Secure Key Management in the Cloud*

Ivan Damgård[1], Thomas P. Jakobsen[1,**],
Jesper Buus Nielsen[1,**], and Jakob I. Pagter[2]

[1] Aarhus University***
{ivan,tpj,jbn}@cs.au.dk
[2] The Alexandra Institute A/S
jakob.i.pagter@alexandra.dk

Abstract. We consider applications involving a number of servers in the cloud that go through a sequence of online periods where the servers communicate, separated by offline periods where the servers are idle. During the offline periods, we assume that the servers need to securely store sensitive information such as cryptographic keys. Applications like this include many cases where secure multiparty computation is outsourced to the cloud, and in particular a number of online auctions and benchmark computations with confidential inputs. We consider *fully autonomous* servers that switch between online and offline periods without communicating with anyone from outside the cloud, and *semi-autonomous* servers that need a limited kind of assistance from outside the cloud when doing the transition. We study the levels of security one can – and cannot – obtain in this model, propose light-weight protocols achieving maximal security, and report on their practical performance.

Keywords: Information security, cloud computing, cloud cryptography, secure key management, secure distributed storage, secure multiparty computation.

* A short version of this paper is to appear at the 14th IMA International Conference on Cryptography and Coding (IMA CC 2013).

** Supported by the Danish Council for Independent Research via DFF Starting Grant 10-081612.

*** The authors acknowledge support from the CFEM research center (supported by the Danish Strategic Research Council), the Danish National Research Foundation, and the National Science Foundation of China (under the grant 61061130540) for the Sino-Danish Center for the Thoery of Interactive Computation, within which part of this work was performed.

1 Introduction

Cloud computing is a disruptive technology, changing the way computing resources are deployed and consumed. The benefits of cloud computing are many, ranging from cost-efficiency to business agility. The main drawback, however, is security and in particular data confidentiality: Users of cloud technology essentially have to trust that the cloud providers do not misuse their data. The recent disclosure of the PRISM surveillance program[3] in which NSA directly monitors all communication going through most world-wide cloud providers such as Yahoo, Google, and Microsoft, is just one out of several incidents emphasizing that this concern about security is real.

In the simple cloud computing case where a user outside the cloud wants to store some data in the cloud for later retrieval, data confidentiality and integrity can relatively easy be ensured. This is typically done using standard cryptography, by encrypting the user's data before it is stored in the cloud, keeping the encryption key secret from the cloud provider. Several products such as CrashPlan[4] and CloudFogger[5] already offer this kind of security.

But the cloud is more than just a storage medium: In particular, computation itself is often outsourced to the cloud. In some cases the computation outsourced is even distributed among several cloud servers and may involve data from many clients. Sometimes the cloud servers may even be controlled by different organizations. Also, the cloud servers may exist in different parts of the cloud, spread across different cloud providers such as Microsoft, Amazon, etc.

An example of this is the Danish site `energiauktion.dk` where electricity for companies is traded at daily online auction. This works by each day starting up a number of auction computations in the cloud. In order to protect the confidentiality of the submitted bids, even against collusions involving the operator of the auction site itself, the bids are encrypted at the clients (the companies), and the auction computations are done using MPC where each MPC server is running in the cloud, controlled by its own organization. Another relevant example is that of the Danish sugarbeet auctions [7]. Here, similar auctions take place, but running on a yearly basis and computing the optimal way to trade Danish sugarbeet contracts instead of electricity. As for `energiauktion.dk`, the confidentiality of bids for the sugarbeet auctions are also ensured via MPC.[6]

Strong notions of security in such more involved cloud applications are generally not as easily obtained as in the simpler case of cloud storage. Promising technologies such as fully homomorphic encryption (FHE) [26] and secure multiparty computation (MPC) [50, 51, 27, 4, 14] definitely have the potential to raise the security for these applications. But despite recent advances [39, 16, 29] they are still quite demanding in terms of performance. While the functions to compute securely in the above examples are simple enough to allow for MPC, securing applications via MPC or FHE in general would still be too resource demanding. More light-weight solutions are therefore needed.

This paper is a study of certain subsets of these cloud computing scenarios. In Section 2 through Section 5 we define these subsets, analyze which levels of security can be obtained and provide concrete protocols achieving this security. In Section 6 we report on

[3] `http://en.wikipedia.org/wiki/PRISM_(surveillance_program)`
[4] `www.crashplan.com`
[5] `www.cloudfogger.com`
[6] While the electricity auctions currently take place in the cloud, the sugarbeet auction servers have until now been running on non-cloud computers, with the MPC keys stored on USB sticks held by administrators of each organization. But the application would fit nicely to a cloud setup.

a prototype implementation of the proposed protocols. For lack of space, the presentation in these sections has been kept at an informal level. A formal model of the protocols and proof of their security within the UC framework [9] are provided in Appendix B.

2 The Model

In this paper we are interested in the following scenario:

- **Distributed computation** A number of n servers $\mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n$ are engaged in some distributed computation taking place in the cloud.
- **Online/offline periods** The computation does not proceed in a continuous fashion. Rather, in some periods there is no need for computation and the servers are therefore idle. We call the first periods for online periods and the latter for offline periods. In this way the application goes through a number of rounds, each round consisting of an online period followed by an offline period, and we assume that the servers receive signals from the application that allows them to agree on the times to switch between online and offline.
- **Sensitive state** During the application's lifetime some or all of the servers possess sensitive data that is needed by these servers in the online periods and that must be stored securely during the offline periods. This could for example be data used in the computation itself or keys needed to authenticate against other servers. We will refer to the sensitive data that S_i must store securely during the offline phase of round r as that server's secret file and we denote it by σ_i^r.

This model of course does not cover all kinds of cloud computing. Regarding the online/offline property we note, however, that many applications naturally only require computation at certain well-defined points in time. For example, online auctions and benchmarks are often designed to be repeated at regular time intervals. Furthermore, most cloud providers operate on a pay-per-use basis (pay per CPU cycle spent, pay per byte sent, etc.) that in itself motivates the design of applications in which computation is "batched" together in time as much as possible such that the cloud servers can be shut down in between these periods of computation in order to save money.

Examples fitting particularly well into our model include those where MPC is done in the cloud. Using MPC is for instance relevant in order to let a client securely outsource a computation to the cloud: By computing via MPC and by making sure that servers are hosted at different cloud providers, strong security is guaranteed since a large number of different cloud providers must collaborate maliciously in order to violate security. Other examples involving MPC in the cloud are the electricity auctions and the Danish sugarbeet auctions mentioned above. Both of these applications involve the regular running of auctions with bids containing confidential client information, and to guarantee confidentiality of the bids, the auction computations are done using MPC where the servers are controlled by different organizations. These applications therefore consist of a distributed system of servers going through a number of online and offline periods (daily offline periods for the electricity auctions, yearly offline periods in the sugarbeet case). During the offline periods, the servers need to store secret data, namely the keys used for doing MPC.

In this paper we do not aim at providing any extra security in the online periods, other than what can already be obtained by other means such as MPC. Rather, we are concerned with the question: *To which extend is it possible to guarantee the confidentiality, integrity, and availability of the servers' secret files in the offline periods, given various*

attacks on the servers and the network over which they communicate. By confidentiality and integrity we mean that a secret file stored by a server at shutdown during an offline period is guaranteed not to be read by others and that the server can be assured that it reads the same unmodified file at wakeup as it stored at the previous shutdown. By availability we refer to the guarantee that a file stored by a server can later be retrieved again.

In the first part of our paper we take into account the following additional requirement:

– **Autonomous servers** The transition between online and offline periods must proceed without anyone from outside the cloud taking action. That is, the servers must be able to switch between online and offline periods communicating only with each other.

This may be essential to some applications. In particular, there simply may not be any relevant parties outside the cloud, such as system administrators or non-cloud servers within an organization, with the right levels of trust, at the times when the cloud servers shut down or wake up.

It turns out that within this model, where the only players are the servers themselves, there is a limit to the level of availability and confidentiality we can hope to get at the same time: Any protocol that guarantees that some subset of the servers can restore a secret file at wake up (availability) of course also allows the same subset of servers to learn the contents of this file, meaning that the file leaks if the servers in that subset are malicious (lack of privacy). Let t_{conf} be the confidentiality threshold, that is, the maximal number of servers an adversary can corrupt without learning anything about the contents of the secret file. Similarly, let t_{avail} be the availability threshold, meaning the maximal number of servers the adversary can corrupt without being able to prevent the reconstruction of the secret file. We can then express this trade-off as follows.

Fact 1 *(Informal) With autonomous servers, the thresholds t_{conf} and t_{avail} must satisfy the equation $t_{conf} + t_{avail} < n$. In particular, no protocol for fully autonomous servers can guarantee at the same time both confidentiality and availability of a secret file in the presence of more than $n/2$ malicious servers.*

In Section 4 we present a protocol for fully autonomous servers that achieves very strong privacy. Because of Fact 1, this means that we have to give up on availability.

The limitation expressed by Fact 1 is a consequence of the requirement that the servers are only allowed to communicate with each other during shutdown and wakeup. We therefore continue our study in the second part of the paper, Section 5, by considering how to most meaningfully relax the requirement of autonomous servers in order to gain more security, while at the same time minimizing the involvement from outside. We end up with the following slightly relaxed requirement:

– **Semi-autonomous servers** Under normal conditions the transition between online and offline periods must proceed without anyone from outside the cloud taking action. However, in case the system has been attacked, we allow the transitions to involve actions from someone from outside the cloud.

We model this more concretely by assuming the existence of certain parties outside the cloud that can fetch data from the cloud servers. For convenience we call these parties administrators, though it could also for example be automated scripts running on trusted (non-cloud) servers within the organizations operating the cloud servers.

The model with semi-autonomous servers covers many applications where the cloud servers are operated by organizations that have their own trusted people or servers elsewhere, outside of the cloud, that can assist the cloud servers in the transitions between online and offline periods. In particular, it models well the scenario where human system administrators are actually willing to log in to the cloud servers in certain situations.

In Section 5 we present a protocol with semi-autonomous servers, providing both very strong confidentiality and availability while at the same time relying only on minimal involvement from the administrators. The protocol essentially guarantees that an administrator can always restore a secret file stored on his server unless *all* the cloud servers have been corrupted.

While definitely suited for increasing security of applications like `energiauktion.dk` and the Danish sugarbeet auctions [7], we believe that these two models capture many other interesting classes of cloud computing applications and that the protocols provided here therefore will be useful for enhancing security for such applications.

3 Related Work

Our work is based on secret sharing. Several secret sharing schemes exists, including schemes allowing various thresholds [5, 45] and schemes with proactive security [30]. However, for the same reason as for Fact 1, secret sharing considered in isolation can never give both availability and confidentiality in case of a dishonest majority of parties. We show how to combine secret sharing with other techniques in a specific context and thereby achieve a level of security that one cannot get with secret sharing alone. In particular, we show how to get both confidentiality and availability in the presence of a dishonest majority in the model with semi-autonomous servers.

Secure multiparty computation (MPC) [50, 51, 27, 14, 4] allows a set of servers to jointly compute on encrypted data. Security, including data confidentiality, is then guaranteed even though some of the servers are malicious and may pool their data together. While still a very resource demanding technology, MPC has reached a level where it has become practical, at least for a limited class of applications [7, 21, 40, 46]. By letting the cloud servers compute using one of the MPC protocols designed to give security against dishonest majority [16, 39] one can achieve a level of security somehow similar to ours in the offline periods, namely that an adversary must break into the offline storage of *all* servers, stealing or modifying *all* MPC keys, to do any harm. However, as mentioned, MPC is still often too heavy and contrary to our protocol for semi-autonomous servers, protocols designed for dishonest majority MPC do not provide strong termination (meaning availability of files in our case).

In any case, for all non-MPC computations as well as computations based on more light-weight MPC protocols that assume less powerful adversaries (that is, honest majority protocols, honest-but-curious or covert adversaries, etc.) our protocols can be used to enhance security during the offline periods at a low cost. In this way, our protocols can be seen not as a substitute, but more as a *complement* to the use of MPC.

Fully homomorphic encryption [26, 25] allows general computations on encrypted data and is in many respects considered the "holy grail" of cloud computing security. FHE allows a user to outsource computation to one or more cloud servers without violating confidentiality even if *all* servers are malicious. Combining FHE with other techniques allows to also guarantee the correctness of the outsourced computation [24, 15, 13]. Furthermore, recent results consider outsourcing computations involving input from several parties [32]. As such FHE can be used to secure essentially any cloud computing scenario,

including those we consider, to a very high degree. Despite recent improvements [8, 29], though, the performance of FHE is still a long way from being efficient enough for practical purposes. Therefore our protocols, perhaps combined with MPC in the online periods, pose a more realistic way to secure cloud computing, at least for the foreseeable future.

Our work essentially consist of protocols for secure, distributed storage of keys and is as such related to the broader field of secure storage and secure distributed file systems. Lots of work has been done in these areas [34], but in many cases such as NFS [38, 47], AFS [31], and SFS [22], security only means that unauthorized clients cannot access or modify files; the storage servers themselves are trusted. Due to increased security demands, a new generation of so-called *cryptographic* storage systems has evolved, exemplified by Windows EFS [20], CFS [6], Tahoe-LAFS [48], NCryptfs [49], and many others. Using various kinds of cryptography, these systems provide the stronger notion of *end-to-end* security, meaning that clients no longer need to trust the storage servers. However, all of these systems require that clients themselves securely store keys and/or remember passwords and are therefore not suitable in our setting where servers must operate autonomously (or at least semi-autonomously) in the cloud.

Some results [23, 2, 36] already consider how data can securely be dispersed among a number of servers *without* the need for storing secret keys on any client. These results combine secret sharing, error correcting codes, variants of Rabin's information dispersal algorithm [41, 35] and other cryptographic techniques in order to guarantee both confidentiality, integrity and availability of the stored data. Forward-secure threshold encryption [37] could also be used to encrypt files at shutdown. As such, these protocols could indeed be used to secure data during the offline phases in our model with autonomous servers.

Common to these results, however, are that they only provide security in the presence of up to $n/2$ malicious servers. In contrast, the constructions provided in this paper are designed to guarantee confidentiality and integrity of the stored data in the presence of up to $n-1$ servers. In Section 4 we achieve this for fully autonomous servers by giving up on availability. In Section 5 we show how to take advantage of the model with semi-autonomous servers in order to also guarantee availability with up to $n-1$ malicious servers.

We are, to the best of our knowledge, the first to consider protocols specifically designed for securing the offline periods in cloud computing environments as described above. In particular, we are not aware of any existing protocols suitable for such cases providing the same combination of high security and good performance as those we present in this paper.

4 Fully Autonomous Servers

We here describe a protocol that increases the offline security in the model with fully autonomous servers and we discuss the limits of the possible security we can in this model. Due to space restrictions the description is kept at an informal level while a formal modelling of the protocol in the UC framework [9] and a rigorous proof of its claimed security are postponed to Appendix B.

Suppose that the overlying application has a security threshold of T_{app}, meaning that an adversary breaking into T_{app} or less servers does not violate security of the overlying application. For many applications $T_{\text{app}} = 0$, but T_{app} may also be higher, say $T_{\text{app}} = n/2$, if for example the overlying application is MPC.

