Universally Composable Disk Encryption Schemes

Ivan Damgård and Kasper Dupont

Dept. of Computer Science, Aarhus University, BRICS

Abstract. We propose a formalization of the security of transparent harddisk-encryption using the universal composability framework. We point out that several commercially available schemes for transparent hard disk encryption are built on principles that limit security, and we propose schemes for disk encryption with passive and active security, respectively. As for the efficiency of the schemes, security against active attacks can be obtained with a constant factor overhead in space and a logarithmic overhead in time. Finally, we also also sketch an actively secure scheme that provides some amount of security, even if the adversary is given temporary access to the internal state of the encryption device used.

1 Introduction

Many end-user encryption products provide facilities for encrypting specific files on a hard disk. However, this seems to be of practical value only in specific cases. Managing the keys for this is too much trouble for most users, and it is often difficult to make sure that the data in a file really is hidden - even if the file is encrypted, it is easy to forget to encrypt a back-up copy. Therefore a transparent solution that encrypts all data written to a particular (part of a) disk seems like a more useful solution. Indeed, many commercial products offer also this facility.

Such a solution will often be placed close to the physical disk, in the sense that the encryption module would take, for instance, the form of a disk driver receiving read/write requests from the operating system, and translating these into read/write requests to the physical disk. In the following, it will be useful to distinguish between *logical sectors* which are addressed in calls to the encryption module, and *physical sectors* which are addressed when the encryption module calls the physical disk. Thus, when the operating system writes to a logical sector, the encryption module will translate this into writing one or more physical sectors on disk. Logical and physical sectors may have the same size, but we make no assumption on this.

Of course, the encryption module will need some cryptographic key material. We assume that the module is initialized by some user, say, a system administrator and is given the required key(s) in this process. This is done whenever the system boots. We do not assume that the module has any persistent memory, since we want to cover the case where the encryption module consists partly or completely of software. In such a case, any persistent memory would have to reside on disk and could easily be read, tampered with or destroyed by an adversary.

We note, however, that one way to have a limited amount of (non-secret) persistent memory would be if a user remembers some data from one session to another and gives them to the encryption module when it is initialized. See section 2 for a discussion of this option.

When is a disk encryption system secure? To say something meaningful about this of course requires that we specify the capabilities of the adversary. We will assume that he has access to the file system as a legal user, that is, he can send data to the encryption module. This may be the case because the operating system accommodates multiple users and some of these may be corrupt, but even a single user system may allow this. For instance, the adversary could send a mail to an honest user which the system then automatically writes to the disk. Moreover, we assume that he can observe the data that is written to the physical disk - this seems clearly justified: after all, if the adversary could not observe what is on disk, there is no point in encrypting the data. A *passive* adversary can only observe the physical disk, while an *active* adversary can modify the physical content of the disk. An active attack may be possible, for instance, if the system can be booted without starting the encryption module, it may then be possible to read and write directly to the disk.

Even before we define formally what security against a passive or active adversary means, we can discuss intuitively the security of some known systems. Many commercial implementations are based on a 1-1 correspondence between logical and physical sectors and encrypt 1 logical sector into 1 physical sector of the same size. Such a scheme is deterministic in the sense that the ciphertext can only be a function of the input to the encryption module, that is, the data to be encrypted, the sector number and the key. No such system can be secure against the following attack

- The adversary knows that a file being has been created by an honest user. Encrypted data is written to it, ending up is some physical sector(s). The adversary doesn't know the data, but has a good guess at what it is.
- When later the file is deleted, the adversary creates his own file and hopes he is assigned the same (logical) sectors as the deleted file.
- The adversary writes his guessed data to the file and compares what shows up on disk to what he saw earlier. This allows him to verify his guess.

It is intuitively clear that this problem can be solved by introducing randomness in the encryption process, and we show later that this is indeed true. This is not enough, however, to protect against an active attack, such as the following:

- The adversary sees a file being created by some user. Encrypted data is written to it, ending up is some physical sector(s). The adversary doesn't know the data yet, but stores the ciphertext C he sees.
- When later the file is deleted, the adversary creates his own file and hopes he is assigned the same sectors as the deleted file.
- The adversary writes C to the sector(s) holding his file and issues a normal read command to the file system. This will give the adversary the decryption of C.

Such an attack may be possible, even if every logical sector is encrypted using a strong chosen ciphertext attack secure encryption scheme, and this makes it clear that authentication has to be part of an actively secure system. It is also clear that an adversarial user may choose the data he writes to disk as a function of the data he observes on the physical disk, and hence it is the entire interaction between the encryption module and the other entities that must be "secure" in some sense.

We therefore believe that a disk encryption scheme should in fact be seen not just as an encryption algorithm, but in fact as an interactive protocol that is carried out between the file system of the operating system, the encryption module, and the physical hard disk. Here, the file system or the disk, or both may be corrupted by an adversary. Note that, by assuming that the file system may be corrupted, we cover the multiuser case, where some users may be adversarial.

