




Composable and Robust Outsourced Storage

Christian Badertscher^(✉)  and Ueli Maurer

Department of Computer Science, ETH Zürich, Zürich, Switzerland
{badi,maurer}@inf.ethz.ch

Abstract. The security of data outsourcing mechanisms has become a crucial aspect of today's IT infrastructures and they are the cryptographic foundations of real-world applications. The very fundamental goals are ensuring storage integrity and auditability, confidentiality, and access pattern hiding, as well as combinations of all of them. Despite sharing a common setting, security analyses of these tasks are often performed in a stand-alone fashion expressed in different models, which makes it hard to assess the overall security of a protocol or application involving several security schemes at once. In this work, we fill this gap and propose a composable framework suitable to capture various aspects of outsourced storage security and its applications. We instantiate the basic client-server setting in this model, where the goal of the honest client is to retain security in the presence of a malicious server. Three specific contributions of this paper are:

1. We present a novel definition for secure and robust outsourcing schemes and underline why this is needed in practice. Our definition is stronger than previous definitions for oblivious RAM or software protection in that it assures strong security guarantees against active attacks. Schemes meeting the definition not only assure that an attacker cannot learn the access pattern, but guarantee resilience to errors and the prevention of targeted attacks to specific locations. Unfortunately, several existing schemes cannot achieve this high level of security. For completeness, we provide a protocol based on Path ORAM that showcases that stronger security is actually achievable.
2. We present a novel definition for auditable storage, capturing the guarantee that a successful audit implies that the current server state allows the client to retrieve his data. We develop an audit mechanism, based on secure and robust outsourcing schemes, that is similar to the construction by Cash et al. (Eurocrypt 2013), but is universally composable and fault-tolerant.
3. We revisit the security claim of a widely-used challenge-response audit mechanism, in which the server has to compute a hash $H(F||c)$ on the file F concatenated with a uniformly random challenge c chosen by the client. Being concerned with composable security, we prove that this audit mechanism is not secure, even in the random oracle model, without additional assumptions. The composable security of this basic audit scheme was implicitly assumed in Ristenpart et al. (Eurocrypt 2011). To complete the picture, we state the additional assumptions for this audit mechanism to be provably secure and discuss the implication on practical security.

1 Introduction

An integral and pervasive part of today's IT infrastructures are large amounts of outsourced data ranging from personal data to important enterprise backups on third-party storage providers. Depending on the various applications and sensitivity of the data, a user paying for remote storage might not fully trust in the provider's content management or security. Client-side countermeasures have to be taken into account, a prominent example of which are the protection of confidentiality and integrity of the uploaded files, or hiding the access pattern to files. A client further might want to audit the server storage to ensure that the provider maintains all his data consistently and is not saving space by deleting a fraction of the content. That is generally known as proofs of retrievability (PoR) or provable data possession (PDP) [2, 13]. Complementary to protocols for clients to retain security against a possibly malicious server, another line of research deals with mechanisms for secure deduplication and proofs of ownership [11, 14]. These protocols allow an honest server to reduce its storage requirements while protecting against malicious clients that try to fool the server by accessing files they do not possess.

In this work, our focus is on malicious server behavior. Reasons for such dishonest behavior include ordinary failures that lead to data loss or data leakage, an active break-in into the provider's infrastructure or intentional malicious server strategies. A client can employ protection mechanisms to ensure integrity, confidentiality, hide its access pattern to the data, or run regular audits to ensure that the server maintains the data reliably such that the client is able to retrieve it. Although service providers advertise availability as an important selling point, such audits are a key tool to increase the confidence or trust in the service since it is often not realistic to rely on the provider to inform reliably about an incident, either due to ignorance or due to the fear of bad reputation.

Despite sharing a common setting, previous security analyses of these tasks are often performed in different models and in a stand-alone fashion, which makes it hard to assess the overall security of a protocol (e.g. a cloud application) that involves several security schemes. In this work, we fill this gap and provide a unified composable model for capturing the security of outsourced storage. As part of this study, we justify the need for stronger security requirements from protocols than what is typically assumed in the literature. Our approach lets us develop outsourcing schemes in modular steps that provably achieves stronger security than existing protocols. For completeness we give one such example.