4.1 What Cannot Be Done

As discussed earlier, there is a limit as to how much confidentiality and availability we can achieve at the same time with fully autonomous servers. In addition we observe that it is clearly impossible to protect against the following kind of attack: If the adversary manages to passively break into server S_i during an offline period, he learns whatever that server knows. If he then also attacks S_i's network channels in the following wakeup phase, he can cut off S_i, that is, silence S_i and pretend to be that server towards the remaining servers, using the keys for authentication stolen from S_i during the offline period. By doing this, the adversary has essentially carried out what corresponds to an active (Byzantine) corruption of S_i only by means of a combination of a passive break-in during the offline period followed by a network attack – two attacks normally considered less difficult than a full active attack on the server. We will denote such attacks as cut-off attacks, and not being able to avoid these can be seen as the price we pay for not involving any external parties in the protocol.

4.2 What Can Be Done

We start out with the simplest possible solution and gradually show, in a number of steps informally discussed below, how to extend the solution in order to increase security. The resulting protocol is presented in its entirety at the end of this section.

Secret sharing. In the most naïve protocol each server simply stores its own secret file locally during offline periods. This of course does not add any extra security. In particular, an adversary can spoil security by breaking into the offline storage of $T_{\text{app}} + 1$ servers. The servers could encrypt their secret files before storage, but not much is gained if the encryption key is also stored locally.

We therefore let each server S_i encrypt its file σ_i using a randomly chosen encryption key L_i and then secret share this key among the full set of servers, each server keeping one share $s_{i,i}$ for itself and sending the remaining shares $s_{i,j}$ to each of the other servers S_j before the offline periods. When the servers wake up for the next online period, each server collects its missing shares from the other servers, reconstructs the encryption key, and decrypts its secret file.

With this approach the trade-off between confidentiality and availability discussed earlier can be adjusted by using secret sharing schemes with different thresholds. For example, using Shamir's secret sharing scheme [45] with threshold $t = n/2$ ensures availability of secret files unless $t+1$ servers are malicious, but also only guarantees confidentiality of the files for up to t malicious servers. For now we aim at optimal confidentiality and therefore instead use a sharing scheme with full threshold ($t = n - 1$), such as additive sharings over a finite field. (Better availability is considered later, in Section 5).

This first solution ensures optimal confidentiality of the secret files against adversaries performing only offline attacks, but we also have to consider network attacks. Consider first the case where we just send the shares in cleartext. Here we can observe that S_i never sends its own share $s_{i,i}$ to anyone. Therefore any attack that only uses the network will miss at least one share for every server and so cannot get any useful information. On the other hand, it is also clear that passive eavesdropping combined with an offline attack on S_i will allow you to get σ_i, and so passive eavesdropping plus offline attacks on $T_{\text{app}} + 1$ servers will break the system.

Diffie-Hellmann key exchange. To improve this, we can encrypt the communication. However, securing the communication channels using standard encryption requires servers to store private keys and therefore does not add extra security: This solution can be still broken by offline attacks on $t+1$ servers and passive eavesdropping, because the adversary then knows the keys he needs for decryption.

Instead, we use Diffie-Hellman (DH) key exchange [17] to set up secure pairwise channels on the fly when the systems starts up. DH lets each pair of servers S_i and S_j agree on a secret session key $K_{i,j}$ that can be used to encrypt the channel. This way no private keys for encrypting the channels need to be stored during the offline periods. This means that we are secure against offline attacks on up to $n-1$ servers combined with passive eavesdropping. This is an improvement for any application with $T_{\text{app}} < n-1$.

The above solution does not authenticate the messages in the DH key exchange, since it is only designed to cope with passive eavesdropping. Using an active network attack, an adversary could therefore impersonate any agent during wakeup, and could therefore get the same information as one would get if everything was sent in the clear. However, such an attack alone will not give him any useful information, for the same reason that we described above (S_i never sends its own share $s_{i,i}$ to anyone). As before, if this is combined with an offline attack on T_{app} servers, one gets the online information for these and nothing more. We are therefore still secure against offline attacks on up to T_{app} servers combined with active network attacks. This is optimal because – as discussed in connection with cut-off attacks above – the same attack on $T_{\text{app}}+1$ servers is equivalent to $T_{\text{app}}+1$ full corruptions which is always fatal. In particular, this shows that we do not get any benefit from authenticating the messages in the DH key exchanges.

In conclusion, the solution sketched so far has optimal security against both offline plus passive network attacks as well as offline plus active network attacks, namely security against attacks on $n-1$, respectively T_{app} servers.

Detecting attacks. It turns out that authenticating the DH key exchanges, for instance using digital signatures as in the STS protocol [18], or more generally, using any scheme for authenticated key exchange (AKE), is not useless, however. As discussed earlier, cut-off attacks cannot be prevented. But using AKE, in case a cut-off attack do in fact occur, the cut-off server S itself will always notice that something is wrong, as long as it is not actively corrupted when it wakes up. The reason is the following: Since the adversary broke passively into S during offline it knows S's private AKE key and can thus pretend to be S towards the remaining servers in the online phase. But the real, but impersonated, S will still try to do AKEs with the remaining servers. Unless the adversary passively breaks into *all* the servers, there will be at least one private AKE key that he does not know. This means that S will experience that at least one of the AKEs he tries to complete will fail and can therefore abort the protocol and try to warn the other servers. In other words, the adversary can cut off S, but cannot prevent S from *detecting* the cut-off attack, and unless the adversary can carry out a denial-of-service attack on S forever (something that is often considered practically impossible), this fact will become known to the rest of the system. For these reasons we will use AKE instead of unauthenticated DH in our solution.

Integrity. In the above discussion we have focused on confidentiality. The solution does not, however, protect against for example a corrupted server modifying a share before sending it back to another server at wakeup. We can protect against this by replacing the basic secret sharing scheme with an extended scheme, that we will denote as a robust secret sharing scheme (RSS). Such a scheme produces along with the shares, $s_1, s_2, \ldots, s_n$

a public verification key V. The key V reveals no information about the secret, and can be kept by the server during the offline period and used at wakeup to verify that the shares reconstruct to the original secret. Details on this kind of secret sharing is provided in Appendix A.1.

Proactive security. Proactive security is a powerful notion of security put forward by Canetti *et al.* [10]. In short, a protocol is proactively secure if it can tolerate *any* number of corruptions during its lifetime as long as only a certain number of corruptions take place within a given time frame. Having proactive security is important for protocols such as ours that are supposed to run for a long time.

Our current protocol already is proactively secure in a limited sense: Due to the fact that fresh session keys are generated in each round, we can tolerate any number of *passively* corrupted servers in the offline phases, as long as at most $n-1$ of the corruptions happen in the same offline phase.

There is no proactiveness for the detection of cut-off attacks discussed above, though, since the servers use the same keys for authenticating the DH throughout all rounds. This means that if one manages to steal the private signing key belonging to a server in one round, this key can be used to cut off that server in a later round. We can remedy this by letting the servers in each round refresh the digital signature keys for authenticating the DH. The refreshment is done by letting each server generate a new key pair, replacing its old private signing key with the fresh signing key while sending the new public verification key to the other servers where similar replacements take place. To prevent an attacker from modifying these new public verification keys while they are in transit, each server attaches a message authentication code (mac) to the key, using the current session key that the sending and receiving server share.[7]

With these extra steps we have now obtained a protocol that is proactively secure with respect to passive corruptions and detection cut-off attacks, with each round being one refreshment period. However, obtaining proactive security against *active* offline attacks, that is, where someone not only gets read access to the servers' offline storage, but who can also modify this state during the offline period, turns out to be impossible, at least without any further assumptions. This stems from the fact that once the server gets actively corrupted, during offline as well as online periods, the adversary can change all state, including the protocol code that specifies how the server behaves. By modifying the code offline, the adversary can in effect control the behaviour of the server for the following online period. In this sense, an active attack on a server during the offline period is equivalent to a full active attack on that server during the following online period.

Making the additional assumption that the *code* of each server cannot be changed during offline periods, we can do better. This assumption is a variant of the Read-Only Memory (ROM) model discussed further by Canetti *et al.* [11].[8] In the ROM model, we can strengthen our protocol by letting each server compute a hash of its secret file plus

[7] Our way of securing the network resembles to some extend the way in which SSL/TLS works: SSL/TLS can be configured to use symmetric encryption and macs with a session key established using authenticated Diffie-Hellmann, and also provides a mechanism for renegotiating the keys used to authenticate the DH on a regular basis. We choose however, to embed encryption, etc., directly in our protocol, not relying on SSL/TLS. We do this because we want to be able to reason formally about the security of our protocol which would not be easy with SSL/TLS that consists of over 100 combinations of encryption modes, handshakes, etc.

[8] The assumption can sometimes be justified by the use of ROM or other special hardware such as TPM modules. Also, one can perhaps argue that this models well a cloud environment with all servers booting up from the same uncorrupted virtual image on every wakeup.

some random salt at shutdown and distribute this hash to all servers (including keeping a copy itself). At wakeup we let the server collect again the hashes and abort if these are not all equal. In the ROM model this implies that an adversary will have to actively corrupt the offline storage of *all* servers in the same offline in order to break integrity.

The Protocol. We denote the protocol resulting from this discussion the *Cloud Key Management* protocol, or just $\mathsf{P}_{\mathsf{CKM}}$. It is illustrated in Fig. 1 and consists of two procedures to be carried out by each server, one before entering an offline period (shutdown) and another before returning to the next online period (wakeup). The entire protocol consists of several rounds, each round r consisting of four phases: An online phase where the application is running, a shutdown phase where the servers run the $\mathsf{P}_{\mathsf{CKM}}$ shutdown procedure, an offline phase with no computation, and finally a wakeup phase where the servers run the $\mathsf{P}_{\mathsf{CKM}}$ wakeup procedure to restore the secret files.

When a server S_i receives a file from the environment at shutdown, it is encrypted under a key L using symmetric encryption (Enc). That key is then split into shares $\{s_{i,j}\}_{j\in[n]}$ using a robust secret sharing scheme (RSS). The server keeps one of the shares, $s_{i,i}$ and distributes the remaining shares among the other servers, using a session key for encryption and message authentication codes (macs) to protect against leakage and modification during network transmission. At the end of the shutdown procedure the server erases most values, including the file itself, from its memory. The only values remaining in the following offline phase are the encrypted file, the keys needed for AKEs in the the following wakeup phase, the server's own share and the shares received from the other servers (that follow the same shutdown procedure). On wakeup, a procedure reverse to the shutdown procedure takes place: The server receives its shares from the other servers, reconstructs the key, verifies its integrity, decrypts the file and returns it to the environment. At the beginning of each wakeup and shutdown phase, a server S_i agrees on a fresh secret session key with each of the other servers using AKE. The private and public keys used for AKEs are refreshed once in each round at shutdown.

A few notes about the protocol are in place: The refreshment of the AKE keys is done once every round, but the session key is refreshed twice each round, using the same AKE keys. The reason for doing two session key refreshments is to avoid any shared session key to reside in memory not only during offline, but also during online periods, as doing so would reduce the number of corrupted servers we can tolerate. Also, for simplicity of presentation, the same session key is used for both encryption and macs in Fig. 1. A secure implementation would require separate keys for macs and encryption, see Appendix B.3 for details.

Security. In order to summarize the security of $\mathsf{P}_{\mathsf{CKM}}$ we first define cut-off attacks more precisely as follows:

Definition 1 (Cut-Off Attack). *A cut-off attack on S_i in round r is a passive corruption of S_i in round $r-1$ or r (stealing the server's private AKE key sk_i^{r-1}) combined with active network attack on all of S_i's channels during the shutdown phase of round r (impersonating S_i in the AKEs done there), or a passive corruption of S_i at some point during round r (stealing sk_i^r) combined with active attacks on all of S_i's channels during the wakeup phase of round r (impersonating S_i in the AKEs done in that phase).*

We also note that the security of $\mathsf{P}_{\mathsf{CKM}}$ builds on a number of assumptions:

Shutdown Each server S_i holds from the previous round a private key sk_i^{r-1} and public keys vk_j^{r-1} for each of the other servers S_j. When receiving the secret file σ from the application, S_i does the following.

1. *Session key refreshment*
   (a) For each of the other servers S_j, invoke (in parallel) the AKE protocol, using sk_i^{r-1} and vk_j^{r-1}. This results in S_i and S_j sharing a fresh secret session key $K_{i,j}^{\text{down}}$.
   (b) Generate a new AKE key pair (sk_i^r, vk_i^r).
2. *Encrypt file and distribute shares of the encryption key.*
   (a) Choose a random encryption key L and compute $C \leftarrow \mathsf{Enc}_L(\sigma)$.
   (b) Compute $V, \{s_{i,j}\}_{j\in[n]} \leftarrow \mathsf{RSS}(L)$.
   (c) For each of the other servers S_j, the server S_i compute $c_{i,j} \leftarrow \mathsf{Enc}_{K_{i,j}^{\text{down}}}(s_{i,j})$ and $d_{i,j} \leftarrow \mathsf{Mac}_{K_{i,j}^{\text{down}}}(vk_i^r)$.
   (d) Sends the concatenated message $M_{i,j} = c_{i,j} \,||\, vk_i^r \,||\, d_{i,j}$ to S_j (keeping $s_{i,i}$).
   (e) Wait to receive messages $M_{j,i} = c_{j,i} \,||\, vk_j^r \,||\, d_{j,i}$ from the other servers S_j. Abort if the mac $d_{j,i}$ is invalid, otherwise compute $s_{j,i} \leftarrow \mathsf{Dec}_{K_{i,j}^{\text{down}}}(c_{j,i})$. This step is repeated until valid shares and public keys have been received from all other servers.
3. *Offline state hashing.* Let $\mathcal{O}$ be the concatenation of $(sk_i^r, C, V, s_{i,i})$ with the shares $s_{j,i}$ and public AKE keys vk_j^r received from the other servers S_j. Compute $\gamma_i \leftarrow \mathrm{H}(\mathcal{O})$ and send γ_i to all other servers along with a mac using $K_{i,j}^{\text{down}}$. Wait until valid hash values γ_j have been received from all other servers.
4. Erase all data except $\mathcal{O}$ and the hashes $\{\gamma_j\}_{j\in[n]}$.

Wakeup On wakeup S_i does the following.

1. *Session key refreshment.* Invoke the AKE protocol, this time using sk_i^r and vk_j^r, resulting in S_i and S_j sharing a fresh secret session key $K_{i,j}^{\text{up}}$.
2. *Offline state verification.* Send γ_j to S_j, along with a mac of it using $K_{i,j}^{\text{up}}$. Wait to receive γ_j from the other servers. Verify that all macs are valid and that $\mathrm{H}(\mathcal{O}) = \gamma_j$ for all $j = 1, 2, \ldots, n$, and abort otherwise.
3. *Reestablishing the secret file.*
   (a) Compute $c_{j,i} \leftarrow \mathsf{Enc}_{K_{i,j}^{\text{up}}}(s_{j,i})$ and send $c_{j,i}$ to S_j. Wait until $c_{i,j}$ is received from all other S_j and compute $s_{i,j} \leftarrow \mathsf{Dec}_{K_{i,j}^{\text{up}}}(c_{i,j})$.
   (b) Reconstruct L from $\{s_{i,j}\}_{j\in[n]}$ and verify integrity of the sharing using V. Abort if invalid, otherwise compute and return to the application $\sigma \leftarrow \mathsf{Dec}_L(C)$.
4. Erase all values except sk_i^r and vk_j^r for the other servers S_j.

Fig. 1. The $\mathsf{P}_{\mathsf{CKM}}$ (Cloud Key Management) protocol.