In the following, we therefore use the UC framework of Canetti[1] to model what is going on, and to specify what we want to achieve. We use this methodology to formally define what passive, respectively active security of a disk encryption scheme means. The definitions basically demand that the scheme must realize an ideal disk that for every read/write request tells the adversary which logical sector was accessed but reveals nothing about the data. While it is in principle possible to hide even the sector numbers, this causes solutions to be much less efficient, so we see our definition as a reasonable practical alternative. In the active case, the adversary can in addition cause the the ideal disk to fail to answer or to rewind to an earlier state when the system boots. These extra capabilities for an active adversary are unavoidable: an active adversary could always stop all traffic to/from the disk, or replace the entire disk content by an old state before the system boots. If the encryption module has no persistent memory, there is no way to detect this.

We give constructions of systems satisfying the definitions. In particular, we show that any semantically secure encryption algorithm leads to a passively secure disk encryption scheme. We also give a special variant of CBC based encryption that is designed to save on the usage of random bits in the encryption module. We then show how the passively secure scheme can be combined with a collision intractable hash function to form an actively secure scheme. This scheme has a linear overhead in space and a logarithmic overhead in time.

Finally, we sketch an extended solution that offers some amount of security even if the adversary is given temporary access to the internal state of the encryption module.

2 Related Work

In [5], Halevi and Rogaway propose a so called tweakable encryption mode, which can be used to encrypt disk sectors “in place”, that is, the plain- and ciphertext have equal size and the sector number is used as “the tweak”. This is a particular case of the deterministic systems we discussed above. Although this kind of encryption does achieve some security - in fact, the scheme from [5] does as well as is possible for a deterministic system - it cannot provide passive security according to our definition.

In recent independent work [2], Gjøsteen proposes several security notions for disk encryption. His definitions are not based on the UC framework, but are of the more traditional distinguishability based type. The exact relation between his and our notions is not entirely clear, we do not know whether Gjøsteen’s notions have the composability properties that our definition has automatically, because we inherit them from the UC framework. However, our passive UC security seems to be equivalent to his IND-CPA security. Our active UC security is weaker than his IND-CCA notion, since the latter cannot be achieved without the ability to remember a state between sessions, something we do not assume.

Gjøsteen also proposes constructions achieving his security notions, using techniques somewhat similar to, but different from ours. In particular, his strongest IND-CCA notion is achieved assuming that a short public string is remembered (say, by a human user) from one session to the next. In section 5.2 we briefly discuss how this type of state can also be used to improve the security of our actively secure solution, namely we can prevent the adversary from rewinding the disk to an earlier state. This achieves security similar to Gjøsteen’s IND-CCA security.

In an independent work [3] Kamp proposes, without proofs, a mode which uses some techniques similar to ours.

The general notion of disk encryption is to some extent similar - but incomparable - to the oblivious RAM model of Goldreich and Ostrovsky[4], where a secure processor with limited internal memory tries to use an insecure memory in a secure way. Here the (large) insecure memory would correspond to our disk. However, in [4] the goal is to hide even the actual memory positions being accessed (our logical sector numbers), while we hide only the data. On the other hand, in our scenario, the adversary can adaptively choose data to be written to specific locations, whereas in the oblivious RAM model, the program executed by the secure processor decides on its own where to write data to.

3 Model

3.1 The UC Framework

Briefly, the UC framework, introduced by Canetti[1] defines a real and an ideal process. The real process involves a number of players, interactive Turing machines executing a protocol. Also, there is an adversary A modeling an attack on the protocol. A may corrupt some of the players, either passively (which allows him to see their messages and internal data) in an attempt to learn private information of honest (uncorrupted) players. Or actively, where he also tries to influence the output of the protocol by taking full control over corrupted players. Finally, everything external to the protocol, such as surrounding systems and other protocols is modeled by a machine called the environment Z. The role of Z is to give input to the honest players and receive their outputs, moreover Z may communicate with A at any time during the protocol.

The ideal process involves an ideal functionality F, modelling the functionality that the protocol is designed to realize. The players now do not execute the protocol, but instead send their inputs to F and get their outputs back from F. There is an ideal model adversary S who may corrupt players as A could do it in the real process. In the ideal process, however, there is no protocol executed, so all S can do is to communicate with F on behalf of corrupted players, in particular S does not have access to inputs/outputs of honest players. The environment Z interacts in the same way as before with honest players and the (ideal model) adversary S.

We say that a protocol securely realizes F, if for any adversary A there exists an ideal model adversary S such that no environment Z can distinguish the real from the ideal process. More formally, Z outputs a bit when it is done, and we demand that the probability that 0 is produced is essentially the same in the two processes.

The intuition is, first that in the ideal process uncorrupted players get correct outputs because they get them directly from F, and this forces the protocol to also produce correct results, since otherwise Z could easily distinguish the processes. Secondly, the protocol does not reveal information it shouldn't, since whatever A communicates to Z during the attack can be convincingly simulated without access to input/output of honest players.

The ideal model adversary S is sometimes called a simulator. This is because a typical way to construct an ideal model adversary is to let it run A as a subroutine and try to convincingly simulate A's view of the protocol during the attack. Clearly, this will cause A to communicate with Z in the same way as in the real process.

We note that ideal functionalities can also be used in the real process. They can be called by the players as a part of the protocol and will return results computed according to their specification. This models, for instance, some resource we assume is available, such as secure hardware. In general, functionalities may also send information to the adversary, modeling "tolerable" leakage of information, or they may receive input from the adversary, modeling influence that the adversary is allowed on the way the functionality works.

3.2 Modelling Disk Encryption

In our model of disk encryption, we will have 4 players called File System (FS), Encryption Module (EM), Disk, and User. We first define an ideal functionality IdealDisk(ID) modelling what we are trying to achieve.