We formulate our model in the language of the constructive cryptography framework (CC) [16, 17]. Our results are not specific to the CC framework itself and choosing another definitional framework like Canetti's Universal Composition (UC) framework [7] would yield closely related findings [12]. A central aspect of CC is that the resources available to the parties, such as communication channels or an untrusted server storage, are made explicit. The goal of a cryptographic protocol is then to securely construct, from certain existing or

assumed resources (often called “real world”), another, more desirable resource (often called “ideal world”). A construction is secure if the real world is as useful to an adversary as the ideal world, the latter world being secure by definition. Formally, one has to construct a simulator in the ideal world to make the two worlds computationally indistinguishable.

The resources we consider in this work are variations of so-called *server-memory resources*. A typical example of a construction would be to construct a server-memory resource providing integrity from one that does not have this property. A constructed resource can then again be used by higher-level protocols or applications. This allows for modular protocol design and to conduct modular security analyses by dividing a complex task into several less complex *construction steps*, where each step precisely specifies what is assumed and what is achieved, and the security follows from a general composition theorem.

1.1 Summary of Results and Contributions of This Work

A model for untrusted storage. The basic functionality we consider is an (insecure) *server-memory resource* which we denote by **SMR** and formally specify in Sect. 3. One or several clients can write to and read from this resource via interfaces. Clients write to the memory in units of blocks, and the resource is parameterized by an alphabet Σ and the size n of blocks. The server can access the entire history of read/write requests made by the clients. To capture the active server influence on the storage, including malicious intrusion, the resource can be adaptively set into a special *server write mode* that allows the server overwrite existing data. Within the scope of this paper, we understand this write phase as being malicious and the server is not supposed to change any data. However, we point out that this server write mode can be used to capture intentional, honest server-side manipulations of the data storage, as in de-duplication schemes or proofs of ownership. See also Fig. 1 for a graphical illustration.

The decision in which “mode” the resource resides, is given directly to the environment (or distinguisher) and not to the adversary. The reason for this is important for technical and motivational reasons. Assume that the capability is provided at the malicious server interface both in the “real world” and in the “ideal world,” then the simulator in the ideal world can always make use of the capability of overwriting the memory content and nothing would prevent the simulator from doing so all the time and hence trivial protocols could be simulated. However, in this work, we want to express security guarantees in both cases, when the resource is “under attack” and when it is not. To achieve this, the “attack mode” is under the control of the environment and not the adversary. Furthermore, in certain cases we only want to give explicit security guarantees that hold only until the next attack happens (for example in the case of audits as explained later). From a motivational point of view, assigning the capability to the environment and not to the attacker yields more general statements, as it also allows us to capture scenarios where the server does indeed

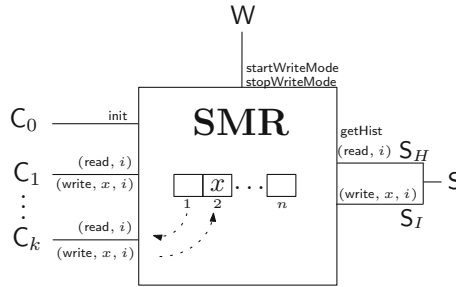


Fig. 1. The basic server-memory resource.

not have the active choice to do so, but where an external event can provoke the server memory to be corrupted. A basic server memory can be strengthened in the following ways to give better guarantees to clients:

- The *authentic* server-memory, providing authenticity of the memory content (meaning that clients detect adversarial modifications).
- The *confidential* and authentic server memory, providing secrecy (in addition to authenticity) of the memory content.
- The *secure* server memory. It provides full secrecy on the entire structure of the memory. An attacker cannot learn anything beyond the number of accesses and cannot tamper with specific logical memory cells.
- The *auditable* memories, including auditable versions of all of the above, gives the client the additional capability to check whether the memory has been modified or deleted, without the need to read the entire memory.

We explain how to achieve each of these resources, either by giving new protocols or showcasing how previous results are cast in this model.