- *Trusted setup* A once-and-for-all setup must be in place, consisting of the initial AKE keys for the first round. This can for instance be established in practice by a PKI.
- *Cryptographic assumptions* Various cryptographic assumptions due to the primitives used in the protocol. For example, the STS protocol for authenticated key exchange [18] builds on the DDH assumption. More details on this can be found in Appendix A.
- *Erasure* That a server can erase part of its state on shutdown such that it is not accessible to an adversary that gets access to the server's offline storage.
- *Randomness* That each server has access to a source of close-to-true randomness.
- *Static adversary* We assume that the adversary decides before each round which servers and channels to corrupt in the following round.
- *Code in Read-Only Memory (ROM)* We assume that at least the code of the protocol itself is stored in ROM and cannot be altered by an active offline attack.

In Appendix B.4 we show how to model these assumptions precisely in the UC framework. The ROM assumption is perhaps the most questionable of these assumptions, so we first summarize what security we have obtained without the ROM assumption.

Theorem 1. *(Informal) Given the assumptions above (except the ROM assumption), the confidentiality and integrity of a file σ_i^r stored and retrieved by S_i in round r using the $\mathsf{P}_{\mathsf{CKM}}$ protocol in Fig. 1 is guaranteed as long as*

1. S_i *has not been actively corrupted (during offline or during online periods) up to and including round* r.
2. S_i *is neither passively corrupted in the shutdown or wakeup phases of round* r.
3. *No cut-off attack on* S_i *takes place up to and including round* r.
4. *No more than* $n-1$ *servers are passively corrupted in each offline phase up to and including round* r.

Furthermore, if σ_i^r *leaked due to* S_i *being exposed to a cut-off attack at any point up to or including round* r*, this will be detected by* S_i.

In particular, without the ROM assumption, a server being actively corrupted at any point, including during offline periods, will stay actively corrupted throughout the protocol. On the other hand, as stated in Theorem 2, the ROM assumption allows us to achieve full proactive security with regards active corruptions.

Theorem 2. *(Informal) Given the assumptions above (including the ROM assumption), the confidentiality and integrity of a file* σ_i^r *stored and retrieved by* S_i *in round* r *using the* $\mathsf{P}_{\mathsf{CKM}}$ *protocol in Fig. 1 is guaranteed as long as*

1. S_i *is neither passively or actively corrupted during the shutdown or wakeup phases of round* r.
2. *No cut-off attack on* S_i *takes place in round* r.
3. *A maximum of* $n-1$ *servers are corrupted, actively or passively, in each round up to and including round* r.

Furthermore, if σ_i^r *leaked due to* S_i *being exposed to a cut-off attack in round* r*, this will be detected by* S_i.

This section has been kept at an informal level, including the two theorems above. For lack of space, more precise definitions of the primitives used, such as the AKE scheme and the robust secret sharing, have been deferred to Appendix A and a formal model of the protocol itself and a proof of its security in the UC framework have been deferred to Appendix B. In particular, formal versions of Theorem 1 and Theorem 2 appear in Appendix B.5 and Appendix B.6.

5 Semi-Autonomous Servers

In the previous section, dealing with fully autonomous servers, we had to choose between guaranteeing either confidentiality or availability in case of dishonest majority as expressed by Fact 1, and we aimed at a protocol with maximal confidentiality. Here we show how to construct a protocol with the same strong confidentiality as before, but with improved availability. This is possible because semi-autonomous servers are allowed to interact with someone from outside the cloud in case of an attack.

As discussed earlier, this is done by providing a special recovery mechanism by which an administrator for a server is guaranteed to be able to recover a file, even if the normal wakeup procedure fails to terminate.

The protocol, which we will denote as $\mathsf{P}_{\mathsf{CKM}}{}^*$, is given below in Fig. 2 and is an extension to the $\mathsf{P}_{\mathsf{CKM}}$ protocol described earlier, that in addition relies on a threshold signature scheme. Such a scheme allows the servers to collectively sign data without any single server being able to sign. In fact, to fit in $\mathsf{P}_{\mathsf{CKM}}{}^*$ the threshold scheme must have full threshold and be proactive. It turns out that the threshold signature scheme of Almansa

et al. [1] is easily modified, giving up on termination, to satisfy our needs. Details are provided in Appendix B.6.

The protocol works as follows: As part of the trusted setup, we also require a threshold signature scheme to have been initialized with the signing key distributed among the servers and such that the administrator and all servers hold the public verification key. In addition we require each administrator to hold a private decryption key for which his server holds the corresponding public encryption key. At shutdown, along with the procedure already specified by $\mathsf{P}_{\mathsf{CKM}}$, the server S_i computes an encryption F_i under the administrator's public encryption key, and the servers then collectively sign F_i. The signature is distributed to all servers, using the session key to authenticate the channels. If normal operation fails during wakeup, the administrator requests the copies of the encrypted file held by the servers. When obtaining one or more of these, he verifies integrity and decrypts the secret file using his private decryption key.

The trusted setup works as in $\mathsf{P}_{\mathsf{CKM}}$, but also includes that the administrator gets a private decryption key dk while each server gets a copy of the corresponding public encryption key ek. Also, the threshold signature scheme is setup, meaning that the administrator and all servers gets the public verification key $\mathcal{W}$ while the shares $\{w_j\}_{j\in[n]}$ of the corresponding signing key is distributed among the servers.

Shutdown As $\mathsf{P}_{\mathsf{CKM}}$, but with the addition that also the proactive refreshment method of the threshold signature scheme is invoked. Also, the following additional steps performed by server S_i the erasing of values in Step 4 of $\mathsf{P}_{\mathsf{CKM}}$:

1. Compute $F_i \leftarrow \mathsf{Enc}_{ek}(\sigma)$.
2. Compute a threshold signature f_i of F_i by invoking $\mathsf{F}_{\mathsf{THSIG}}$.
3. Verify f_i using the public verification key $\mathcal{W}$ and abort if invalid.
4. Place F_i and f_i somewhere that is accessible by the administrator.
5. Send (F_i, f_i) to all other servers.
6. When a pair (F_j, f_j) is received from another server S_j, verify the signature f_j and abort if invalid. Otherwise, make the pair accessible to the administrator and return an OK message to S_j with channel integrity ensured by the session key $K_{i,j}^{\text{down}}$.
7. Abort unless valid signatures have been received from all other servers, and valid OK messages from all servers have been received for the (F_i, f_i) that was sent out from this server.

Wakeup As $\mathsf{P}_{\mathsf{CKM}}$.

File Recovery When the administrator wants to recover the file σ during the wakeup phase he does the following:

1. Fetch messages (F_i, f_i) from the servers (can be done in parallel).
2. When a message (F_i, f_i) is fetched from a server S_j, verify the signature f_i. If valid, output $\sigma \leftarrow \mathsf{Dec}_{dk}(F_i)$. If invalid, fetch a message from another server.

Fig. 2. The protocol $\mathsf{P}_{\mathsf{CKM}}{}^*$ for semi-autonomous servers.

Some additional comments on the protocol: The receipts ensure that if S_i goes offline without aborting, all honest servers have marked the encrypted file of S_i as accessible to the administrator. The operation of "making data accessible" typically involves that this information is stored in a dedicated location on the server's disk, but one could also imagine that on shutdown, this public information is collected, say on a trusted mail server. If normal file recovery by the servers fails, the administrator can, with his verification key, log in to this email server and access the information needed to restore the file.

The security of $\mathsf{P}_{\mathsf{CKM}}{}^*$ is summarized in the following theorem.

Theorem 3 (File Availability). *(Informal) The protocol* $\mathsf{P_{CKM}}^*$ *has the same guarantees as* $\mathsf{P_{CKM}}$ *regarding confidentiality and integrity of stored files. Furthermore, once a server that has not been actively corrupted up to and including round* r*, finishes the shutdown procedure, the file* σ_i *stored at that server is guaranteed to be recoverable by the corresponding administrator, unless all servers are actively corrupted during the following offline and wakeup phase.*

Again, for lack of space, a more precise modelling of the protocol in the UC framework, including the modelling of the administrator, is deferred to Appendix B. The intuitive reason for the strong availability is that because of the threshold signature scheme, the adversary must corrupt *all* servers during the offline period in order to forge the signature or delete all copies of the encrypted file, F_i. If not, the administrator will be able to restore the correct file by fetching F_i from just one honest server, verify the threshold signature, and decrypt it using his private encryption key.

This is a considerably stronger availability guarantee than what was achieved by the $\mathsf{P_{CKM}}$ protocol. We stress that the extended protocol $\mathsf{P_{CKM}}^*$ works in the semi-autonomous model and therefore requires the involvement of administrators, but only if retrieving secret files in the normal, autonomous, way fails due to an attack on the system.

6 A Prototype Implementation

A prototype of the basic $\mathsf{P_{CKM}}$ protocol (without the mechanism for recovery of files by administrators) has been implemented and benchmarked in the Amazon Web Services cloud environment [3]. We here report on these benchmarks and discuss a few practical aspects related to the implementation.

For the benchmarks, each server was running on its own EC2 instance with an Elastic Block Store (EBS) volume as permanent storage. Before each offline period, the shutdown procedure of $\mathsf{P_{CKM}}$ was executed following by disposing each EC2 instance such that during the offline phase only the EBS storage volumes remained. On wakeup, new EC2 instances were started up, the EBS volumes re-associated to the EC2 instances, and the wakeup procedure of $\mathsf{P_{CKM}}$ subsequently executed in order to restore the secret files of the servers.

Table 6 shows the performance of the CKM protocol itself, that is, excluding the 10-30 seconds it typically takes to start up or dispose the EC2 instances. From these results we conclude that the protocol indeed is practical.

Most applications will only require storage of small files such as cryptographic keys. To reflect this, the servers in the benchmark all store and retrieve secret files of size 1 Kb. Storing larger secrets of course increases the execution time, but the size of secrets was found to have relatively little impact: For example, storing 100 Mb instead of 1 Kb secrets roughly costs 2 seconds extra. The reason for this is that that encryption and decryption of secrets take place locally and only the encryption keys are shared.

Also, the results in Table 6 are benchmarks with all servers located in the same Amazon region (with network latency time being roughly 5-10 ms). Other benchmarks have been carried out with servers located worldwide, again with only little impact on the performance: As an example, five servers located across Europe, US, and Singapore were found to decrease performance by roughly 10 percent compared to a single-region setup.

Detecting cut-off attacks. As already discussed, cut-off attacks cannot be prevented, but in case a cut-off attack do in fact occur, the cut-off server S itself will always notice that

	$\mathsf{P}_{\mathsf{CKM}}$ Shutdown	$\mathsf{P}_{\mathsf{CKM}}$ Wakeup
2 servers	5.6 ± 0.5	4.4 ± 1.1
5 servers	9.2 ± 1.2	7.4 ± 0.9
10 servers	16.7 ± 2.8	15.7 ± 1.0
20 servers	33.3 ± 18.8	30.4 ± 18.8

Table 1. Performance of the CKM protocol, $\mathsf{P}_{\mathsf{CKM}}$, in seconds (with 95% confidence intervals). Timings do not include EC2 disposal and start-up times. Each server runs on a small EC2 instance corresponding roughly to 1.7 GiB RAM and a 1.0-1.2 GHz 2007 Xeon processor [19]. Each server stores a 1 Kb file using 1024 bit asymmetric keys and 128 bit symmetric keys.

something is wrong. In order to make this detection as likely as possible in practice, the servers should listen for (authentic) abort messages from the other servers and if such an abort message is received, a server should immediately forward the message to all other servers and to the application. Also, letting the servers wait some time after completing the AKEs, but before sending their shares over the network, will in practice make the task of breaking security by cutting-off servers considerably harder, because the adversary must then silence the cut-off server for at least an amount of time corresponding to this delay before being able to collect shares. Inserting such delays comes, of course, at the price of decreased protocol performance (and are not included in the benchmarks above).

Entropy in the cloud. The servers in the $\mathsf{P}_{\mathsf{CKM}}$ protocol require sources of good randomness in order to generate keys, shares, etc. In Appendix B this is modelled by letting the servers be *probabilistic* Turing machines. In practice, however, this randomness has to come from somewhere. Perhaps the most straightforward solution is to require a random seed to be passed to the $\mathsf{P}_{\mathsf{CKM}}$ protocol from the application and then expand the seed using a secure pseudo-random generator. If done correctly, a polynomial-time adversary will not be able to distinguish the expanded randomness from true randomness if the initial seed is truly random.

However, this just pushes the problem of finding good randomness to the application layer. Another approach is to let the $\mathsf{P}_{\mathsf{CKM}}$ obtain its randomness from the operating system, for example by using the `SecureRandom` Java class which as the default on Linux obtains a random seed from the OS entropy pool though the `\dev\random` interface that blocks until enough entropy has been gathered from the internal clock, network traffic, etc. A somewhat surprising finding from the implementation was that this seriously affects the performance of $\mathsf{P}_{\mathsf{CKM}}$. For example, in the case of five servers, this approach was found to cause a slowdown of 5-10 times for wakeup and 15-20 times for wakeup compared to the benchmark results in Table 6 that use the non-blocking, but potentially less secure, `\dev\urandom` that never blocks, but instead falls back to generating pseudo-random numbers using SHA1 when the OS entropy pool is empty: It takes a considerable time for the entropy pool to acquire enough entropy in newly started virtual instances in the Amazon cloud environment.

Acknowledgements

The authors would like to thank Tim Rasmussen for providing the implementation of the protocol as part of his Master's thesis [43].