Data ID holds a list of N sectors known as the current disk contents. ID also holds a list of historic disk contents each identified by a numeric tag. Initially the current disk contents are all zero, and the historic list contains this state identified by the tag 0.

Write On input $Write, i, data$ from FS, ID updates the current disk contents, adds the updated disk contents to the historic list using the first unused *tag*, and sends $Write, i, tag$ to the adversary.

Read On input *Read*, i from FS, ID sends *Read*, i to the adversary. If OK is returned, ID returns sector i from the current disk contents. If the adversary returns any value different from OK, return *ERROR*.

Boot On input a $Boot$-command from the User, send *Boot* to the adversary. If a *tag* identifying an entry in the list of historic disk contents is returned from the adversary, the current disk contents are updated to be a copy of these historic contents and OK is returned to the User. If any other value is returned by the adversary *ERROR* is returned to the User.

This functionality models a disk that is completely reliable, except that the adversary may force errors to occur, and may return the disk to a state it was in at some previous point in time. It reveals which (logical) sectors are written or read, but gives away no information on the data involved.

We now describe a general scheme for how our 4 players will act in the real life protocol.

FS On input $Write, i, data$, FS will send a command $Write, i, data$ to EM, here *data* is a string to be written to logical sector i, we assume for simplicity that the length always fits the length of a sector. FS will ignore further input until a result is returned from EM, this may be OK or *ERROR*. FS copies the result to its output.

On input *Read*, i, FS sends a command *Read*, i to EM, and will ignore further input until a result is returned from EM. This result may be a string *data*, or *ERROR*. When the result is received, FS copies it to its output.

User The user initially chooses at random a password *pw*, which we think of as a binary string of length k bits, where k is the security parameter. He ignores input until $Initialize$ has been received.

On input $Initialize$, which will be executed only once, the User uses *pw* to compute a string *initdata*, and sends to Disk *Init*, *initdata*. User then sends *Boot*, *pw* to EM.

On input *Boot*, the User will send command *Boot*, *pw* to EM; wait for a response from EM, which may be *accept* or $reject$, and copy the result to its output.

EM The encryption module has an internal state *State* : *Exp*, which initially is set to 0. EM will ignore input until the first Boot command has been received.

On input *Boot*, *pw*, the EM sets *State* = 0, and then interacts with Disk via some *Read* and $Write$ commands, updates *State* : *Exp* and computes a result (*accept*/$reject$) which is returned to User. Setting *State* = 0 models the assumption that EM has no persistent storage that survives after it has been off-line.

On input $Write, i, data$, from FS, EM uses *data*, i and *State* : *Exp* as input to execute a sequence of *Read* and $Write$ commands to $Disk$. It computes a result that may be OK or *ERROR* and sends it to FS. Also, *State* : *Exp* may be updated.

On input *Read*, i from FS, EM uses K, *data*, i and *State* : *Exp* as input to execute a sequence of *Read* and $Write$ commands to $Disk$. A result is computed that may be a string *data* or *ERROR*. The result is sent to FS. Also, *State* : *Exp* may be updated.

The sequences of read/write commands used in the above are defined by the encryption algoritm used, some examples are given in the following. We will assume that EM has access to a source of random coins so it can execute a randomized encryption algorithm. This could always be implemented using a pseudorandom function with a key derived from the password.

Disk Holds an array of D strings, each of size equal to that of a sector. Disk ignores input until $Init, initdata$ is received from the User. It then initializes D with the data specified in $initdata$.

On input $Write, i, data$, Disk sets $D[i] = data$. On input $Read, i$, it returns $D[i]$.

Once we have specified an encryption algorithm Alg for EM, this completely specifies a protocol to be executed by FS, EM and Disk. We call this the protocol induced by Alg.

An *admissible* adversary A is one who corrupts (only) the Disk initially - note that since FS just forwards information in the protocol, it makes no real difference whether FS is corrupted or not. Note also that no security would be possible if the User could be corrupted, since the User is the EM's only source of persistent data that are unknown to the adversary.

The UC framework allows the environment Z to give input to honest players such as FS and see the resulting output. Since Z and adversary A are allowed to communicate at any time, this models the fact that a real life adversary may have some amount of "legal" access to the file system, perhaps as a user, or just by interacting with legal users, e.g, sending an email that is then written to the disk of the attacked system. In real life, the adversary probably doesn't have permission to use the entire system freely, however, our model actually makes the worst case assumption that he can read or write any logical sector through FS.

Definition 1. *An encryption algorithm for EM is said to be* passively secure *if the protocol it induces realizes Ideal Disk securely against admissible passive adversaries, furthermore the ideal model adversary guaranteed by the definition must always return OK to IdealDisk, and must always reply to a Boot command with the most recent tag thus causing no change to the current disk contents in ID.*

An encryption algorithm for EM is said to be actively secure *if the protocol it induces realizes Ideal Disk securely against admissible active adversaries.*

4 Passive Security

A first observation is that no encryption algorithm that is deterministic can be passively secure. Here, deterministic means that the data written to the physical disk resulting from a $Write, i, data$ command are completely determined from $i, data$ and the key K. Note that commercial products typically maintain a 1-1 correspondence between logical and physcial sectors of the same size, such that a particular single physical sector is written when a logical sector is written. Clearly, any such algorithm must be deterministic.

