent and of the erasable memory as a different and small piece of memory having (typically) some kind of physical implementation (we discuss three possible ones later, in Section 4).

We will consider an *erasable memory implementation* as a probabilistic data structure algorithm that translates read and write operations for a logical array, called the *virtual memory*, into read and write operations to a physically implemented array, called the *physical memory*. At the beginning of the system, the physical memory is preprocessed into two parts: the erasable memory and the persistent memory, where we should think of the erasable memory as being much smaller than the persistent one. The goal of the erasable memory implementation is to transform the physical memory into a single erasable memory.

Informally speaking, an attack on an erasable memory implementation proceeds in the following way. An adversary picks two sequences of read and write operations which are not trivially distinguishable. The data structure is then simulated on both sequences, computing the final state of the erasable memory and the entire history of writes to the persistent memory. Then the two pairs of histories and final erasable memory states are sent to the adversary in a random order. The adversary tries to predict which physical memory history corresponds to which sequence of operations. The implementation is considered secure if no adversary can succeed significantly better than by a random guess.

Definition 1. Let m, e, p be integers, and let M, EM be arrays denoting, respectively, the memory and the erasable memory, and such that $|M| = m$, $|EM| = e$. Let PM be an array denoting the persistent memory, and made of p lists pm_1, \dots, pm_p , such that $pm_i = (v_{i,1}, \dots, v_{i,s_i})$ denotes the sequence of values ever written into location i of the persistent memory (i.e., $v_{i,1}$ is the less recent and v_{i,s_i} the most recent, where s_i is the number of values ever stored

into location i). Let $OP = \{\text{READ}, \text{WRITE}\}$ be the set of *memory operations*. An *instance* of a memory operation $op \in OP$ is the tuple $ins = (op; i; inp_1, \dots, inp_c; out_1, \dots, out_d)$, where argument i points to the input location, arguments inp_j are additional inputs for op and arguments out_j are outputs of op when executed on input location i and additional inputs inp_1, \dots, inp_c . We will denote as *valid* all instances of the form

1. $(\text{READ}; i; EM, PM; cont)$;
2. $(\text{WRITE}; i; EM, PM, cont; EM, PM)$;

where $i \in \{1, \dots, p\}$, $cont \in \Sigma$, for some alphabet Σ . Here, the read operation transfers into $cont$ the most recent value v_{i, s_i} previously stored at location i of array PM ; the operation possibly uses EM , and returns $cont$. The write operation, possibly using array EM , inserts $cont$ into the last position of the list associated with location i in array PM . Namely, it increments s_i by 1 and sets $v_{i, s_i} = cont$. We say that two sequences of length T of valid instances of memory operations are *T-equivalent* if they contain the same subsequence of pairs (op, i) , for $op = \text{WRITE}$ (but possibly different additional inputs or outputs, e.g. $cont$, or different subsequences with READ operations).¹

We are now ready to define a secure erasable memory implementation.

Definition 2. An *erasable memory implementation* (EMI) is a pair of probabilistic algorithms $\text{EMI} = (\text{PREPROCESS}, \text{UPDATE})$. On input a memory array M , algorithm PREPROCESS returns an erasable memory array EM and a persistent memory array PM . On input arrays EM, PM , and an instance of a memory operation ins , algorithm UPDATE checks if the operation ins is valid; if so, it returns updated arrays EM, PM ; otherwise it returns the unchanged input arrays EM, PM . Now, let A be an algorithm and let $\text{DISTINGUISH}_A(M, OP)$ be the following probabilistic experiment:

$$\begin{aligned} & \{ (EM, PM) \leftarrow \text{PREPROCESS}(M); \\ & ((ins_{0,1}, \dots, ins_{0,T}), (ins_{1,1}, \dots, ins_{1,T})) \leftarrow A(EM, PM); \\ & EM_0 \leftarrow EM; PM_0 \leftarrow PM; EM_1 \leftarrow EM; PM_1 \leftarrow PM; \\ & (EM_0, PM_0) \leftarrow \text{UPDATE}(ins_{0,i}, EM_0, PM_0), \text{ for } i = 1, \dots, T; \\ & (EM_1, PM_1) \leftarrow \text{UPDATE}(ins_{1,i}, EM_1, PM_1), \text{ for } i = 1, \dots, T; \\ & b \stackrel{R}{\leftarrow} \{0, 1\}; d \leftarrow A(EM_b, PM_b, EM_{1-b}, PM_{1-b}); \\ & \text{if } ((ins_{0,1}, \dots, ins_{0,T}), (ins_{1,1}, \dots, ins_{1,T})) \text{ are } T\text{-equivalent and } b = d \text{ then} \\ & \quad \text{return: 1 else return: 0; } \end{aligned}$$

We say that the erasable memory implementation is (T, ϵ) -*secure* if for any adversary A making T memory operations, the probability that the experiment $\text{DISTINGUISH}_A(M, OP)$ returns 1 is at most $1/2 + \epsilon$.

We note that in the above definition we are asking the adversary to perform only T -equivalent sequences of valid virtual memory operations. The reason for this

¹ We note that we can also handle different definitions of T -equivalent sequences, in different scenarios.

is to avoid the adversary to have trivial ways of distinguishing the two sequences (e.g., as for two sequences writing to different memory locations).

Related notions. The notion of *crypto-paging* [19, 20] was introduced to study the problem of hiding information from secondary storage devices, and can be used to give a preliminary solution to our problem (see Section 2.3). A problem similar to ours is considered in [3], where a method similar to crypto-paging is suggested, but without a formal model or proofs. In [10] the authors study software protection in a model where one wants to protect any information about the communication between CPU and memory, including any access patterns. In [16] the authors study the problem of storing information in memory in such a way that the location of the stored item is hidden to the observer of the communication. The fact that in our model an adversary is allowed at some time to look at the content of both the erasable memory and the memory (i.e., reading private keys and opening any encryption) makes these last two models incomparable to ours.

2.3 An inefficient solution

A simple (but inefficient) solution to the problem of constructing a secure erasable memory implementation can be obtained as an immediate adaptation of the crypto-paging concept [20], as we now show. The preprocessing operation consists in uniformly choosing a key k for a pseudo-random permutation, and encrypting all the data using k ; then the encrypted data is stored in the persistent memory, and the key k is stored in the erasable memory. In order to read a value, using the current key k in the erasable memory, the encryption of the value is decrypted and the data is recovered. In order to write a value, a new key k' for the pseudo-random permutation is uniformly chosen, the value is encrypted under key k' and the encryption is stored in the persistent memory. In order to guarantee security, the new key k' needs to be stored into the erasable memory, replacing the old key k ; moreover, all items in the memory need to be re-encrypted using the new key k' . Notice that the number of encryptions/decryptions at each memory operation is quite large (*linear* in the size of the memory); this motivates the search for more efficient solutions.

3 Our method for obtaining large erasable memory

In this section we present our construction for obtaining a secure erasable memory implementation. The main cryptographic tool we use is a family of pseudo-random permutations $\{E(k, \cdot), D(k, \cdot)\}_k$ (which, in practice, can be implemented by any block cipher). Formally, we achieve the following:

Theorem 3. Let m be the size of the memory. Given a family of pseudo-random permutations $F = \{E(k, \cdot), D(k, \cdot)\}_k$, there exists a secure erasable memory implementation $\text{EMI} = (\text{PREPROCESS}, \text{UPDATE})$, such that, for any integer $l > 1$, the following holds:

1. *Security.* If F is (T, q, ϵ) -pseudorandom, then EMI is (T', ϵ') -secure, for $T' = T - O(lT \log_l m)$, and $\epsilon' = \epsilon / (T \log_l m)$;
2. *Space Complexity.* If F has block size a and key size K , and is computable in space s , then EMI uses a persistent memory of size $\Theta(m)$ and an erasable memory of size $\Theta(a + s + K)$;
3. *Time Complexity.* Let \mathcal{D} be the distribution of the location accessed to the memory; if $E(k, \cdot), D(k, \cdot)$ can be computed in time t then, in order to process T memory operations, EMI takes time $\Theta(Ttl \log_l m)$ for an arbitrary \mathcal{D} or time $\Theta(t(T + l \log_l m))$ if \mathcal{D} returns consecutive values.

We observe that the size of the persistent memory can be any polynomial in the size of the erasable memory input to EMI. Moreover, our erasable memory implementation requires only a logarithmic (in the size of the memory) number of encryption/decryption operations in the worst case distribution over the possible memory operations, and it achieves an amortized constant number of encryption/decryption operations in the case the memory operations are made on consecutive locations. In the following, we first present an overview of our construction and then a formal description. Further issues on the efficiency of our scheme, discussions about dividing the write operation into a write and an erase stage, and a proof that our construction satisfies Theorem 3 are in [5].

An overview of our construction. Let $G = \{E(k, \cdot), D(k, \cdot)\}_k$ denote a family of pseudo-random permutations (or, secure block ciphers). We arrange the persistent memory in a complete l -ary tree H , for $l \geq 2$. The root of the tree is contained in the erasable memory, the internal nodes correspond to the persistent memory addresses, and the leaves contain the data from the virtual memory. Therefore, we need more persistent memory than virtual memory, mainly for the interior nodes, whose number is that of the virtual locations divided by $l - 1$ (thus, increasing l decreases the memory overhead). At each interior node x , there is an associated key k_x and the list of values ever stored at this node's location. The key associated to a leaf is equal to the content of the corresponding location in the persistent memory. At each physical location x , we store $E(k_{p(x)}, k_x \circ j)$ where x is the j 'th child of its parent $p(x)$. To perform either a read or a write operation on a certain position of the persistent memory, we need to access the content of the corresponding leaf in the tree; therefore we follow its path starting from the root, and decrypting each physical location's contents with its parent's key to get its key. Then, in the case of a read operation, we just return the most recently stored value at that leaf. In the case of a write operation, we first insert the value to be written into the list associated to that leaf. Then, we follow the same path now starting from the leaf, and we pick new keys for each node along the path. For each such node, we decrypt its children's most recently stored value and re-encrypt their keys with the parent's new key. This encryption is then inserted into the list associated to that node. We note that before and while computing the encryption/decryption function, we must transfer to the erasable memory all input values, keys and intermediate values used. However, we can do the above so that only a constant number constant number of keys are in the erasable memory at any one time.

A formal description of our construction. Our formal description considers the most general case of arbitrary distribution over the memory operations (the case of consecutive locations can be directly derived from it and is omitted). In order to make the description cleaner, we assume that the intermediate values in the computation of algorithms E and D which need to be stored into the memory will always be stored into the erasable memory. By $p(x)$ and $c_j(x)$ we denote the parent of node x , and the j -th child of node x , respectively.

The algorithm PREPROCESS

Input: an array $M[1 \dots m]$.

Instructions:

1. Initialize empty arrays $PM[1 \dots p]$ and $EM[1 \dots e]$.
2. Arrange PM as a complete l -ary tree H of height $\lceil \log_l m \rceil$, where $2 \leq l \leq 2^{a-K}$.
3. Denote by l_i the i -th leaf of H and let $k_{l_i} = M[i]$, for $i = 1, \dots, m$.
4. For each node x of H ,
 - if x is not a leaf then uniformly choose a key $k_x \in \{0, 1\}^K$;
 - let j be such that x is the j -th child of node $p(x)$;
 - store $k_x \circ j$ into $EM[2]$ and $k_{p(x)}$ into $EM[3]$ and set $z_x = E(k_{p(x)}, k_x \circ j)$;
 - if x is the root then store k_x into $EM[1]$;
 - else insert z_x into list pm_i ,
 - where i is the location in PM associated with node x ;
5. return: (EM, PM) and halt.