References

1. Jesús F. Almansa, Ivan Damgård, and Jesper Buus Nielsen. Simplified threshold RSA with adaptive and proactive security. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 593–611. Springer, 2006.
2. Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi, and Julien P. Stern. Scalable secure storage when half the system is faulty. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 576–587. Springer, 2000.
3. Amazon Web Services cloud computing. http://aws.amazon.com.
4. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 1–10, New York, NY, USA, 1988. ACM.
5. G. R. Blakely. Safeguarding cryptographic keys. *National Computer Conference Proceedings A.F.I.P.S*, 48:313–317, 1979.
6. Matt Blaze. Key management in an encrypting file system. In *Proceedings of the Summer 1994 USENIX Conference*, pages 27–35, 1994.
7. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.
8. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS*, pages 309–325. ACM, 2012.
9. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.
10. Ran Canetti, Rosario Gennaro, and Amir Herzberg. Proactive security: Long-term protection against break-ins. *CryptoBytes*, 3:1–8, 1997.
11. Ran Canetti, Shai Halevi, and Amir Herzberg. Maintaining authenticated communication in the presence of break-ins. *J. Cryptology*, 13(1):61–105, 2000.
12. Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 453–474. Springer, 2001.
13. Ran Canetti, Ben Riva, and Guy N. Rothblum. Refereed delegation of computation. *Inf. Comput.*, 226:16–36, 2013.
14. David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. AC.
15. Kai-Min Chung, Yael Tauman Kalai, and Salil P. Vadhan. Improved delegation of computation using fully homomorphic encryption. In Rabin [42], pages 483–501.
16. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Safavi-Naini and Canetti [44], pages 643–662.
17. Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
18. Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Des. Codes Cryptography*, 2(2):107–125, 1992.
19. Amazon EC2 instance types. http://aws.amazon.com/ec2/instance-types.
20. The Encrypting File System (EFS). http://technet.microsoft.com/en-us/library/cc700811.aspx. A white paper from Microsoft Corporation.
21. Danish Energy Auctions. http://energiauktion.dk.
22. Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Trans. Comput. Syst.*, 20(1):1–24, February 2002.
23. Juan A. Garay, Rosario Gennaro, Charanjit S. Jutla, and Tal Rabin. Secure distributed storage and retrieval. *Theor. Comput. Sci.*, 243(1-2):363–389, 2000.
24. Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Rabin [42], pages 465–482.
25. Craig Gentry. Computing on encrypted data. In Juan A. Garay, Atsuko Miyaji, and Akira Otsuka, editors, *CANS*, volume 5888 of *Lecture Notes in Computer Science*, page 477. Springer, 2009.
26. Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

27. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *STOC*, pages 218–229. ACM, 1987.
28. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *J. Comput. Syst. Sci.*, 28(2):270–299, 1984.
29. HELib, a software library implementing fully homomorphic encryption (copyrighted by IBM). https://github.com/shaih/HElib, 2012.
30. Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In Don Coppersmith, editor, *CRYPTO*, volume 963 of *Lecture Notes in Computer Science*, pages 339–352. Springer, 1995.
31. John H Howard. An overview of the Andrew File System. In *Winter 1988 USENIX Conference Proceedings*, pages 23–26, 1988.
32. Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. *IACR Cryptology ePrint Archive*, 2011:272, 2011.
33. Jonathan Katz, Ueli Maurer, Björn Tackmann, and Vassilis Zikas. Universally composable synchronous computation. In *TCC*, pages 477–498, 2013.
34. Vishal Kher and Yongdae Kim. Securing distributed storage: Challenges, techniques, and systems. In Vijay Atluri, Pierangela Samarati, William Yurcik, Larry Brumbaugh, and Yuanyuan Zhou, editors, *StorageSS*, pages 9–25. ACM, 2005.
35. Hugo Krawczyk. Distributed fingerprints and secure information dispersal. In Jim Anderson and Sam Toueg, editors, *PODC*, pages 207–218. ACM, 1993.
36. Subramanian Lakshmanan, Mustaque Ahamad, and H. Venkateswaran. Responsive security for stored data. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 146–, Washington, DC, USA, 2003. IEEE Computer Society.
37. Benoît Libert and Moti Yung. Adaptively secure forward-secure non-interactive threshold cryptosystems. In Chuankun Wu, Moti Yung, and Dongdai Lin, editors, *Inscrypt*, volume 7537 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2011.
38. The NFS distributed file service. http://users.soe.ucsc.edu/s̃brandt/290S/nfs.ps, 1995. A white paper from SunSoft.
39. Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Safavi-Naini and Canetti [44], pages 681–700.
40. Partisia. http://partisia.com.
41. Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2):335–348, 1989.
42. Tal Rabin, editor. *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings*, volume 6223 of *Lecture Notes in Computer Science*. Springer, 2010.
43. Tim Rasmussen. Key Management in the Cloud. Master's thesis, Aarhus University, Aabogade 34, DK-8200 Aarhus N, Denmark, 2012. Master's Thesis.
44. Reihaneh Safavi-Naini and Ran Canetti, editors. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*. Springer, 2012.
45. Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.
46. ShareMind. http://sharemind.cyber.ee.
47. Brian Pawlowski Spencer, David Noveck, David Robinson, and Robert Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE) 2000*, 2000.
48. Zooko Wilcox-O'Hearn and Brian Warner. Tahoe: The least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, pages 21–26, New York, NY, USA, 2008. ACM.
49. Charles P. Wright, Michael C. Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210. USENIX Association, 2003.
50. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE Computer Society, 1982.
51. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.

A Preliminaries

We take a formal approach where security is reduced to the security of a number of primitives such as encryption, message authentication codes (macs), hash functions, and signatures: We assume computational security of these primitives in the standard computational sense as introduced by Goldwasser and Micali [28]. In particular, we assume the encryption to be IND-CPA secure, and that mac schemes and digital signatures are secure against existential forgery.

A.1 Robust Secret Sharing

We also use a variant of secret sharing that we denote robust secret sharing (RSS): This is basic additive secret sharing with the extra property that the dealer apart from the shares also produces a public key that can later be used to verify the correctness of the shares.[9]

More precisely, a robust secret sharing scheme consists of three PPT Turing machines $(\mathsf{RssGen}, \mathsf{RSS}, \mathsf{Rec})$ parametrized by the number of players n and the security parameter κ. A global key rk is first generated as $rk \leftarrow \mathsf{RssGen}(n, 1^\kappa)$. For a message $m \in \mathcal{M}$, a sharing is then computed as $(V, \boldsymbol{s}) \leftarrow \mathsf{RSS}_{rk}(m)$ where V is the public verification key for the sharing and $\boldsymbol{s} = (s_1, s_2, \ldots, s_n)$ are the shares. The message can then be reconstructed as $m \leftarrow \mathsf{Rec}_{rk}(V, \boldsymbol{s})$.

We capture the privacy requirement of robust secret sharing by the following indistinguishability game. It essentially says that even if revealing both the global key and the public verification key and any $n-1$ shares for a particular sharing to a computationally bounded adversary, the adversary does not learn anything about the plaintext m that was shared.

Definition 2 (IND-RSS). *Let $\mathcal{S} = (\mathsf{RssGen}, \mathsf{RSS}, \mathsf{Rec})$ be a secret sharing scheme as defined above. Let G_b for $b \in \{0, 1\}$ be the PPT TM game that first computes and outputs $rk \leftarrow \mathsf{RssGen}(1^\kappa)$ to the adversary. It then receives $m_0, m_1 \in \mathcal{M}$ and $T \subset \{1, 2, \ldots, n\}$, computes $(V, \boldsymbol{s}) \leftarrow \mathsf{RSS}_{rk}(m_b)$ and outputs V and $\{s_i\}_{i \in T}$. Then $\mathcal{S}$ is said to be IND-RSS if G_0 and G_1 are indistinguishable.*

We will capture the robustness by the following definition. It states in a precise game-based way the requirement that for any $\boldsymbol{s}^* \neq \boldsymbol{s}$, $\perp \leftarrow \mathsf{Rec}_{rk}(V, \boldsymbol{s}^*)$ if $(V, \boldsymbol{s}) \leftarrow \mathsf{RSS}_{rk}(m)$ for some $m \in \mathcal{M}$.

Definition 3 (Robustness). *Let $\mathcal{S} = (\mathsf{RssGen}, \mathsf{RSS}, \mathsf{Rec})$ be a secret sharing scheme as defined above. Let G be the following PPT TM game: It first receives a public key rk from the adversary together with a message $m \in \mathcal{M}$. It then computes and outputs $(V, \boldsymbol{s}) \leftarrow \mathsf{RSS}_{rk}(m)$, receives $\boldsymbol{s}'$ and computes $m' \leftarrow \mathsf{Rec}_{rk}(V, \boldsymbol{s}')$. Finally, if $m = m'$ or $m' = \perp$, it outputs 0 and otherwise it outputs 1. For any possibly unbounded adversary A, let $\mathsf{win}(A, G)$ be the probability that A makes G output 1. Then $\mathcal{S}$ is said to be perfectly robust if $\mathsf{win}(A, G) = 0$.*

We can now formally define what we mean by a robust sharing scheme as follows.

Definition 4. *A robust secret sharing scheme $\mathcal{S}$ is a sharing scheme $(\mathsf{RssGen}, \mathsf{RSS}, \mathsf{Rec})$ that is: 1) correct, meaning that for any $m \in \mathcal{M}$, $m \leftarrow \mathsf{Rec}_{rk}(V, \boldsymbol{s})$ if $(V, \boldsymbol{s}) \leftarrow \mathsf{RSS}_{rk}(m)$, 2) IND-RSS, and 3) perfectly robust.*

[9] This should not be confused with verifiable secret sharing (VSS) where even shares from a malicious dealer are guaranteed to be consistent.

A RSS scheme can for example be constructed from basic additive secret sharing and any commitment scheme with perfect binding, such as a public-key encryption scheme: Given a message m we would first compute its additive secret sharing $\boldsymbol{a} \leftarrow \mathsf{SS}(m)$ and sample uniform values $\boldsymbol{r}$. The sharing would then be $(V, \boldsymbol{s}) \leftarrow \mathsf{RSS}$ where $s_i = (a_i, r_i)$ and $V = (\mathsf{commit}(a_i; r_i))_{i=1}^n$. Recombination would then involve verification of the commitments. Another way of realizing RSS in practice would be to let V be a hash of all shares plus some randomness, utilizing the fact that a hash function is a perfectly binding, computationally hiding commitment scheme, if we assume that it is collision-resistant.

A.2 Authenticated Key Exchange

Finally, we rely on a protocol for authenticated key exchange (AKE): By this we mean a protocol that given a trusted setup allows a number of parties to establish shared secret session keys, despite only being able to communicate over insecure channels. We will use a notion of AKE security closely related to that of *SK-security* defined by Canetti and Krawczyk [12]. However, since we do not need all the security captured by SK-security (which for example includes security against fully adaptive adversaries), we can simplify our proof by sticking to a simpler notion captured by the game in Fig. 3.

Key Generation On input $(\texttt{key-gen}, i)$ for a server S_i, a round r, if this message has not been input before and no sessions have started yet, compute and store (i, vk_i, sk_i) and output vk_i. Mark S_i as honest.

Start Session On input $(\texttt{start-session}, i, j, sid)$, if $\texttt{key-gen}$ has already been invoked for S_i and S_j and (i, j, sid) is not already stored, store (i, j, sid) and start an internal simulation of the AKE protocol between two parties S_i and S_j. (Multiple AKE simulations can go on simultaneously). Every time a message is sent between S_i and S_j it is output, and the adversary gets the chance to modify it or delete it. He also gets the chance to insert extra messages between the two parties.

Corruption On input $(\texttt{corrupt}, i)$, if none of the sessions of S_i are marked as test sessions and all sessions at S_i are completed, mark S_i as corrupt and output sk_i. All sessions of S_i are marked as exposed.

Session Key Query On input $(\texttt{session-key-query}, i, j, sid)$, if all sessions are completed and the session with this sid is not marked as a test session, output the session key $K_{i,j,sid}$ generated by the AKE simulation and mark that session as exposed.

Test Session On input $(\texttt{get-test-session-key}, i, j, sid)$, if this session is completed and unexposed (that is, both S_i and S_j are marked as honest and the key is not marked as exposed), and if at most $T-1$ test sessions have already been marked, do as follows: If $b = 0$, output the key $K_{i,j,sid}$ that was the result of the simulated session and halt. Otherwise, if $b = 1$, output instead a uniformly random key K. Finally, the session (i, j, sid) is marked as a test session.

Fig. 3. The security game $\mathsf{MultiAKE}_b^T$ for security of AKEs.

Definition 5 (IND-AKE). *A protocol π for AKE is secure if for any probabilistic polynomial-time distinguisher A requesting T test sessions and the game $\mathsf{MultiAKE}_\mathsf{b}^\mathsf{T}$ in Fig. 3, the advantage of A when playing the game $\mathsf{MultiAKE}_b^T$ is negligible in κ. We further require that if a session completes at both parties and if both parties have not been corrupted, the keys output at the two parties are identical and uniformly distributed (correctness).*

It is straight-forward to show that SK-security implies security according to the above definition and we therefore leave out the reduction here. Concretely, since basic Diffie-Hellmann key agreement authenticated with digital signatures is known to be SK-secure [12] we can use this scheme to instantiate our protocol.

B Modelling the Protocol in UC

We here provide a formal model of protocols and their security in the UC framework by Canetti [9] and proof that the protocols indeed achieves this security. The appendix assumes that the reader has at least some familiarity with UC security or other kinds of simulation based security based on the real/ideal-world paradigm.

B.1 The UC Framework

The UC framework is an example of the real/ideal world simulation paradigm. This paradigm tries to capture the security of a protocol as follows: First an "ideal world" is considered in which we are allowed to specify the behavior of a special trusted party, called the ideal functionality, that everyone can trust and to which all other parties (servers, in our case) are connected with secure channels. In this world, our desired functionality is easily captured by just letting each server send its secret file to the ideal functionality on shutdown and by receiving it again on wakeup.

The ideal world also consists of an adversary that has the ability to corrupt parties, not including the ideal functionality. When a corruption takes place, the ideal functionality decides exactly what happens. In our case, we can for example let the ideal functionality reveal to the adversary not the content, but only the size of a secret file, once a server is corrupted, and we could specify whether a corrupted server is allowed to prevent another server from restoring its files, etc. In this way, the ideal functionality defines the *ideal leakage* and *ideal influence* of our protocol, that we are willing to accept.

Apart from this ideal world, a "real world" is also specified. In this world there is no trusted party, and the players therefore have to execute the actual protocol in order to store the secret files. In this world there is also an adversary, corrupting players, but when doing so, he gets to fully control these players.

The protocol is then defined to be secure if the two worlds are indistinguishable in a certain well-defined sense. Of course, one could just look to see whether there is an ideal functionality or not, so we instead introduce another Turing machine, denoted the simulator, in the ideal world whose task it is to try to hide this structural difference. In other words, if for any adversary it is so that any distinguisher looking at the trace from the real execution with that adversary and another trace produced by the simulator from the ideal world with the same adversary, that distinguisher cannot tell the difference, then we say that the protocol is secure.

Since this definition implies that in a strong way that no real world adversary can make the real world "deviate from" the ideal world, this makes out a very appealing way to define security. An additional benefit is that this approach allows protocols to be composed from other ideal functionalities that have previously been proved secure, and we call the real world with such ideal sub-functionalities for a hybrid world. It can be proved that the protocol that results from replacing these sub-functionalities with their corresponding protocols yield a protocol that is still secure.

Compared to previous simulation-based definitions, the UC framework yields a considerably stronger notion of security: Instead of presenting the two transcripts of the ideal and real world executions to a distinguisher after the fact, the UC distinguisher, called the environment, is an *interactive* distinguisher. This implies that the simulator has a harder time hiding the differences of the two worlds to the distinguisher (the environment), because it has to make the ideal world look like the real world not only after, but also during, the executions. The reason for this change is that contrary to earlier definitions,

UC secure protocols are also guaranteed to be secure when they execute in a dynamic environment where execution of one protocol is interleaved with the execution of several other instances of the same or other protocols.

Following the notation of [9], we let $\mathsf{EXEC}_{\mathsf{P},\mathsf{A},\mathsf{E}}$ be the view of the environment E in the real protocol execution of protocol P in the presence of the adversary A, and we let $\mathsf{IDEAL}_{\mathsf{F},\mathsf{A},\mathsf{E}}$ be E's view in the ideal protocol execution with ideal functionality F. Then P is said to securely realize F if for every adversary A there exists a simulator S such that for every environment E it holds that $\mathsf{EXEC}_{\mathsf{P},\mathsf{A},\mathsf{E}} = \mathsf{IDEAL}_{\mathsf{F},\mathsf{S},\mathsf{E}}$. For this latter equality we will distinguish between perfect equality ($\stackrel{\mathsf{P}}{=}$), meaning that the distributions are identical, and computational equality ($\stackrel{\mathsf{C}}{=}$), meaning that a polynomially bounded distinguisher cannot tell the difference.

B.2 The Ideal Functionality

Fig. 4 shows the ideal functionality of our protocol, $\mathsf{F}_{\mathsf{CKM}}$, capturing exactly what functionality and security we want it to possess. In other words, if the servers had access to a trusted third party behaving as $\mathsf{F}_{\mathsf{CKM}}$, we would have exactly what we wanted. $\mathsf{F}_{\mathsf{CKM}}$ consists of procedures to be invoked by servers for securely storing a secret file at shutdown and for restoring the file on wakeup. $\mathsf{F}_{\mathsf{CKM}}$ keeps track of the current round r as well as whether a server is in its online or offline phase.