In a real system, the following attack exists against any deterministic algorithm:

- The adversary knows that a file being has been created by an honest user. Encrypted data is written to it, ending up is some physical sector(s). The adversary doesn't know the data, but has a good guess at what it is.
- When later the file is deleted, the adversary creates his own file and hopes he is assigned the same (logical) sectors as the deleted file.
- The adversary writes his guessed data to the file and compares what shows up on disk to what he saw earlier. This allows him to verify his guess.

In our formal model, this translates to the following pair Z, A, where Z can distinguish between real and ideal process, no matter what the simulator does:

- Z selects two distinct strings d_0, d_1. Flips a coin b and gives input $Write, i, d_b$ to FS. A sees some string str being written to Disk as a result, and sends this string to Z.

- Z gives input $Write, i, d_0$ to FS, again A observes str' being written to Disk and sends it to Z. Set bit b' to 0 if $str' = str$ and 1 otherwise.
- Finally, Z outputs 0 if $b = b'$ and 1 otherwise.

Clearly, when interacting with the real protocol and adversary as specified, it will always be the case that $b = b'$ and so Z will always output 0. In the ideal process, Z interacts with S who, however, has no access to the data in the write commands. Hence, the strings produced by S do not depend on these data, and so neither does b'. Consequently $b = b'$ with probability 1/2, and so this environment successfully distinguishes between ideal process and real protocol.

The above shows in particular that we cannot have passive security, unless the encryption algorithm is allowed to expand its input, i.e., produce ciphertext that is longer than the input plaintext. This is not surprising, it is well known that the standard security notion for a passive adversary, namely semantic security, cannot be achieved without expanding the plaintext.

On the other hand, as one might expect, any semantically secure cryptosystem can be used to obtain passively secure disk encryption. We will consider any encryption algorithm for EM of the following form: We will assume that EM holds an encryption key K in its State and that this key is derived from the password it receives initially. Also, the Disk is initialized with all-0 contents. Then, on input $Write, i, data$, EM computes $C = E_K(r, data)$, where $E_K()$ is a semantically secure encryption algorithm, and r is the random bits required for $E_K()$. Then computes

$$f(i) = (i_1, ..., i_t), \quad g(i, C) = (d_1, ..., d_t)$$

, where f, g are fixed functions, d_j is a string to be written to a physical sector and i_j is the index of a physical sector. Finally write d_j to sector i_j on Disk, for $i = 1..t$. We assume that C is easy to reconstruct from $g(i, C), i$. Hence, for a $Read, i$ command, simply read the sectors with indices $i_1, ..i_t$, reconstruct C and decrypt.

Theorem 1. *Any disk encryption scheme as specified above is passively secure.*

Proof. We exhibit an ideal model adversary S that works for any real adversary A: S will run A as a subrutine and will forward data directly back and forth between Z and A. The goal is to simulate convincingly the data that A would see being written and read from Disk in a real process.

Note that in the ideal process, S is activated by IdealDisk every time a *Read* or a $Write$ command is issued from FS. Initially, choose a key K as EM would have done. When receiving *Read*, i from IdealDisk, return OK, compute $f(i) = (i_1, ..., i_t)$ and tell A that *Disk* received a *Read*, i_j command and returned $D[i_j]$, for $j = 1..t$. When receiving $Write, i$ from IdealDisk, return OK and compute $f(i) = (i_1, ..., i_t)$. Choose random string R of the same length as a logical sector, and encrypt it under K to obtain ciphertext C. Compute $g(i, C) = (d_1, ..., d_t)$ and tell A that Disk received commands $Write, i_j, d_j$, for $j = 1..t$. Any *Boot* commands from A are ignored.

It is easy to see that the only difference between what A sees in the ideal and the real process is that random data instead of real data are encrypted in the ideal process. This cannot be detected if the cryptosystem is semantically secure.

More formally, the definition of semantic security says that it is infeasible to distinguish an encryption oracle that on input a message m returns $E_K(m)$ from one that returns $E_K(R)$ where R is randomly chosen of the same length as m. Now, given such an oracle O, and assuming Z could distinguish real from ideal process, we could do the following: run the real process with the only difference that EM each time it would encrypt data, it will instead send the plaintext to O and use what is returned in place of the ciphertext. It is now clear that if O is producing real ciphertext we are simply running the real process, if O is producing random ciphertext, we are doing something equivalent to the ideal process. Therefore Z's assumed ability to distinguish real and ideal process allows us to distinguish the two kinds of oracles, contradicting the semantic security.

Even though any semantically secure algorithm works, there can be significant differences in efficiency between different algorithms. For instance, if one naively uses CBC mode with random IV, this will require EM to generate random bits on the fly, and this may not be realistic. Intuitively it seems we could avoid the use of random bits by simply continuing the CBC encryption where we left off. That is use as IV the last cipher block of what was already in the sector. After all what the adversary sees in this case is just a CBC mode encryption of the concatenation of sector contents over time. However this simplistic approach has a weakness, the adversary knows the new IV before the data being encrypted using this IV is chosen. If the environment chose a sequence of writes to the same sector that each start with the IV, it will result in a recognizable pattern in the encrypted sectors.

We propose the following algorithm to reduce consumption of random bits. This comes at the cost of reading data from disk for every write operation, and hence represents a sort of tradeoff between two resources. Note, however, that in section 5.1 we describe a setting where the cost of the extra read is almost removed.