The algorithm UPDATE

Input:

1. an instance ins of an operation $op \in OP = \{\text{READ}, \text{WRITE}\}$;
2. arrays $PM[1 \dots p]$ (storing tree H) and $EM[1 \dots e]$.

Instructions:

1. If ins is not valid then return: \perp and halt.
2. Let q be the path in tree H starting from the root and finishing at node associated with location loc to be read or written.
3. For all nodes x in path q (in descending order),
 - if $p(x)$ is the root of H then set $k_{p(x)} = EM[1]$;
 - else if x is not the root then
 - store z_x into $EM[2]$ and $k_{p(x)}$ into $EM[3]$ and set $y_x = D(k_{p(x)}, z_x)$;
 - let $k_x \in \{0, 1\}^K$ and $j \in \{1, \dots, l\}$ be such that $y_x = k_x \circ j$;
4. If $ins = (\text{READ}; loc; EM, PM; cont)$ then
 - set $cont = k_x$, return: $cont$ and halt.
5. If $ins = (\text{WRITE}; loc; EM, PM, cont; EM, PM)$ then
 - for all nodes x in path q (in ascending order),
 - if x is not a leaf then
 - uniformly choose $k'_x \in \{0, 1\}^K$;
 - for $j = 1, \dots, l$,
 - store $z_{c_j(x)}$ into $EM[2]$ and k_x into $EM[3]$ and set $y_{c_j(x)} = D(k_x, z_{c_j(x)})$;
 - let $k_{c_j(x)} \in \{0, 1\}^K$ and $j \in \{1, \dots, l\}$ be s.t. $y_{c_j(x)} = k_{c_j(x)} \circ j$;
 - set $z_{c_j(x)} = E(k'_x, y_{c_j(x)})$;

```

if  $x$  is the root then store  $k_x$  into  $EM[1]$ ;
  else insert  $z_x$  into list  $pm_i$ ,
    where  $i$  is the location in  $PM$  associated with node  $x$ ;
  set  $k_x = k'_x$ ;
return:  $(EM, PM)$  and halt.

```

4 Concrete realizations of small erasable memory

We describe different concrete ways of constructing physically erasable memory. All of these ways seem prohibitive for large amounts of memory, but much more reasonable for the small amount such as what we require. Discussions about comparing and combining the above methods can be found in [5].

Trusted erasable memory. The simplest example of an implementation for an erasable memory is to use a *trusted* or *guardable* piece of memory. In this solution, the erasable memory is not so much a construction as a trust relationship. The system trusts the erasable memory *not to reveal any information*, which is at least as strong as *forgetting* former information. In a network system, the erasable memory might be a separate computer in a locked room. Due to the better physical security of this machine, it is less likely to be corrupted, and thus more likely to forget, in the sense that the information is not revealed to other parties. In the example of a multi-user personal computer (e.g., a device used by many people *one at a time*), each user might maintain physical control of one piece of memory, say a smart card or disk, and use that piece of memory as their personal erasable memory.

Disposable erasable memory. Another construction for erasable memory is based on *limited physical destruction*. More specifically, we can associate the *physical* erasable memory with a disposable memory device, such as an inexpensive smart card or disk. Each time we need to forget a value, a new device is obtained, and the old one physically destroyed, e.g., burnt or melted. (This frequency can be limited by a combination of methods, e.g., that of a *temporarily* trusted memory that is replaced at regular intervals.)