Apart from the ideal functionality, the definition also captures the ideal adversarial leakage and influence, that is, the leakage and damage that we accept when the adversary corrupts a server: Only the size of a stored file is allowed to leak from an honest server. If, however, a server happens to be corrupt during wakeup or shutdown, the file itself is leaked, and if actively corrupted, the adversary also gets to modify the file.[10]

Initially, $\mathsf{F}_{\mathsf{CKM}}$ stores $(\texttt{active}, 0, \mathsf{S}_i)$ for all servers S_i.

Shutdown On input $(\texttt{shutting-down}, r, \sigma_i)$ from server S_i, proceed as follows:

1. If S_i is marked as online corrupted in this round, $(\texttt{corrupted}, r, \mathsf{S}_i, \sigma_i)$ is sent to the adversary A. Note that $\mathsf{F}_{\mathsf{CKM}}$ at this time knows whether this is the case since A is static in each round
2. If no value on the form $(\texttt{active}, r-1, \mathsf{S}_i)$ is stored, then return.
3. Replace $(\texttt{active}, r, \mathsf{S}_i)$ with $(\texttt{shutting-down}, r, \mathsf{S}_i, \sigma_i)$.
4. Send $(\texttt{shutting-down}, r, \mathsf{S}_i, |\sigma_i|)$ to the adversary and if it responds with $(\texttt{shutdown-ok}, r, \mathsf{S}_i)$ replace $(\texttt{shutting-down}, r, \mathsf{S}_i, \sigma_i)$ with $(\texttt{offline}, r, \mathsf{S}_i, \sigma_i)$ and send $(\texttt{offline}, r)$ directly to S_i.

Wakeup On input $(\texttt{waking-up}, r)$ from server S_i, do:

1. If no value on the form $(\texttt{offline}, r, \mathsf{S}_i, \sigma_i)$ is stored, then return.
2. Replace $(\texttt{offline}, r, \mathsf{S}_i, \sigma_i)$ with $(\texttt{waking-up}, r, \mathsf{S}_i, \sigma_i)$.
3. Send $(\texttt{waking-up}, r, \mathsf{S}_i)$ to the adversary and if it responds with $(\texttt{wakeup-ok}, r, \mathsf{S}_i)$, replace $(\texttt{waking-up}, r, \mathsf{S}_i, \sigma_i)$ with $(\texttt{active}, r, \mathsf{S}_i)$ and send $(\texttt{active}, r, \sigma_i)$ to S_i.

Corruption $\mathsf{F}_{\mathsf{CKM}}$ receives corruption messages for a round from A *before* the round starts, and saves them. In any phase where S_i is actively corrupted and in its wakeup or shutdown phase, a message $(\texttt{modify}, \mathsf{S}_i, \sigma')$ from A causes $\mathsf{F}_{\mathsf{CKM}}$ to replace σ_i with σ_i'.

Fig. 4. The ideal functionality $\mathsf{F}_{\mathsf{CKM}}$.

[10] At first sight one would perhaps expect $\mathsf{F}_{\mathsf{CKM}}$ to also leak σ_i if S_i is corrupted during its online phase and let the adversary modify σ_i if actively corrupt during the online phase. The reason for not doing this is that during the online phase, the secret file is returned to the environment and its protection is no longer taken care of by $\mathsf{F}_{\mathsf{CKM}}$: Implementations of $\mathsf{F}_{\mathsf{CKM}}$ simply erase σ_i after returning it to the server just before entering the online phase.

B.3 The Protocol $\mathsf{P_{CKM}}'$

In this section we model the protocol $\mathsf{P_{CKM}}$ itself within the UC framework. We do not include the hashing of the offline state (Shutdown Step 3 and Wakeup Step 2 in Fig. 1) here. Instead, the full version of $\mathsf{P_{CKM}}$, including these steps, is more easily modelled in connection with the extension involving administrators.

We will denote by $\mathsf{P_{CKM}}'$ the $\mathsf{P_{CKM}}$ without the extra step for offline security in the ROM model and it is outlined in Fig. 5. It consists of concrete instructions for the servers, meant to realize $\mathsf{F_{CKM}}$ without access to a trusted party. It is cast in the $\mathsf{F_{SETUP}}'$-hybrid model, meaning that the servers have access to an ideal functionality $\mathsf{F_{SETUP}}'$ that ensures a trusted setup by which the servers can agree on the AKE keys for the initial round. The protocol is essentially a more precise variant of Fig. 1 in Section 4.2, but here the servers are interactive Turing machines communicating with an adversary as defined in UC.

Initialization The server S_i obtains its initial private and public AKE keys, sk_i^0 and vk_j^0, from the trusted setup.

Shutdown When S_i receives $(\texttt{shutting-down}, r, \sigma_i^r)$ from E, it does as follows:

1. If not $(\texttt{active}, r-1)$ is stored, then return. Otherwise, replace $(\texttt{active}, r)$ with $(\texttt{shutting-down}, r)$.
2. *Session key refreshment.* For each of the other servers S_j, S_i invokes (in parallel) the AKE protocol, using sk_i^{r-1} and vk_j^{r-1}, resulting in S_i and S_j sharing a fresh secret session key $K^r_{\text{down},i,j} = (K1^{\text{enc},r}_{i,j}, K1^{\text{mac},r}_{i,j})$.
3. Generate a new AKE key pair $(sk_i^r, vk_i^r) \leftarrow \mathsf{GenAKE}(1^\kappa)$ (where κ is the security parameter) to be used for AKE.
4. *Encrypt file and distribute shares of the encryption key.*
   (a) Generate a random encryption key L_i^r and compute $C_i^r \leftarrow \mathsf{Enc}_{L_i^r}(\sigma_i^r)$.
   (b) Compute $V_i^r, s_{i,1}^r, s_{i,2}^r, \ldots, s_{i,n}^r \leftarrow \mathsf{RSS}(L_i^r)$.
   (c) For each of the other servers S_j compute $c_{i,j}^r \leftarrow \mathsf{Enc}_{K1^{\text{enc},r-1}_{i,j}}(s_{i,j}^r)$ and $d_{i,j}^r \leftarrow \mathsf{Mac}_{K1^{\text{mac},r-1}_{i,j}}(vk_i^r)$.
   (d) Send the concatenated message $M_{i,j}^r = c_{i,j}^r \,||\, vk_i^r \,||\, d_{i,j}^r$ to S_j. (S_i does *not* send $M_{i,i}^r$ to itself via the network, but instead just keeps $s_{i,i}^r$.)
   (e) Wait to receive messages $M_{j,i}^r = c_{j,i}^r \,||\, vk_j^r \,||\, d_{j,i}^r$ from the other servers S_j. When $M_{j,i}^r$ is received, verify the mac $d_{j,i}^r$. If invalid, output $(\texttt{abort})$ to the environment and halt. Otherwise, compute $s_{j,i}^r \leftarrow \mathsf{Dec}_{K1^{\text{enc},r-1}_{i,j}}(c_{j,i}^r)$ and store $(s_{j,i}^r, vk_j^r)$. This step is repeated until valid shares and public keys have been received from all other servers.
5. Erase all the local values including $K1^{\text{enc},r-1}_{i,j}$ and $K1^{\text{mac},r-1}_{i,j}$. The only values not erased are sk_i^r, C_i^r, V_i^r, and $s_{i,i}^r$ as well as the shares $s_{j,i}^r$ and public AKE keys vk_j^r received from the other servers S_j.
6. Replace $(\texttt{shutting-down}, r)$ with $(\texttt{offline}, r)$ and output $(\texttt{offline}, r)$ to the environment.

Wakeup On input $(\texttt{waking-up}, r)$ from E, server S_i proceeds as follows:

1. If not $(\texttt{offline}, r)$ is stored, then return. Otherwise, replace $(\texttt{offline}, r)$ with $(\texttt{waking-up}, r)$.
2. *Session key refreshment.* For each of the other servers S_j, S_i invokes the AKE protocol, this time using sk_i^r and vk_j^r, resulting in S_i and S_j sharing a fresh secret session key $K^r_{\text{up},i,j} = (K2^{\text{enc},r}_{i,j}, K2^{\text{mac},r}_{i,j})$.
3. *Reestablishing the secret file.*
   (a) Compute $c_{j,i}^r \leftarrow \mathsf{Enc}_{K2^{\text{enc},r}_{i,j}}(s_{j,i}^r)$ and send $c_{j,i}^r$ to S_j. Wait until $c_{i,j}^r$ is received from all other S_j and compute $s_{i,j}^r \leftarrow \mathsf{Dec}_{K2^{\text{enc},r}_{i,j}}(c_{i,j}^r)$.
   (b) Reconstruct L_i^r from $s_{i,1}^r, s_{i,2}^r, \ldots, s_{i,n}^r$ and verify integrity using V_i^r. If invalid, output $(\texttt{abort})$ to E and halt, otherwise compute $\sigma_i^r \leftarrow \mathsf{Dec}_{L_i^r}(C_i^r)$.
4. Erase all values except sk_i^r and vk_j^r for the other servers S_j.
5. Replace the message $(\texttt{waking-up}, r)$ with $(\texttt{active}, r, \sigma_i^r)$ and output $(\texttt{active}, r, \sigma_i^r)$ to the environment.

Fig. 5. The protocol $\mathsf{P_{CKM}}'$ (without the ROM assumption).

B.4 Modelling Security Assumptions

The UC model is quite flexible regarding the power and limitations of the adversary and the parties that can be modelled. For example, it allows us to model communication channels with different semantics at the same time. In our case we model pairwise asynchronous and insecure channels between the servers and reliable channels (with eventual delivery) by which an administrator can fetch information from the servers. The ability of servers to erase state is modelled by only requiring corrupted servers to reveal their current state to the adversary. The remaining aspects of our security are modelled by considering a restricted class of adversaries. As a first step, modelling the security of the $\mathsf{P}_{\mathsf{CKM}}{}'$ protocol as outlined in Fig. 5, that is, without the ROM assumption, we restrict this class to adversaries that are

- Polynomially bounded.
- Static in each round, meaning that they always at some point *before* the shutdown in a round is initialized must decide what to corrupt during that round: He must specify exactly which channels and servers to corrupt, whether they are actively or passively corrupted, and in which phases (shutdown, offline, wakeup, online) the corruption happens.
- Can corrupt any number of network channels in each round.
- Can corrupt at most $n - 1$ servers in each round.
- If at any time a server becomes actively corrupted, that server must remain actively corrupted in all future phases and rounds in the protocol.
- No cut-off attack is allowed, that is, if a server is passively corrupted in an offline or online phase in round r then at least one of that server's network channels must not be actively corrupted until all AKEs have successfully completed in the shutdown phase in round $r + 1$.

Note that a server that is passively corrupted in one round might be honest in the next round. But once actively corrupted in a given phase, a server will remain actively corrupted throughout the rest of that round and in all subsequent rounds. Further, the rules guarantee that in any round there will be at least one server that is honest during both the online and the offline phases.

Definition 6. *We denote by $\mathcal{E}_1$ the class of environments in which the adversary follows the rules outlined above.*

What we capture with these rules is essentially proactive security [10] with respect to passive corruptions, but not with respect to active server corruptions: An adversary may "leave" a passively corrupted server, but not a server that has been actively corrupted.

B.5 Security of $\mathsf{P}_{\mathsf{CKM}}{}'$

Having modelled both the protocol and our assumptions about the adversary, we can now formally state what our security involves.

Theorem 4. *Let $\mathsf{P}_{\mathsf{CKM}}{}'$ denote the $\mathsf{F}_{\mathsf{SETUP}}{}'$-hybrid multiparty protocol defined by Fig. 5. Assuming that the encryption, mac, robust secret sharing and AKE primitives are secure as explained in Appendix A, $\mathsf{P}_{\mathsf{CKM}}{}'$ UC-realizes the ideal functionality $\mathsf{F}_{\mathsf{CKM}}$ in Fig. 4 with respect to the environment class $\mathcal{E}_1$. That is, for any PPT adversary A there exists a PPT simulator S with running time polynomial in that of A and such that for any environment $\mathsf{E} \in \mathcal{E}_1$ it holds that*

$$\mathsf{IDEAL}_{\mathsf{F}_{\mathsf{CKM}},\mathsf{S},\mathsf{E}} \stackrel{c}{=} \mathsf{EXEC}_{\mathsf{P}_{\mathsf{CKM}}{}',\mathsf{A},\mathsf{E}} \ . \tag{1}$$

Proof. Let A be a PPT adversary. In order to prove Theorem 4 we must argue that there exists a simulator S with running time polynomial in the running time of A and such that the above equation (1) holds. Recall that a simulator must try to simulate the environment's view of the adversary throughout the protocol execution. That is, E can be seen as an interactive distinguisher that tries to determine whether it executes in the real or the ideal setup. The difficulty arises from the fact that, contrary to the real world adversary A, the simulator has only access to the *ideal* leakage and can only do *ideal* influence as specified by $\mathsf{F}_{\mathsf{CKM}}$.

The overall strategy of our proof is to use the fact that indistinguishability of distribution ensembles is transitive and therefore (1) can be split up into a sequence of intermediary protocol executions, the first one identical to the ideal world execution and the last one identical to the real world execution (from the point of view of the environment). We then show that the environment cannot distinguish between each consecutive pair of these protocol executions. In the first step we argue that the ideal world execution is "structurally" equal to a modified version of the real world execution (again, from the point of view of the environment). In each of the remaining steps we slightly alter the modified real world protocol execution such that it finally equals the real real-world execution. In proving "equality", or rather indistinguishability, between these steps, we typically split each step into even more intermediary protocol executions such that two consecutive protocol executions only differ with respect to a single usage of one primitive such as for example one encryption. By carefully ordering the steps (usually such that keys used for the primitive become independent from anything the adversary sees), this allows us to use the security properties for the primitive to prove that the steps are indistinguishable.

Consider the simulator S in Fig. 6. It simulates internally the real protocol $\mathsf{P}_{\mathsf{CKM}}{}'$ while acting itself as environment. Since the operations of the parties in $\mathsf{P}_{\mathsf{CKM}}{}'$ are PPT, this can be done by S in probabilistic polynomial time in the running time of A. If S had access to the secret files that E inputs to the servers on shutdown, it could forward these inputs to the servers in its internal simulation, resulting in a perfect simulation. However, the simulator only sees the ideal leakage of $\mathsf{F}_{\mathsf{CKM}}$ which – for honest servers – is just the *length* of the secret files. For honest servers we therefore instead let the simulator input "dummy" files consisting of only zero-bits, but of the same length as the real files, to its simulated servers. Our task is then to show that despite of this, E will not be able to decide whether it executes in the real world with A and real parties, or in the ideal world with $\mathsf{F}_{\mathsf{CKM}}$ and S.

As a tool in the proof we first define a special version of $\mathsf{P}_{\mathsf{CKM}}{}'$ parametrized by five bits a, b, c, d, and e. We call the protocol, outlined in Fig. 7, $\mathsf{P}_{\mathsf{CKM}}(a,b,c,d,e)$ and it differs slightly from $\mathsf{P}_{\mathsf{CKM}}{}'$ in five possible ways depending on which of the bits are turned on.

Note that for $d = 1$ and $e = 1$ two servers *magically* agree on certain values (e.g. a new session key in case of $d = 1$). This is not normally possible in protocol executions in the UC model. However, we have the power to let this happen here since the protocol execution is merely a thought experiment used to argue about indistinguishability of protocols. A similar kind of magic allows us to not return the σ_i^r held during offline if $c = 1$ even though S_i gets corrupted.