The initialization is modified to fill the disk with CBC encryptions of zero filled logical sectors. The initialization uses random IVs. When booted EM chooses a single random padding block. When receiving a $Write$ command EM encrypts the padding block concatenated with the cleartext using as IV the last cipher block from the previous encrypted contents of the sector. Suppose this results in a sequence of blocks $IV, C_1, C_2, ...$ Notice that the sequence $C_1, C_2, ..$ is a CBC encryption of the data only, where C_1 plays the role of IV, so this will be written to the physical disk in the same way as in the previous scheme. Reading never required random bits and can simply be done in the same way as in the previous scheme.

Theorem 2. *The padded CBC continuation scheme as specified above is passively secure assuming the block cipher is indistinguishable from a random function.*

Proof. The construction is secure because the following simple simulator will work. Whenever the real world adversary would receive a CBC encryption instead use a random bitstring of the same length. The proof works by considering the following three settings.

- A - The environment communicates with the real world.
- B - The environment communicates with the real world except the cipher has been replaced by a random function.
- C - The environment communicates with the ideal world simulator.

First we realize that an environment which could distinguish A from B would also be able to distinguish the cipher from a random function. Next we prove that B can be distinguished from C only with a negligible advantage.

As long as the random function in setting B never evaluates the same input twice, the outputted encryptions are in fact random bitstrings just as in setting C. Thus it suffices to show, that the probability of evaluating the random function twice on the same input is negligible, and as we go on with the proof it will turn out, that the probability does not depend on any actions performed by adversary or environment.

In setting B randomness is used in three different places. For each of these three places we will show that there are a limited number of random choices which could lead to two identical inputs to the random function.

1. The initialization use random IVs. Since we are encrypting all zero bits, the IV will be given directly to the oracle. Bad choices for the IV are only those bitstrings we have previously given to the oracle.
2. When booting a random padding block is chosen. This padding may later be XORed with any of the IVs from the current disk contents. Each of them may collide with a previously used input, thus the number of bad choices is given by the disk size multiplied by the number of previously used inputs.
3. The oracle will produce random output when given an input distinct from all previous inputs. This output will be XORed with the next block of data before being fed back to the oracle. Except from the case handled above, the data will be fixed before the oracle chose the random output. Thus there is a 1:1 correspondance between bad random choices from this invocation and bad inputs when it is used as IV at some later time. This means the number of bad random choices is again the number of previously used inputs plus the number of sectors on the disk. Here the number of sectors on the disk is added because each have a last cipher block which contains an IV we are already commited to be using and thus must not chose again.

What has been shown is, that the probability of colliding inputs is no worse than a CBC mode encryption of $M + N * (B + 2)$ cipher blocks where M is the number of cipher blocks used by the diskencryption, N is the number of sectors on the disk, and B is the number of boots. Assuming the cipher is chosen to securely encrypt this amount of data, this concludes the proof.

5 Active Security

It is not hard to see that no solution based on semantic security as we described above can be actively secure. In a real system, the following attack exists against any such algorithm:

- The adversary sees a file being created by some user. Encrypted data is written to it, ending up is some physical sector(s). The adversary doesn't know the data yet, but stores the ciphertext C he sees.
- When later the file is deleted, the adversary creates his own file and hopes he is assigned the same sectors as the deleted file.
- The adversary writes C to the sector(s) holding his file and issues a normal read command to the file system. This will give the adversary the decryption of C.

In our formal model, this translates to the following pair Z, A, where Z can distinguish between real and ideal process, no matter what the simulator does:

- Z selects distinct strings d, d', and gives input $Write, i, d$ to FS. A sees some string str being written to Disk as a result, and stores it. Z gives input $Write, i, d'$ to FS (resulting in some string str' being written to $Disk$).
- Z gives input $Read, i$ to FS. When A observes the resulting $Read$ commands issued from EM, he returns data consistent with str being on Disk (instead of str'). Z then learns $\tilde{d}$, the result FS obtains for the read command.
- Z outputs 0 if $\tilde{d} = d$ and 1 otherwise.

In the real process, it will always be the case that $\tilde{d} = d$, so Z outputs 0. In the ideal process, IdealDisk will always return d' from the *Read*-command, so Z outputs 1.

It is clear that what we need is some sort of authentication in order to prevent the adversary from inserting data on disk that he would like to have decrypted. One might be tempted to propose using some encryption mode that provides authentication as well as confidentiality, but this will not work, at least not if each logical sector is individually encrypted. When EM reads the encrypted data for such a sector and the authentication checks out, this only proves that the data was written on disk at some earlier time, not that the entire disk content is valid. Therefore, the above attack will also break any such method.

On the other hand, authenticating the entire disk content for each command may work in theory, but is of course hopeless in practice. We now propose an encryption algorithm that is actively secure and has a constant factor overhead in space and logarithmic size overhead in time.

We propose the following changes to the padded CBC continuation mode which will allow us to achieve active security. We need some extra physical sectors to store the same number of logical sectors as before. The extra sectors will be one header sector and a hash tree spanning multiple sectors. The structure of the hash tree is fixed once the disk size is known, the contents of the hash tree are updated dynamically as data changes.