Randomly updated erasable memory. The main problem with erasing memory in physical sources is that imprints are left if the memory has the same value for a long period. Updating with the same value will not prevent this. However, it is reasonable to assume that updating a memory device frequently with uncorrelated random values will not leave traces. Heuristically, our reasoning is as follows: The memory device has a maximum amount of retrievable data that can be stored on it. If there is no distinction between old and new data in terms of time the data has been stored, or correlations with other data, then it seems reasonable to assume that the newer data will be the easiest to recover. If at most a finite amount of data is possible to recover, and newer data is easier to recover than older data, it follows that sufficiently old data is impossible to recover. We can use this to make a fixed item erasable even if held for a long period of time. We distribute the item m by sharing it over two or more memory devices. At any

time, one device holds a random string r , and the other $m \oplus r$. The value of r is updated regularly, and much more frequently than m , by \oplus 'ing both memory devices with the same random string. (Note that the random strings do not have to be kept secret, so they could probably be generated pseudo-randomly even if the seed becomes known later. The randomness is intended to 'fool' the memory device, not the adversary.) This is reminiscent of the *zero-sharing* method in proactive secret sharing schemes [8, 13, 17].

References

1. B. Aiello, M. Bellare, G. Di Crescenzo, and R. Venkatesan, *Security amplification by composition: the case of doubly-iterated, ideal ciphers*, Proc. of CRYPTO 98.
2. M. Bellare, J. Kilian and P. Rogaway, *The security of cipher block chaining*, Proc. of CRYPTO 94.
3. D. Boneh, and R. Lipton, *A revocable backup system*, Proc. of USENIX 97.
4. J. Bos, *Booting problems with the JEC Computer*, personal communication, 1983.
5. G. Di Crescenzo, N. Ferguson, R. Impagliazzo, and M. Jakobsson, *How to forget a secret*, full version of this paper, available from authors.
6. W. Diffie, P. Van Oorschot, and M. Wiener, *Authentication and authenticated key exchanges*, *Design, Codes and Cryptography*, vol. 2, 1992.
7. T. ElGamal, *A public-key cryptosystem and a signature scheme based on discrete logarithms*, Proc. of CRYPTO 84.
8. Y. Frankel, P. Gemmel, P. MacKenzie, M. Yung, *Proactive RSA*, Proc. of CRYPTO 97.
9. O. Goldreich, S. Goldwasser and S. Micali, *How to construct random functions*, *Journal of the ACM*, Vol. 33, No. 4, 210–217, (1986).
10. O. Goldreich and R. Ostrovsky, *Software protection and simulation by oblivious RAMs*, *Journal of the ACM*, 1996.
11. J. Hastad, R. Impagliazzo, L. Levin, and M. Luby, *Construction of a pseudo-random generator from any one-way function*, *SIAM Journal on Computing*, to appear (previous versions: FOCS 89, and STOC 90).
12. A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, M. Yung, *Proactive public key and signature systems*, Proc. of ACM CCS 97.
13. A. Herzberg, S. Jarecki, H. Krawczyk, M. Yung, *Proactive secret sharing, or how to cope with perpetual leakage*, Proc. of CRYPTO '95.
14. M. Luby and C. Rackoff, *How to construct pseudorandom permutations from pseudorandom functions*, *SIAM Journal on Computing*, Vol. 17, No. 2, April 1988.
15. National Institute for Standards and Technology, *Digital signature standard (DSS)*, Federal Register Vol. 56 (169), Aug 30, 1991.
16. R. Ostrovsky and V. Shoup, *Private information storage*, Proc. of STOC 1997.
17. R. Ostrovsky and M. Yung, *How to withstand mobile virus attacks*, Proc. of PODC 91.
18. C. P. Schnorr, *Efficient signature generation for smart cards*, Proc. CRYPTO 89.
19. B. Yee, D. Tygar, *Secure coprocessors in electronic commerce applications*, Proc. of USENIX 95.
20. B. Yee, *Using secure coprocessors*, Ph.D. Thesis, CMU-CS-94-149, 1994.