In the remaining part of the proof, when reasoning about the indistinguishability of distribution ensembles, we will use the shorthand notation $\mathsf{F}_{\mathsf{CKM}} = \mathsf{P}_{\mathsf{CKM}}{}'$ meaning $\mathsf{IDEAL}_{\mathsf{F}_{\mathsf{CKM}},\mathsf{S},\mathsf{E}} = \mathsf{EXEC}_{\mathsf{P}_{\mathsf{CKM}}{}',\mathsf{A},\mathsf{E}}$ and $\mathsf{P}_{\mathsf{CKM}}(a,b,c,d,e) = \mathsf{P}_{\mathsf{CKM}}(a',b',c',d',e')$ meaning $\mathsf{EXEC}_{\mathsf{P}_{\mathsf{CKM}}(a,b,c,d,e),\mathsf{A},\mathsf{E}} = \mathsf{EXEC}_{\mathsf{P}_{\mathsf{CKM}}(a',b',c',d',e'),\mathsf{A},\mathsf{E}}$. From this it follows directly that $\mathsf{P}_{\mathsf{CKM}}(0,0,0,0,0) = \mathsf{P}_{\mathsf{CKM}}{}'$. We first argue that $\mathsf{F}_{\mathsf{CKM}} = \mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0)$ and then that

S simulates internally the real world protocol execution consisting of the adversary A, the servers $\mathsf{S}_1, \ldots, \mathsf{S}_n$, and the ideal functionality $\mathsf{F}_{\mathsf{SETUP}}{}'$. During the simulation, S itself plays the role of the environment. Any input from the actual environment E is forwarded as input to the simulated adversary and output from the simulated adversary is handed back to the E.

Shutdown On receiving $(\texttt{shutting-down}, r, S_i, |\sigma_i^r|)$ from $\mathsf{F}_{\mathsf{CKM}}$ the simulator does as follows: If S_i is going to be passively or actively online corrupted in round r, S also at this point receives a message $(\texttt{corrupted}, \mathsf{S}_i, \sigma_i^r)$ from $\mathsf{F}_{\mathsf{CKM}}$. The simulator uses this to simulate S_i "honestly", that is, it inputs $(\texttt{shutting-down}, r, \sigma_i^r)$ to its simulated server. If, on the other hand, S_i is known to remain honest in the online phases of round r, S uses instead a *dummy* file for simulation, i.e., it sends $(\texttt{shutting-down}, r, 0^{|\sigma_i^r|})$ to the simulated server S_i. In both cases, on output $(\texttt{offline}, r)$ from a simulated server S_i, send $(\texttt{shutdown-ok}, r, \mathsf{S}_i)$ to $\mathsf{F}_{\mathsf{CKM}}$.

Wakeup On receiving $(\texttt{waking-up}, r, S_i)$ from $\mathsf{F}_{\mathsf{CKM}}$, S inputs $(\texttt{waking-up}, r)$ to S_i.
On output $(\texttt{online}, r, \tilde{\sigma}_i^r)$ from S_i, if S_i is actively corrupted at this time, S sends $(\texttt{modify}, \mathsf{S}_i, \tilde{\sigma}_i^r)$ to $\mathsf{F}_{\mathsf{CKM}}$. This ensures that for an actively corrupted server, the environment will always receive the same file on wakeup in the ideal as well as in the real protocol execution. In both cases, S then sends $(\texttt{wakeup-ok}, r, \mathsf{S}_i)$ to $\mathsf{F}_{\mathsf{CKM}}$.

Corruption All corruption messages received from E is forwarded to the simulated adversary A, and S keeps track of the corruptions.
When a server S_i is passively or actively online corrupted in round r, S receives σ_i^r from $\mathsf{F}_{\mathsf{CKM}}$ which it then saves (and uses to simulate S_i "honestly" as described in Shutdown above).

Fig. 6. The simulator S.

$\mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0) = \mathsf{P}_{\mathsf{CKM}}(0,0,0,0,0)$. To prove the latter we go through a number of hybrid arguments.

The overall reasoning for the sequence of hybrids is the following: We must switch off the b-bit before switching off the c-bit. Switching off b can be done via IND-CPA (Step 5) if A cannot influence the L-key (and it is correctly, that is, uniformly, distributed), so we first have to switch on a-bit. This can be done by the IND-RSS (privacy) of the RSS scheme (Step 4) if dummy shares are encrypted and sent over the wires instead of real shares, so we must switch on the e-bit before switching on the a-bit. Switching on the e-bit (Step 3) can be done by IND-CPA if the adversary has no influence on the session keys, so the d-bit must be switched on before the e-bit can be switched on. Switching on the d-bit (Step 2) can be done using a combination of AKE-security and MAC-security.

After switching off the b-bit (Step 5) we would like to switch off the c-bit. But switching off c-bit (Step 7) requires us to first switch off the a-bit in Step 6 (since using random L-key otherwise allows adversary to easily distinguish). After switching off the c-bit, we just need to switch off d and e-bits. For similar reasons as before, we first have to switch off the e-bit (Step 8) before switching off the d-bit (Step 9).

Step 1: $\mathsf{F}_{\mathsf{CKM}} = \mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0)$ Consider first an execution where servers either remain honest or are only corrupt during the offline phases. When a server S_i in $\mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0)$ receives a shutdown request from E its file σ_i^r immediately gets replaced with a dummy file $0^{|\sigma_i^r|}$ that is used throughout the round, except at the end of the wakeup phase where the server returns the original file σ_i^r to E. The only difference between this and the simulated protocol execution in $\mathsf{F}_{\mathsf{CKM}}$ is the initial and final replacement of secret files in $\mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0)$, which the environment cannot see. Since S forwards messages between the simulated adversary and the environment, it follows that from the environments point of view, the two protocol executions are identical, that is, $\mathsf{F}_{\mathsf{CKM}} \overset{\mathsf{P}}{=} \mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0)$.

Consider then what takes place if a server happens to be passively corrupted during one of the online phases of some round r (that is, during either shutdown, wakeup or the application phase): Whether this is going to happen is known to the functionality $\mathsf{F}_{\mathsf{CKM}}$ already when the shutdown phase in round r is initialized because the adversary is static

in each round. The functionality immediately leaks the secret file to the simulator that uses it as input to its corresponding simulated server. So in such a case, the environment also cannot distinguish.

Finally, if a server happens to be actively corrupted in one of the online phases of the round, the real secret file is also leaked from the functionality and the simulator can therefore also use that file as input to the server in its simulation. Furthermore, the final replacement of the file at wakeup does not take place in $\mathsf{P_{CKM}}(0,1,1,0,0)$, so in both protocol executions, the environment will have the same view, also if the adversary chooses to modify the file.

All in all we conclude that $\mathsf{F_{CKM}} \overset{\mathsf{P}}{=} \mathsf{P_{CKM}}(0,1,1,0,0)$.

A server in $\mathsf{P_{CKM}}(a,b,c,d,e)$ behaves exactly as in $\mathsf{P_{CKM}}'$ except:

1. If $a=1$, the original key L_i^r used to encrypt σ_i^r is erased immediately after Step 3a in Shutdown, and a new and uniformly random key $\tilde{L}_i^r$ is secret shared in Shutdown Step 3b instead.
2. If $b=1$, the σ_i^r received from E at Shutdown is immediately replaced by the *dummy* file $0^{|\sigma_i^r|}$ if S_i is known to remain honest throughout the online phases of this round (this is known since the adversary is static within each round). This dummy file is used throughout this round in the protocol whenever the real file was used in $\mathsf{P_{CKM}}'$.
3. If $c=1$, a copy of the original file received at shutdown is made, i.e. $\Sigma_i^r \leftarrow \sigma_i^r$. This copy is stored throughout the protocol, that is, also if $b=1$. In the wakeup phase Σ_i^r is returned to E instead of the decrypted value $\mathsf{Dec}_{L_i^{r-1}}(C_i^{r-1})$ computed in Step 4b of Wakeup unless the server has been actively corrupt at some time during the online phases of this round. Note that the copy Σ_i^r is not part of the internal state of the server and is therefore not handed to the adversary upon corruption.
4. If $d=1$, on both of two servers S_i and S_j, the session keys $K^r_{\mathrm{down},i,j}$ and $K^r_{\mathrm{up},i,j}$ obtained from the AKE sub-protocol in Step 3 of Shutdown and Step 3 of Wakeup are immediately replaced by new and uniformly random session keys $\tilde{K}1^r_{i,j}$ and $\tilde{K}2^r_{i,j}$ which are then used in the rest of the protocol whenever the original $K^r_{\mathrm{down},i,j}$ and $K^r_{\mathrm{down},i,j}$ are used in $\mathsf{P_{CKM}}'$. I.e. both servers end up sharing session keys that are *unrelated* to the keys they jointly generate via the AKEs. This replacement is only done if both S_i and S_j are honest throughout the online phases in round r. If not, the key on neither of the servers gets replaced.
5. If $e=1$, $\tilde{M}^r_{i,j} = \tilde{c}^r_{i,j} \,||\, \mathrm{vk}_i^r \,||\, d^r_{i,j}$ where $\tilde{c}^r_{i,j} = \mathsf{Enc}_{K1^{\mathrm{enc},r}_{i,j}}(r_{i,j})$, where r_i is chosen uniformly at random from the same (finite) field as $s_{i,j}$, is sent from S_i to S_j in Step 4d of Shutdown instead of $c_{i,j} \,||\, \mathrm{vk}_i^r \,||\, d^r_{i,j}$. When S_j receives $\tilde{M}^r_{i,j}$ it is immediately replaced by the correct $M^r_{i,j}$.

Fig. 7. The protocol $\mathsf{P_{CKM}}(a,b,c,d,e)$

Step 2: $\mathsf{P_{CKM}}(0,1,1,0,0) = \mathsf{P_{CKM}}(0,1,1,1,0)$ $\mathsf{P_{CKM}}(0,1,1,1,0)$ differs only from $\mathsf{P_{CKM}}(0,1,1,0,0)$ in that in each round, pairs of honest servers use other, but still uniformly random, session keys than those obtained by the AKE.

Since each honest server in each of its rounds synchronize with all the other honest servers through the AKE we can say that the protocol execution as a whole proceeds in a number of rounds $r = 1, 2, \ldots, R$ (where the total number of rounds, R, depends on A, E, and the random coins used by all Turing-machines in the protocol execution).[11]

Let H^l, for $0 \le l \le R$, be the hybrid protocol execution that proceeds in the same way as $\mathsf{P_{CKM}}(0,1,1,0,0)$ for all rounds $r > l$ and as $\mathsf{P_{CKM}}(0,1,1,1,0)$ for all rounds $r \le l$. Then, by definition, $\mathsf{H}^0 \overset{\mathsf{P}}{=} \mathsf{P_{CKM}}(0,1,1,0,0)$ and $\mathsf{H}^R \overset{\mathsf{P}}{=} \mathsf{P_{CKM}}(0,1,1,1,0)$. Let I^l be as H^{l+1} except that it is only the session keys from the AKEs between honest servers *in the*

[11] Here we assume for simplicity that the execution consists of a whole number of rounds. The proof for the case where servers halt in the middle of a round is an easy modification of the case where servers halt at the end of a round.

shutdown phase of round $l+1$ that are replaced by random keys, that is, I^l is "half-way" between H^l and H^{l+1}. Then, for any $0 \leq l < R$, H^l and H^{l+1} differ only in that all session keys generated by AKEs in round $l+1$ are used in H^l whereas they are immediately replaced by random session keys on both servers in H^{l+1}. Also, H^l differs only from I^l by the AKE keys used in the shutdown phase of round $l+1$ and I^l differs from H^{l+1} only by the AKE keys used in the wakeup phase of round $l+1$.

In the following we will say that the AKE shutdown keys of S_i in round r are consistent if 1) the AKE keys vk_i^{r-1} and sk_i^{r-1} are correctly distributed (that is, they are generated independently of E and according to GenAKE), and 2) at the beginning of the shutdown phase of round r, S_i holds sk_i^{r-1} and all other servers S_j hold vk_i^{r-1}. Similarly, we say that AKE wakeup keys in round r are consistent if the same properties hold for the keys vk_i^r and sk_i^r of S_i at the beginning of the wakeup phase of round r.

Consider the following two statements:

> $P(r)$: $\mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0) \stackrel{\mathsf{c}}{=} \mathsf{H}^r$ *and except with probability negligible in* κ*, the AKE shutdown keys of all honest servers in round* $r+1$ *are consistent.*
>
> $Q(r)$: $\mathsf{H}^r \stackrel{\mathsf{c}}{=} \mathsf{I}^r$ *and except with negligible probability in* κ*, the AKE wakeup keys of all honest servers in round* $r+1$ *are consistent.*

$P(0)$ follows directly from the construction of H^0 and the use of the ideal setup $\mathsf{F}_{\mathsf{SETUP}}$. We now prove that $P(r)$ for some $0 < r < R$ implies $P(r+1)$. Since $P(R)$ implies $\mathsf{H}^R \stackrel{\mathsf{c}}{=} \mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0)$, which by transitivity of indistinguishability implies that $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0) \stackrel{\mathsf{c}}{=} \mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0)$, this completes this step. Now, assume $P(r)$ for some $0 < r < R$. Let G_b be the AKE game from Fig. 3 and consider the following reduction R^{G_b} with oracle access to G_b. R^{G_b} simulates the protocol execution $\mathsf{EXEC}_{\mathsf{H}^r,\mathsf{E},\mathsf{A}}$, outputting whatever the simulated E outputs. However, the following modification are made:

1. In round r, every time an honest server S_i would otherwise generate public and private keys for AKE via GenAKE in the shutdown phase of this execution, R^{G_b} inputs $(\texttt{key-gen}, i)$ to G_b and embeds the public AKE key received from G_b into the protocol as vk_i^r.
2. Every time a new AKE sub-protocol using sk_i^r is initiated, that is, in the wakeup phase of round r and the shutdown phase of round $r+1$, a G_b session is initiated using either $sid = \text{up}$ or $sid = \text{down}$, respectively. Every time a message m is sent between S_i and S_j in the simulated AKE sub-protocol, G_b outputs m to the adversary. The adversary gets to modify or delete m before delivery and he is allowed to insert extra messages.
3. When S_i is corrupted such that E would learn sk_i^r, R^{G_b} sends $(\texttt{corrupt}, i)$ to G_b and receives sk_i^r which is embedded
4. In round r, when the AKE in wakeup completes in H^r, R^{G_b} sends the message $(\texttt{session-key-query}, sid)$ for $sid = (i, j, r, \text{up})$ to G_b and the returned session key is embedded into the protocol as the result of the AKE. The E will not be able to distinguish because of this, since in H^r the resulting session key is already random and independent of E. Embedding keys here reflects the fact that the adversary should have no advantage in guessing the outcome of a particular AKE even though the he learns something about previously generated session keys using the same key, say, through attacks on the network.
5. When an AKE between pairs of honest servers S_i and S_j in the shutdown of round $r+1$ are completed, R^{G_b} calls $\texttt{get-test-session-key}, i, j, \text{down})$ and embeds the test

keys returned in the protocol execution as the keys produced by the servers by this AKE

By construction, $\mathsf{R}^{G_0} \stackrel{\mathsf{P}}{=} \mathsf{H}^r$ and $\mathsf{R}^{G_1} \stackrel{\mathsf{P}}{=} \mathsf{I}^r$. Since $G_0 \stackrel{\mathsf{C}}{=} G_1$ by Definition 5 and indistinguishability is preserved under efficient transformations, we get that $\mathsf{R}^{G_0} \stackrel{\mathsf{C}}{=} \mathsf{R}^{G_1}$. By transitivity of indistinguishability it then follows that $\mathsf{H}^r \stackrel{\mathsf{C}}{=} \mathsf{I}^r$.