Each node in the hash tree contains the concatenation of hash values of its children. The lowest level of the hash tree contains the concatenation of hash values of the CBC encryptions. The header sector contains a MAC of the root sector of the hash tree, computed using a key that is initially derived from the password and held inside EM.

EM keeps an error flag in memory indicating if anything wrong was ever returned from the disk.

When receiving a *Boot* command EM reads the header sector with the MAC and the root of the hash tree. If the MAC is valid set the error flag to 0 otherwise set the error flag to 1. The root is stored in memory for use in later *Read* and $Write$ commands.

When receiving a *Read* command EM first looks at the error flag. If the flag is 1 $ERROR$ is returned otherwise the path from the root to the encrypted sector is read verifying all hashes on the way. If all hashes are valid decrypt and return the data as in the passive case. If any hash value is wrong set the error flag to 1 and return $ERROR$.

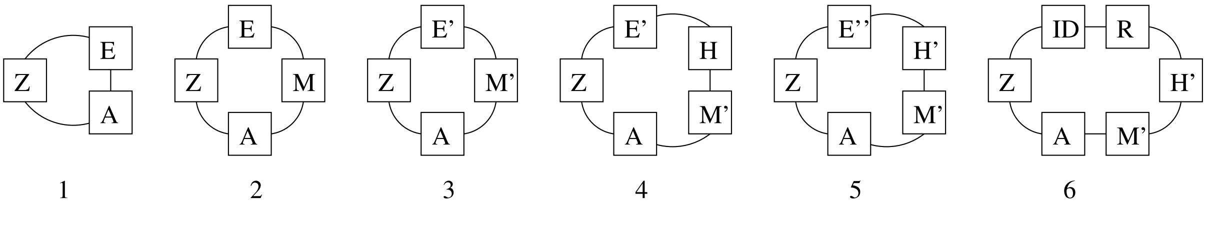
When receiving a $Write$ command EM first looks at the error flag. If the flag is 1 the $Write$ command is ignored. Otherwise all internal nodes on the path from the root to the sector to be written are read into memory. If any hash on the path is invalid set the error flag to 1 and ignore this $Write$ command. Otherwise encrypt the new data, recompute hashes, generate a new MAC on the root of the hash tree, and write all of it to their physical sectors.

Theorem 3. *The disk encryption scheme described above is actively secure*

Proof. Given a real world adversary we can construct an ideal model simulator by keeping track of MAC values and corresponding tags. The simulator essentially behaves as EM but substitutes random bitstrings for CBC mode encryptions.

The *Boot* command is the only tricky part. The simulator must send a *tag* to ID, which is normally not provided by EM. The simulator is able to provide this *tag* by remembering the *tag* on each $Write$ command along with the MAC written to the physical disk.

We see that the environment cannot distinguish the simulator from the real world by looking on this sequence of scenarios.



M is a player which remembers all (root,MAC) pairs send from E to A. As long as both players follows the protocol M will forward all messages unmodified. If A ever attempts to send a (root,MAC) pair which M does not remember M will replace it with a malformed message which will be rejected by E the same way E would reject an invalid MAC.

Z cannot distinguish scenario 1 and 2 because the only case where they would behave differently is if A is able to produce a MAC which is valid under a key known only by E.

E' and M' are similar to E and M except that they no longer exchange MACs but rather tags, and E remembers which hash values each tag corresponds to. When M' needs to send MACs to A, they will be generated using a key with the same distribution as the one used by E.

Scenario 2 and 3 cannot be distinguished by Z. The communication between E' and M' is entirely different, but Z cannot see or modify this communication. Seen from the outside $E' + M'$ behaves exactly the same as $E + M$.

H is a player which remembers the current tag and disk contents corresponding to all tags. Whenever M' reply to a *Read* command H will verify the contents against the remembered value. If it is correct it will be forwarded to E' otherwise H will replace it by a malformed reply, which E' will reject in the same way it would reject an invalid hash. Any other message is simply forwarded unmodified by H.

Z cannot distinguish scenario 3 and 4 because the only case where they would behave differently is if M' send a message to H which produce a hash collision with a message previously sent from M' to H.

E'' and H' are similar to E' and H except H' no longer sends disk contents as reply to *Read* commands but instead simply OK or $ERROR$, and E'' remembers the disk contents itself so it can use the correct disk contents whenever OK is returned by H'. We also reduce the number of $Write$ commands sent by E' such that it no longer sends $Write$ commands with hash tree sectors, H' has all the informations it needs to compute them itself.

Scenario 4 and 5 cannot be distinguished by Z. Like with scenario 2 and 3, Z cannot see the communication between E'' and H' and seen from the outside the two behave exactly like E' and H.

Notice that E'' behaves essentially the same way that ID does. The only difference is that it still leaks the encrypted version of the logical sectors. This can be solved by replacing E'' by ID and R which is a player that simulates encrypted data sectors by using random bitstrings.

The proof from the passive case also applies here and shows that Z cannot distinguish scenario 5 and 6.

This completes the proof. We have shown that an environment cannot distinguish scenario 1 from scenario 6 unless it can either forge a MAC, produce a hash collision, or distinguish the cipher from a random permutation. The four players A, M', H', and R together make up the simulator.

5.1 Implementation

Theorem 4. *Given a passively or actively secure EM we can add a caching, journaling, or restructuring layer below EM and get EM' which is still secure.*

Proof. Any adversary A against the EM with extra layers can be used to construct an adversary A' against EM without those extra layers by simply let A' implement those layers itself and let them communicate with an instance of A.

There are a number of important uses of the extra layers mentioned above.

– CBC mode encryptions will take up more space than a single physical sector, but in reality a lot less than two sectors. By reorganizing data and eliminating the otherwise wasted space a typical implementation could put 32 logical sectors in 33 physical sectors. This change in layout requires reading at least one physical sector to perform a write. By putting the last cipher block of each CBC encryption in the same physical sector a single sector read will suffice and will also contain the information required by the reduced randomness technique described on page 8. Caching this sector may speed up read and write operations.
– Caching parts of the hash tree will give a significant performance improvement for sequential disk access, but no hard numbers can be given as it depends on real world access patterns.
– The reorganization described above, but also the hash tree described earlier requires consistency among multiple sectors. A partial update of the physical media would cause data loss, this means a journaling layer is necessary. The journaling layer needs to know the start and end of each transaction, those can be deduced from the sector numbers of $Write$ commands.

It is important to keep a strict layering. Interactions between layers others than those specified by the EM could introduce security problems. For example having a journaling layer below EM know about transactions on the FS layer could introduce information leaks.

5.2 Improved security with public state information

In [2], it was suggested to use a small public string as a state that is remembered from one session to the next. Here, we briefly discuss how this can be used to improve the security of our actively secure scheme. As state we will use a nonce *str*, where a fresh value is generated periodically, ideally for every write operation. One might use the system time or a counter for this. The idea is that the MAC which is written to disk after every write operation is now computed over the concatenation of the root of the hash tree and the current value of *str*. The last used value of *str* is given to EM at boot time, and as usual EM returns an error if the MAC is not valid. It is trivial to modify the previous proof to show that this will implement a variant of the Ideal Disk that only accepts to use the most recent disk content when it is booted, rather than accepting any previous state.

As we discussed earlier, we do not believe that it is in general realistic to assume that EM could itself remember such a value reliably from one session to the next. But it might in some cases be possible for a human user to remember the value of *str* and give it to the EM at boot time. This assumes, of course, that the user somehow monitors the current value of *str* while the system runs.

6 Ideas for improvements: incorporating Password Updates and Attacks on the EM

So far we have assumed EM was non-corruptible and was given at boot time a password that was fixed initially.

However, it seems natural to allow updates of the password, hoping that this would limit the damage if the current or an old password would leak to the adversary. Another aspect we would like to study what sort of security we can obtain if we allow some amount of corruption of the EM. For both these purposes, we will need to come up with a better encryption mode than what we have seen so far.

In this section, we first describe some basic problems and limitations on what kind of security can be achieved, and finally we propose without proof a mode which we conjecture to be as secure as possible for an EM with no persistent state.

6.1 Security Properties and Limitations

Of course, for any encryption method, given physical disk contents and matching key (or the password from which the key was derived), the adversary can decrypt the disk contents. This means that if a password leaks, the adversary gains full information about any clear text that exists on disk during the life time of that password. This applies also to outdated passwords, since the adversary may have stored the old physical disk content. However, we would like that the adversary learns no other data. Such an attack can be modeled in the UC framework by simply allowing the adversary to ask for the current or an old password. Correspondingly, the simulator is allowed to issue the same command to the ideal disk, which will cause it to reveal all data that the password in question would give access to in the real world.

As mentioned, we would also like to study what can happen if the adversary can attack the EM. Of course, if the adversary can take over the EM completely, there can be no security left. So we ask if there is some limited form of attack that would still allow some security properties to be preserved. This turns out to be the case if the adversary is limited to taking a snapshot of the internal state of the EM. After this, the adversary must leave again and he is not allowed to change anything in the EM. One way such an attack could be performed is if the EM is a software module and the adversary forces a crash so that the memory is dumped (in clear). Having done such an attack, it is clear that the adversary can decrypt what is currently on the physical disk. It is also clear that the adversary can produce arbitrary disk contents which will be accepted after the next boot – simply by simulating the EM. We want that these are the only new powers this attack gives to the adversary. That is, he should not be able to decrypt data written after he took the snapshot, and he should only be able to change disk content at boot time. Again, these possibilities for the adversary can modeled in the UC framework by changing the specification of the ideal disk.

It is easy to see that the security properties we ask for here cannot be achieved unless the EM has a source of random bits. This source can no longer be implemented as we suggested earlier, by a pseudorandom function with a seed known to the EM: this seed would become known to the adversary when he takes a snapshot, he could then predict random coins used in the future inside the EM, hence we could not expect that data written to disk in the future would be hidden. We therefore make the assumption that the system on which we operate includes an uncorruptible unit RO, say a secure part of the CPU, that offers access to something indistinguishable from a random oracle, this could be implemented using a pseudorandom function with a fixed, secret key residing inside RO. Of course, one might object that if we assume such a hardware unit, we might as well put the entire EM with a fixed key inside secure hardware, and not use passwords from the outside. However, putting into secure hardware a simple functionality that will respond to inputs from anyone with pseudorandom answers can be done relatively cheaply – for instance, it could be a hardware implementation of AES with a key hardwired in. On the other hand, putting into secure hardware the entire functionality of an EM with RAM, CPU etc. is much more expensive. We therefore believe this model makes sense in practice. Note that such a unit can offer oracle access to a pseudorandom function as well as "random coins on demand".