We now argue that in the execution I^r the AKE keys vk_i^r and sk_i^r used at the beginning of the wakeup phase in round $r+1$ are consistent. Note first that the secret key sk_i^r is kept by server S_i and are therefore only modified if S_i is actively corrupted. So we only have to argue that the public key vk_i^r is consistent at the wakeup phase in round $r+1$. Let E be the event that there exists two servers S_i and S_j (which are not actively corrupted and do not abort in the wakeup phase of round $r+1$) such that in the wakeup phase of round $r+1$ in the execution $\mathsf{EXEC}_{\mathsf{I}^r,\mathsf{A},\mathsf{E}}$, the key $\tilde{vk}_i^r$ held by S_j does not match the key vk_i^r produced by S_i. We then want to prove that $\mathsf{Pr}[E]$ is negligible in κ. Suppose for the sake of contradiction that $\mathsf{Pr}[E]$ is non-negligible. We can write $E = \bigcup_{i,j} E_{i,j}$ where $E_{i,j}$ is the event that the public key is inconsistent between S_i and S_j in the wakeup of round $r+1$. Since by the union bound $\mathsf{Pr}[E] \leq \sum \mathsf{Pr}[E_{i,j}]$ then there must exist one (i,j) such that $\mathsf{Pr}[E_{i,j}]$ is non-negligible. We can then construct an adversary B for the game G for unforgeable macs as follows: B simulates $\mathsf{EXEC}_{\mathsf{I}^r,\mathsf{A},\mathsf{E}}$ but the message vk_j^r sent to S_i by S_j as well as the key vk_i^r computed locally by S_i are input to G and the resulting macs embedded in the execution as $d_{j,i}$ and $d_{i,j}$, respectively. The altered mac $\tilde{d}_{i,j}$ and the corresponding altered key $\tilde{vk}_i^r$ received by S_j are input to G as the challenge. From the fact that S_j does not abort on the received key and mac, it follows that B wins the game G with the same probability $p = \mathsf{Pr}[E_{i,j}]$ which is non-negligible. This contradicts the security of the mac scheme, so we conclude that all AKE keys among servers that are not actively corrupted in the beginning of the wakeup phase of round $r+1$ are consistent.

Taken together, this implies $Q(r)$. By $Q(r)$ we have that the AKE keys are correctly distributed in the wakeup phase of round $r+1$ except with negligible probability. Therefore, by an argument similar to the argument involving AKE security above, also the session keys resulting from the AKEs in the wakeup phase of round $r+1$ can be replaced by random keys without E being able to distinguish, that is, we get that $\mathsf{I}^r \stackrel{\mathsf{C}}{=} \mathsf{H}^{r+1}$. Unless servers are actively corrupted in round $r+1$, the AKE keys vk_i^r and sk_i^r will also be consistent in the beginning of the shutdown phase of round $r+2$. All in all, this implies $P(r+1)$.

Since $P(r)$ implies $P(r+1)$ and $P(0)$ is true because of the trusted setup, we get $P(R)$, in particular $\mathsf{P}_{\mathsf{CKM}}(0,1,1,0,9) \stackrel{\mathsf{C}}{=} \mathsf{H}^R$. Since by construction $\mathsf{H}^r \stackrel{\mathsf{P}}{=} \mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0)$ we get by transitivity of indistinguishability that $\mathsf{P}_{\mathsf{CKM}}(0,1,1,0,0) \stackrel{\mathsf{C}}{=} \mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0)$.

Step 3: $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0) = \mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1)$ The only difference here is that in both the shutdown and wakeup phases, encrypted dummy shares $\tilde{c}_{i,j} = \mathsf{Enc}_K(r_{i,j})$ are sent over the wires instead of the real encrypted shares $c_{i,j} = \mathsf{Enc}_{K_2}(s_{i,j})$. That is, when A for example corrupts a channel in the shutdown phase, E may see either $\mathsf{Enc}_K(r_{i,j})$ or $\mathsf{Enc}_K(r_{i,j})$. Due to the previous step, E now has no influence on which K is used: It is guaranteed to be sampled uniformly at random and independent of E. We can therefore use the IND-CPA security property of the encryption scheme as detailed in the following.

Let L be the total number of session keys used by servers in $\mathsf{EXEC}_{\mathsf{P}_{\mathsf{CKM}},\mathsf{A},\mathsf{E}}$. These can be ordered according to some ordering $\pi(i,j,r,\cdot) \mapsto \{0,1,\ldots,L-1\}$ such that the l'th session key is used by S_i and S_j in the shutdown phase of round r for $(i,j,r,\text{down}) = \pi^{-1}(l)$ and by these servers in the following wakeup phase if $(i,j,r,\text{up}) = \pi^{-1}(l)$ (note that this

ordering does not necessarily correspond to the order in time in which the keys are used in the execution). Define the hybrid protocols H^l for $0 \leq l \leq L$ as $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0)$ with the following modification: In the shutdown phase of round r, servers S_i and S_j encrypt random shares as in $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1)$ if $\pi(i,j,r,\text{down}) \leq l$, and similarly these servers use random shares in the wakeup phase of round r if $\pi(i,j,r,\text{up}) \leq l$. Then, by construction, $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0) \stackrel{\mathsf{P}}{=} \mathsf{H}^0$ and $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1) \stackrel{\mathsf{P}}{=} \mathsf{H}^L$.

Let $0 \leq l < L$ and consider the reduction $\mathsf{R}^{G_b^2}$ with access to the IND-CPA encryption oracle G_b^2 for two encryptions from the IND-CPA game: $\mathsf{R}^{G_b^2}$ simulates $\mathsf{EXEC}_{\mathsf{H}^l,\mathsf{A},\mathsf{E}}$ with the modification that between the servers S_i and S_j in the round r and phase defined by the index l, neither the real nor dummy shares are encrypted. Instead $\Big((s_{i,j}^r, s_{j,i}^r), (r_{i,j}, r_{j,i})\Big)$ is input to G_b^2 and the returned ciphertext (c_1, c_2) embedded in the simulation such that these values are sent on the wire between S_i and S_j. $\mathsf{R}^{G_b^2}$ finally outputs whatever the simulated environment outputs. By construction we then have $\mathsf{R}^{G_0^2} \stackrel{\mathsf{P}}{=} \mathsf{H}^l$ and $\mathsf{R}^{G_1^2} \stackrel{\mathsf{P}}{=} \mathsf{H}^{l+1}$. Since $G_0^2 \stackrel{\mathsf{C}}{=} G_1^2$ and efficient transformations preserves indistinguishability, $\mathsf{R}^{G_0^2} \stackrel{\mathsf{C}}{=} \mathsf{R}^{G_1^2}$. By transitivity of indistinguishability it then follows that $\mathsf{H}^l \stackrel{\mathsf{C}}{=} \mathsf{H}^{l+1}$, Finally, again by transitivity of indistinguishability, this time a polynomial number of times in κ, we obtain that $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,0) = \mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1)$.

Step 4: $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1) = \mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1)$ The protocol $\mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1)$ differs from $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1)$ only in that a server S_i secret shares another key $\tilde{L}_i^r$ than the key L_i^r used to encrypt the file σ_i^r.

Because of the previous step the same values are sent on the network in both cases, so E cannot distinguish if only network corruptions occurs. By means of passively offline corruptions, E gets to see either shares of L_i^r or $\tilde{L}_i^r$ (the encrypted file C_i^r is the same in both cases). But since $\mathsf{E} \in \mathcal{E}\infty$, there will at most be $n = 1$ offline attacks in round r, and E will miss at least one share. This means that we can use the privacy property of the RSS scheme from Definition 2: Analogous to the previous steps we introduce an ordering $\pi(i,r) \mapsto \{0,1,\ldots,L-1\}$ and hybrid protocols H^l such that server S_i secret shares the same key L_i^r as used for encryption of σ_i^r as in $\mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1)$ if $\pi(i,r) \leq l$ and samples and shares another key $\tilde{L}_i^r$ as in $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1)$ if $\pi(i,r) > l$. Then $\mathsf{H}^0 \stackrel{\mathsf{P}}{=} \mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1)$, $\mathsf{H}^l \stackrel{\mathsf{P}}{=} \mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1)$.

H^l and H^{l+1} differ only by the secret sharing of a key by one server S_i in one round r and we show that for any $0 \leq l < L$ it holds that $\mathsf{H}^l \stackrel{\mathsf{C}}{=} \mathsf{H}^{l+1}$. This time the reduction R^{G_b} has oracle access to the IND-RSS game G_b: It simulates H^l with the following modifications: Let (i,r) be such that $\pi(i,r) = l$. Since corruptions are static in each round, the reduction knows at the beginning of round r which servers are corrupted in that round. Define these as $\{\mathsf{S}_j\}_{j \in T}$. The reduction first inputs $(L_i^r, \tilde{L}_i^r)$ to G_b. After receiving V_i^r it inputs T and receives as output a set of shares $\{\tilde{s}_{i,j}^r\}_{j \in T}$ which are embedded into the simulation, such that $\tilde{s}_{i,j}^r$ replaces $s_{i,j}^r$. The reduction outputs the same as its simulated environment. Since $G_0 \stackrel{\mathsf{C}}{=} G_1$, $\mathsf{P}_{\mathsf{CKM}}(0,1,1,1,1) \stackrel{\mathsf{C}}{=} \mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1)$ follows for the same reasons as in the previous step, that is, by the fact that indistinguishability is transitive and preserved under efficient transformations.

Step 5: $\mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1) = \mathsf{P}_{\mathsf{CKM}}(1,0,1,1,1)$ The difference here is that S_i in $\mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1)$ in each round r stores an encrypted dummy file $C_i^r = \mathsf{Enc}_{L_i^r}(0^{|\sigma_i^r|})$ while S_i in $\mathsf{P}_{\mathsf{CKM}}(1,0,1,1,1)$ stores instead an encryption of the real file $\mathsf{Enc}_{L_i^r}(\sigma_i^r)$. That is,

Due to the previous step, the keys used for encryption are chosen uniformly and without influence from E, so we get $\mathsf{P}_{\mathsf{CKM}}(1,1,1,1,1) = \mathsf{P}_{\mathsf{CKM}}(1,0,1,1,1)$ by an argument similar to Step 3 above, this time using IND-CPA of the encryption.

Step 6: $\mathsf{P}_{\mathsf{CKM}}(1,0,1,1,1) = \mathsf{P}_{\mathsf{CKM}}(0,0,1,1,1)$ The difference here is as in Step 4 whether the same key or an independent key is used for encrypting the secret file. This time, it is the real file and not a dummy file, but otherwise the argument is identical to that of Step 4.

Step 7: $\mathsf{P}_{\mathsf{CKM}}(0,0,1,1,1) = \mathsf{P}_{\mathsf{CKM}}(0,0,0,1,1)$ In $\mathsf{P}_{\mathsf{CKM}}(0,0,1,1,1)$ an honest server always returns the correct file to E on wakeup, that is, the file that E handed to the server at the previous shutdown. In $\mathsf{P}_{\mathsf{CKM}}(0,0,0,1,1)$ the file returned to E is that which is actually reconstructed in the protocol, so this step is essentially an argument for the correctness of the protocol: If no corruption occurs, correctness follows directly from the correctness of the encryption and secret sharing primitives used.

In addition, we need to argue that E cannot distinguish the file σ_i^r returned by honest S_i in round r from the correct file even if some or all of the returned shares have been modified – either due to active network attacks or because one or more of the other servers have been actively corrupted. Since rk_i^r and V_i^r never leaves S_i, this follows from the robustness of the RSS scheme.

More precisely, consider again a sequence of hybrid protocols H^l where the key L_i^r output by the RSS reconstruction algorithm in Step 3b of the wakeup phase is replaced by the correct key L_i^r for $\pi(i,r) \leq l$. That is, $\mathsf{H}^0 \stackrel{\mathsf{P}}{=} \mathsf{P}_{\mathsf{CKM}}(0,0,1,1,1)$, $\mathsf{H}^L \stackrel{\mathsf{P}}{=} \mathsf{P}_{\mathsf{CKM}}(0,0,0,1,1)$, and the difference between H^l and H^{l+1} is only with regard to the secret sharing at S_i in round $r+1$. Let E be the event that the reconstructed key is different from the key originally secret shared by S_i. We prove that $\mathsf{Pr}[E]$ is negligible in κ and the indistinguishability then follows since the conditional distribution $\mathsf{EXEC}_{\mathsf{H}^l,\mathsf{A},\mathsf{E}}|\bar{E} = \mathsf{EXEC}_{\mathsf{H}^{l+1},\mathsf{A},\mathsf{E}}$. Assume for the sake of contradiction that $\mathsf{Pr}[E]$ is non-negligible. We can then construct an adversary B for the RSS robustness game G from Definition 3 as follows: B simulates the execution and inputs the RSS key rk_i^r to G along with the key L_i^r shared by S_i. The shares $\boldsymbol{s}$ and public key V_i^r output from G are embedded in the execution (they have the same distribution as those originally used in the execution, so this cannot affect the output of E). The possibly modified shares $\boldsymbol{s}'$ that S_i receives at the wakeup phase of round $r+1$ are finally input to G. It follows that B wins the game G with the same non-negligible probability $\mathsf{Pr}[E]$ which contradicts the robustness of the RSS scheme.

Step 8: $\mathsf{P}_{\mathsf{CKM}}(0,0,0,1,1) = \mathsf{P}_{\mathsf{CKM}}(0,0,0,1,0)$ This step is needed before we can switch off the d-bit. The argument is similar to that of Step 3. The only difference is that now the real files and not dummy files are used in the protocol, but we can still use IND-CPA as in Step 3.

Step 9: $\mathsf{P}_{\mathsf{CKM}}(0,0,0,1,0) = \mathsf{P}_{\mathsf{CKM}}(0,0,0,0,0)$ This step is similar to Step 2 and relies on the security of the underlying AKE and Mac schemes. Again, the fact that real and not dummy files are used in the protocol does not make any difference.

Step 10: $\mathsf{P}_{\mathsf{CKM}}(0,0,0,0,0) = \mathsf{P}_{\mathsf{CKM}}$ This step follows simply by construction of $\mathsf{P}_{\mathsf{CKM}}(0,0,0,0,0)$.

Now, since the total number of reductions in the steps above is polynomial in the security parameter κ we get by transitivity of indistinguishability that $\mathsf{P}_{\mathsf{CKM}'} \stackrel{\mathsf{C}}{=} \mathsf{F}_{\mathsf{CKM}}$ or, using the standard UC notation, $\mathsf{EXEC}_{\mathsf{P}_{\mathsf{CKM}'},\mathsf{A},\mathsf{E}} \stackrel{\mathsf{C}}{=} \mathsf{IDEAL}_{\mathsf{F}_{\mathsf{CKM}},\mathsf{S},\mathsf{E}}$. □

The informal Theorem 1 in Section 4.2 follows immediately from Theorem 4 and the definition of the ideal functionality $\mathsf{F}_{\mathsf{CKM}}$.