Clearly, the actively secure encryption mode we already described is not secure against the new attacks. This already follows from the fact that one fixed key is used throughout. But there is a more fundamental problem with the encryption algorithms we have used so far: the new types of attacks allows the adversary to see ciphertext and then decide if he wants to see the plaintext, e.g., by asking for the current password. This means that in a proof of security, the simulator must produce ciphertext to show the adversary without knowing the cleartext. Then later, if the adversary issues a "reveal password" command, the simulator must come up with a password and derived key such that the ciphertext decrypts under this key to whatever is released by the ideal disk. This problem is well known and comes up in several incarnations in simulation based proofs. The problem can be solved in the random oracle model, as follows: let $CBC_{K,IV}(m)$ be a standard CBC encryption of m with key K and initialization vector IV. Then we will use instead $CBC_{K,IV}(N, H(N) \oplus m)$, where H is the random oracle and N is a nonce chosen freshly (randomly) by EM for every encryption. This allows to simulate encryptions by simply encrypting random data of the correct length, say we set $C = CBC_{K,IV}(R)$. When it becomes known that m should be contained in C, we reveal K, IV, N, m to the adversary and define the value of $H(N)$ to be $R \oplus m$. This output value clearly has the right distribution, and since N is used only once and has been encrypted up this point, the adversary has queried the oracle earlier on N with only negligible probability.

The scheme is essentially the non-committing encryption scheme suggested in [6]. It was pointed out there that in this scheme, it is not possible to instantiate the random oracle with *any* public function and still have the scheme be secure. This sounds like bad news, but in our particular scenario, we can replace calls to the random oracle by calls to the RO-unit we assume is available. Using this encryption scheme throughout makes the disk encryption scheme we suggest below provably UC secure in the model where RO is available, and where the adversary may learn passwords or take snapshots of the internal state of EM. We suspect, however, that the usage of RO is only necessary to make the standard simulation proof technique work, and that in fact the scheme where we do not use RO but use standard encryption is also secure.

6.2 An Improved Disk Encryption Scheme

The improved security is achieved as follows. A number of different encryption keys are introduced, these are put into a tree structure built as follows: Each leaf contains the cleartext of a logical sector, and each internal node contains a randomly generated key. On the physical disk we store, for each node except the root, the content of the node encrypted under the key sitting in the parent node. The key in the root K_{root} is encrypted under the public key of an asymmetric key pair (sk, pk), where the public key pk is stored in memory of the the EM. The private key is stored on disk, encrypted under the password. The authentication using the hash tree and MAC works as before, as described in the previous section.

At boot time, the private key sk is decrypted using the password, and sk is then used to decrypt K_{root} which is kept in memory. Also the key used to generate a MAC on the root of the hash tree is derived from the password. After this, password and sk are erased. When a logical sector is written, new random keys are generated for the path from root to the leaf where the sector is located. The data for the path are reencrypted under the new keys and the ciphertexts are written on disk. Note that K_{root} can also be encrypted because the EM keeps the public key pk. Changing password in this scheme is done by decrypting K_{root}, generating a new MAC on the root of the hash tree and a new asymmetric key pair. Finally, we encrypt the private key using the new password.

Intuitively, the reason why this encryption mode has the security properties we claimed are as follows:

Assume the adversary obtains a password pw, current or old. Then any data that did not exist on disk during the life time of pw was written using a set of keys that are independent from those used when pw was the password. And hence such data cannot be accessed. To see this, note that, although the keys in the tree are not changed when the password is changed, new keys are generated and old keys erased as soon as new data are written.

If the adversary gets a snapshot of the current state, this may be just at boot time in which case the situation is equivalent to leaking the password. Otherwise, the adversary will learn the current K_{root}, but not the private key sk. This means that data written after the snapshot was taken cannot be decrypted: the write operation will update K_{root} and all keys on the path to the data written, and the adversary cannot decrypt any of these keys since he does not know sk. This is the reason for using public-key cryptography here: if K_{root} was encrypted using a symmetric key, the adversary would learn this key from the snapshot, and could continue to decrypt data written later.

Finally, concerning authenticity of data, a password leak or a snapshot will give the adversary the key currently used for generating the MAC on the root of the hash tree. Therefore, at boot time, he can give any root for the hash tree he wants to the EM, together with a valid MAC. This essentially commits the adversary to a complete disk content, ether one that existed earlier or one that the adversary fabricates. However, after booting, the adversary cannot change this contents, since the EM keeps the current hash value in memory.

References

1. R.Canetti: Universally Composable Security: A New Paradigm for Cryptographic Protocols, Eprint archive, no. 2000/67, www.iacr.org.
2. K.Gjøsteen: Security Notions for Disk Encryption, the Eprint archive, no. 2005-88, www.iacr.org.
3. P.H.Kamp: GBDE-GEOM Based Disk Encryption. Proc. of BSDCon 2003: 57-68, www.usenix.org.
4. Oded Goldreich, Rafail Ostrovsky: Software Protection and Simulation on Oblivious RAMs. J. ACM 43(3): 431-473 (1996).
5. S.Halevi and P.Rogaway: A Tweakable Enciphering Mode. Proc. of Crypto 2003: 482-499
6. J.Nielsen: Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case. Proc. of Crypto 2002: pp. 111-126.