B.6 The ROM Model and Semi-Autonomous Servers

In the previous section we modelled the basic $\mathsf{P}_{\mathsf{CKM}'}$ protocol without the ROM assumption and without administrators. In this section we show how to extend our model in the UC framework to also cover the ROM assumption with increased offline security and the semi-autonomous model with administrators.

Threshold Signatures In order to formally model our protocol for semi-autonomous servers we will need a proactive threshold signature scheme. Such a scheme allows the signing key to be split into several shares distributed among the servers in such a way that all servers must participate in order to generate a valid signature and such that the adversary must know *all* shares in the same round before he can produce his own signatures. We can simplify our proof by treating this as an ideal functionality $\mathsf{F}_{\mathsf{THSIG}}$, proving security in the $\mathsf{F}_{\mathsf{THSIG}}$-hybrid model.

In the informal discussion, strong security against active offline attacks in the ROM model was achieved by having each serer hash its state before the offline period. It turns out that the same offline security can be achieved, still in the ROM model, but using a threshold signature scheme, if the ROM assumption means that not only the code, but also the public verification key of the threshold scheme is embedded in ROM and cannot be altered. Since the extension for the semi-autonomous servers already involves such a threshold signature scheme, in this section we choose to model the full $\mathsf{P}_{\mathsf{CKM}}$ protocol using this.

Almansa *et al.* [1] provide a UC-secure protocol that realizes the ideal functionality $\mathsf{F}_{\mathsf{THSIG}}$ listed in Fig. 8. Their protocol relies on secure point-to-point channels and setup assumptions. It is originally designed to always terminate, but has only full threshold in the passive setting while requiring honest majority against active adversaries. We show below that by giving up on termination, we can get full threshold security also against active adversaries.

Refreshment $\mathsf{F}_{\mathsf{THSIG}}$ maintains a round counter r that is incremented on input (`refresh`) from all honest servers.

Signature Generation The first time in each round that input $(\texttt{sign}, m)$ from all honest servers have been received, send $(\texttt{sign}, m)$ to A. If A returns $(\texttt{sign-ok}, m, \sigma)$ and $(m, \sigma, \texttt{invalid})$ was not already stored, record $(m, \sigma, \texttt{valid})$ and send $(\texttt{signed}, m, \sigma)$ to all honest servers.

Signature Verification On $(\texttt{verify}, m, \sigma, v')$ from S_i, send $(\texttt{verify}, m, \sigma, v')$ to A. If $(\texttt{verified}, m, \sigma, \phi)$ is then received from A, send $(\texttt{verified}, m, \sigma, f)$ to S_i where f is determined as follows:

1. If $v' = v$ and $(m, \sigma, \texttt{valid})$ is stored, set $f = 1$ (ensures that if v' is the right public key and σ is correctly generated, then verification succeeds).
2. Else if $v' = v$ and no entry $(m, \sigma, \texttt{valid})$ is stored, set $f = 0$ (ensures that if v' is correct and m was not legitimately signed, then verification fails). Record $(m, \sigma, \texttt{invalid})$.
3. Else, if $v \neq v'$, set $f = \phi$.

Fig. 8. The ideal functionality $\mathsf{F}_{\mathsf{THSIG}}$ adapted from Almansa *et al.* [1]

$\mathsf{P_{CKM}}$ in the ROM Model We build the trusted setup needed by $\mathsf{F_{THSIG}}$, that is, the distribution of a public verification key $\mathcal{W}$, into our already existing setup $\mathsf{F_{SETUP}}'$ and denote the extended setup by $\mathsf{F_{SETUP}}$. The extended protocol $\mathsf{P_{CKM}}$ is depicted in Fig. 9.

Initialization As $\mathsf{P_{CKM}}'$ except that the extended $\mathsf{F_{SETUP}}$ is used to also receive a public verification key $\mathcal{W}$ and that also $\mathsf{F_{THSIG}}$ is initiated.

Shutdown As $\mathsf{P_{CKM}}'$ except that just before returning, S_i concatenates all state that is not erased into one string Δ_i^r. Then all servers together compute a signature δ_i^r of Δ_j^r using $\mathsf{F_{THSIG}}$. Server S_i finally stores the signature δ_i^r along with Δ_i^r during the offline phase.

Wakeup As $\mathsf{P_{CKM}}'$, but before everything else, S_i obtains its online state by first using the public verification key $\mathcal{W}$ and $\mathsf{F_{THSIG}}$ to verify the signature δ_i^{r-1} on Δ_i^{r-1}. If the signature is valid, S_i extracts the encrypted file, etc., from Δ_i^{r-1} and continues as in $\mathsf{P_{CKM}}'$. Otherwise it aborts. After the verification, all servers refresh the signature scheme by calling (`refresh`) on $\mathsf{F_{THSIG}}$.

Fig. 9. The protocol $\mathsf{P_{CKM}}$ (with strong offline security in the ROM model).

In the previous section we considered the environments $\mathcal{E}_1$. We now consider another environment class $\mathcal{E}_2$:

Definition 7. *Let $\mathcal{E}_2$ be as $\mathcal{E}_1$ except that if an adversary in $\mathcal{E}_2$ actively offline corrupts a server, that server is no longer, as in $\mathcal{E}_2$, forced to be actively corrupted from that point and onwards: The adversary is allowed to let the server be only passively corrupted or honest in the following online phase. If he does so, the adversary must however restore the correct public verification key $\mathcal{W}$ on the server.*

That is, $\mathcal{E}_2$ models the ROM assumption described above: An actively offline corrupted server turning honest in the following wakeup phase corresponds to a server that during the offline phase has stored its code and the threshold verification key in read-only memory thereby preventing the adversary from modifying it even though he actively breaks into the server during the offline phase. We stress, however, that for $\mathcal{E}_2$ it is still so that if a server at any time gets actively corrupted *online* it must remain actively corrupted from that point and throughout the protocol.

We now describe a modification of the threshold RSA protocol by Almansa *et al.* [1] that will realize $\mathsf{F_{THSIG}}$ securely with respect to adversaries in $\mathcal{E}_2$. Here it is crucial that $\mathsf{F_{THSIG}}$ explicitly allows the adversary to stop signatures from being generated.

In the original protocol [1] the secret RSA exponent d is additively shared among the servers and each share d_i is verifiably secret shared (VSS'ed) among the servers. The VSS is done with threshold $n/2$. The changes we introduce are as follows: First, we do all VSSs with threshold $n-1$. This means that whenever a VSS is opened, the correct secret is reconstructed, or we abort. Second, at the end of each signature generation, each server checks that a correct signature has been generated and aborts if not. Third, in the refreshment protocol, a server S_i who has been offline corrupted may not have any reliable information stored about his state. This issue also occurs in the original protocol [1] where it is solved by having each server send information (VSS shares of d_i and public values) to S_i. Because the original protocol [1] assumes honest majority, in that protocol S_i can always reconstruct a correct state. In our case we demand instead that all the information S_i receives must be consistent and sufficient to reconstruct the state, and otherwise S_i will abort.

Lemma 1. *The RSA threshold signature protocol of Almansa et al. [1] modified as above UC-securely realizes $\mathsf{F_{THSIG}}$ in our model, that is, with respect to the environment class $\mathcal{E}_2$.*

This can be shown by basically repeating the proof and simulator from the original paper [1], except that at each point where the adversary behaves in a way that makes an honest player abort, the simulator makes the functionality $\mathsf{F}_{\mathsf{THSIG}}$ stop. Since we work with threshold $n-1$, no signatures can be generated in real life after an abort, and this way we ensure that this is also the case in the simulation. As long as there is no abort, the simulation works and is indistinguishable for the same reason as in the original proof.

Theorem 5. *Let* $\mathsf{F}_{\mathsf{CKM}}{}'$ *be as* $\mathsf{F}_{\mathsf{CKM}}$ *but with the addition that it does not allow* A *to modify* σ_i^r *if* S_i *is actively corrupted only in the offline phase. Let* $\mathsf{P}_{\mathsf{CKM}}$ *denote the* $\mathsf{F}_{\mathsf{SETUP}}, \mathsf{F}_{\mathsf{THSIG}}$*-hybrid multiparty protocol defined in Fig. 9. Then* $\mathsf{P}_{\mathsf{CKM}}$ *UC-realizes* $\mathsf{F}_{\mathsf{CKM}}{}'$ *with respect to the class of environments* $\mathcal{E}_2$*.*

The proof of Theorem 5 works in a similar fashion as that of Theorem 4, except that servers in the real world have access to the ideal functionality for threshold signatures, and is not included here.

Semi-Autonomous Servers In the UC framework we model the semi-autonomous servers by associating with each server S_i a corresponding incorruptible party Adm_i.

$\mathsf{F}_{\mathsf{DROPBOX}}$ in Fig. 10 below models what is essentially a public "dropbox". An administrator Adm_i can use this to reliably fetch information from any honest server as long as this information has been marked as "fetchable" by the server. In many real-life scenarios Adm_i would not directly fetch information on other servers than his own, but rather contact the other servers' administrators who would then, in turn, fetch the required information from the servers that they control and send it back to Adm_i. However, for simplicity we do not model this in $\mathsf{F}_{\mathsf{DROPBOX}}$.

Drop On input $(\texttt{drop}, m)$ from server S_i output $(\texttt{drop}, m, i)$ to the adversary and store $(\texttt{drop}, m, i)$.
Fetch On input $(\texttt{fetch}, i)$ from any administrator Adm_j, if $(\texttt{drop}, m, i)$ is stored for some m, send $(\texttt{fetch}, m, i)$ to Adm_j.
Corruption If S_i gets passively corrupted, output $(\texttt{dropped}, m)$ to A. If S_i gets actively corrupted, output $(\texttt{dropped}, m)$ to A and furthermore, on input $(\texttt{modify}, m')$ from A, replace m by m'.

Fig. 10. The ideal functionality $\mathsf{F}_{\mathsf{DROPBOX}}$.

Note that in $\mathsf{F}_{\mathsf{DROPBOX}}$ it is the administrator and not the servers that initiate communication. That is, servers cannot push messages to administrators, but instead administrators have to "visit" servers in order to see whether there is any messages. Note also that the dropbox comes with erasure: Once a server stores a new message, only the new message and not the previously stored messages leak to the adversary upon corruption. Since the dropbox will typically reside on the same computer as the server itself and thus be fully controlled by the server (for which we already assume erasure), this seems to be a reasonable assumption.

We do not provide any protocol realizing $\mathsf{F}_{\mathsf{DROPBOX}}$. Rather, we argue that in reality, if the humans controlling the servers, say, the system administrators, get actively involved, achieving a functionality as defined by $\mathsf{F}_{\mathsf{DROPBOX}}$ will almost always be possible. The administrators may for example communicate by meeting physically or via some external trusted channels such as telephone lines or mail. Communicating via $\mathsf{F}_{\mathsf{DROPBOX}}$ will of course often be much more expensive and time consuming than communication via the standard network.

Note also that $\mathsf{F}_{\mathsf{DROPBOX}}$ delivers the fetchable information *directly* to the administrator upon request. This means that the adversary is not even allowed to delay the

message for a while. While defined this way here for simplicity, this assumption is actually stronger than we need. Roughly speaking we only require that the adversary cannot delay the messages *forever*. In [33] it is shown how to model this in the UC framework.

Let $\mathsf{F_{CKM}}^*$ be as $\mathsf{F_{CKM}}$, but with the following addition:

File Recovery On input (`recover-file`) from Adm_i, send back $(\texttt{recover-file}, r, \sigma_i)$ to Adm_i where r is the largest r such that $(\texttt{offline}, r, \mathsf{S}_j, \sigma_j)$ is stored for all honest servers S_j.

We will also need an extended trusted setup $\mathsf{F_{SETUP}}^*$ that apart from setting up initial session keys as in $\mathsf{F_{SETUP}}$, also equips each S_i with a public encryption key $\mathrm{ek}_i^{\mathrm{adm}}$ and hands the corresponding decryption key $\mathrm{dk}_i^{\mathrm{adm}}$ to its administrator Adm_i. $\mathsf{F_{SETUP}}^*$ also initializes a threshold signature scheme, handing the public verification key $\mathcal{W}$ to all servers and administrators and a share w_i of the signing key to each server S_i.

Initialization As $\mathsf{P_{CKM}}$ except that also $\mathsf{F_{DROPBOX}}$ is initialized. Also, from $\mathsf{F_{SETUP}}^*$ the additional public key $\mathrm{ek}_i^{\mathrm{adm}}$ of the server's administrator and the public and private keys $\mathcal{W}$ and w_i for a proactive threshold signature scheme are obtained.

Shutdown As $\mathsf{P_{CKM}}$, with these additional steps performed by server S_i before Step 4 (erasing values) of $\mathsf{P_{CKM}}$:

1. Compute $F_i \leftarrow \mathsf{Enc}_{\mathrm{ek}_i^{\mathrm{adm}}}(\sigma_i)$.
2. Compute a threshold signature f_i of F_i by invoking $\mathsf{F_{THSIG}}$.
3. Verify f_i using the public verification key $\mathcal{W}$ and abort if invalid.
4. Mark F_i and f_i as fetchable by sending $(\texttt{drop}, (F_i, f_i))$ to $\mathsf{F_{DROPBOX}}$.
5. Send (F_i, f_i) to all other servers.
6. When a pair (F_j, f_j) is received from another server S_j, verify the signature f_j and abort if invalid. Otherwise, mark the pair as fetchable by outputting $(\texttt{drop}, (F_j, f_j))$ to $\mathsf{F_{DROPBOX}}$ and return an authentic receipt to S_j using $K1_{i,j}^{\mathrm{mac},r}$. [a]
7. Only return if valid signatures have been received from all other servers, and valid receipts from all servers have been received for the (F_i, f_i) that was sent out from this server.

Wakeup As $\mathsf{P_{CKM}}$ but servers also invoke the (`refresh`) method of $\mathsf{F_{THSIG}}$.

File Recovery When Adm_i receives (`recover`) from the E, it does as follows:

1. Use $\mathsf{F_{DROPBOX}}$ to fetch messages (F_j, f_j) from the servers (can be done in parallel).
2. When a message (F_i, f_i) is fetched from a server S_j the signature f_i is verified. If valid, compute $\sigma_i \leftarrow \mathsf{Dec}_{\mathrm{dk}_i^{\mathrm{adm}}}(F_i)$ and return (r, σ_i) to E. If invalid, fetch a message from another server.

Corruption As $\mathsf{P_{CKM}}$.

[a] The receipts ensure that once S_i outputs (`offline`), all honest servers have marked S_i's file as fetchable.

Fig. 11. The operations of the protocol $\mathsf{P_{CKM}}^*$ (semi-autonomous servers) .

The extended protocol $\mathsf{P_{CKM}}^*$ is defined Fig. 11. It is identical to $\mathsf{P_{CKM}}'$ except that some additional steps are performed.

Definition 8. *Denote by $\mathcal{E}_3$ the same environments as $\mathcal{E}_1$, but with the extra constraint that adversaries in $\mathcal{E}_3$ do not corrupt administrators.*

Theorem 6. *Let $\mathsf{P_{CKM}}^*$ denote the $\mathsf{F_{SETUP}}*, \mathsf{F_{DROPBOX}}, \mathsf{F_{THSIG}}$-hybrid multiparty protocol defined in Fig. 11. Then $\mathsf{P_{CKM}}*$ UC-realizes the ideal functionality $\mathsf{F_{CKM}}^*$ with respect to the class of environments $\mathcal{E}_3$.*

The informal Theorem 3 in Section 4.2 follows immediately from the above theorem and the definition of $\mathsf{F_{CKM}}^*$.