A novel notion for secure and robust outsourcing schemes. Our definition of a *secure server-memory resource* can be seen as a novel security goal: The specification demands secrecy of content and access pattern, resilience to errors, and also that active attacks cannot be targeted at specific locations. On a more technical level, our secure server-memory resource is specified as a basic server-memory resource, but where roughly only the number of accesses leak to the server, and in particular not the content. In addition, the active influence by an attacker is restricted to being able to set a failure probability α . This parameter defines with which probability a client’s read or write operation fails. This failure probability is the same for all memory locations and each memory location fails independently of other memory locations. This means that whatever the attacker does to the memory of the server, any modification will result in clients being more or less successful in reading or updating the data. In case of a failure, the client cannot read or update the corresponding block anymore. We further demand that the memory, and thus any protocol achieving it, remains operational for the faultless part of the memory and hence is *robust* in the presence of failures. As outlined above, this is technically enforced by not giving the

simulator the power to always block operations and hence to abort. This makes the functionality stronger than existing models such as [1, 8, 9].

Surprisingly, the existing definitions for access-pattern hiding (such as ORAM) and software protection are insufficient for realizing secure server-memory resources. We exemplify this by two concrete examples that do not realize a secure server memory either because the failure probability is not the same for all locations (as in [10]), or failures among memory locations are correlated (as in [21]) and explain how this can be abused in practice as a side-channel leaking information about the access pattern. On the positive side, we provide a modification of an existing ORAM scheme that realizes a secure server memory.

A novel notion for audit schemes. What is generally expected from an audit scheme is that if the test is successful, then, in principle, the data is retrievable. Being concerned with realistic client-side security guarantees, we observe that the position of the client is very weak: no scheme can prevent that after a successful audit, the server denies to reveal the data. The best we can hope for in terms of practical guarantees is therefore to formalize that if (a) the audit succeeds and (b) the adversary does not corrupt further memory cells after the audit (i.e., it does not prevent them from being read by the client), then the server state is indeed sufficient for the client to reconstruct his data. We follow this idea and propose the *auditable* server memory resources as a formalization of this goal. They are, to the best of our knowledge, the first composable security definition for audit schemes. In a nutshell, these are server-memory resources with the additional client-side capability of asking whether the current memory content is unchanged. This retrievability guarantee is valid if the resource is not in “adversarial write mode”, as explained above, and holds up to the point when the server writes or deletes a location of the memory. A new audit has to reveal whether any change affected the client’s data.

While this notion seems at first sight to be much weaker than what is formalized by proofs of retrievability or provable data possession [2, 13, 19], the above guarantee is in fact not implied by existing definitions and there exist schemes that do not provide it. An example follows in Sect. 1.2. A second issue to practical security with existing definitions of PoR and PDP is their use of knowledge-extractors: while useful to argue about the principle of knowledge, the extractor has a drawback: it needs the client secrets and the server strategy to recover the data.¹ Both of these inputs are arguably not provided by the respective entities as this information is way too crucial and jeopardizes security. Hence, even though such an algorithm exists that recovers the data, it cannot be applied by the client, nor by the server in general. We give an in-depth comparison in the full version of this work [4].

¹ As for proofs of knowledge, the extractor needs the capability to “rewind” the server an efficient number of times and would therefore need special access to the server program, which is not covered by a typical API.

Audits for secure memories. In the particular case of secure server-memory resources, the audit reduces to a statistical estimate of the failure parameter α in combination with appropriate data replication. Our simple protocol resembles the protocol by Cash et al. [8], but is more robust against failures: While their construction aborts when detecting an error, our scheme keeps operating even in the presence of arbitrarily many errors. As we outline in Sect. 1.2, this robustness is not simply a theoretical need, but a realistic desire and in fact achievable.

A critical look at hash-based challenge-response audits. A composable formalization of storage audits in the spirit of indistinguishability and constructive cryptography [18, Sect. 7] has been envisioned in [20] but has not, to the best of our knowledge, been formalized. With our formalization, we are now able to re-assess the security of the main example in [20], which is the standard challenge-response audit mechanism in which the server computes a hash on the current memory content concatenated with a uniformly random challenge chosen by the client to convince the client that the data is available. We show that this scheme is not secure even in the random oracle model, contradicting the claimed security in [20]. We further prove that the additional assumption needed for the hash-based audit to be secure is to restrict inputs to the random oracle to bitstrings stored in the server memory itself. This condition is sufficient for a “monolithic” random oracle with no particular underlying structure, and we show that it is in general insufficient if the random oracle is replaced by a construction (like NMAC) from ideal compression functions.

Development of efficient, provably secure schemes. Our model is expressed in a composable language and thereby suitable for the design of practical, efficient, and provably secure protocols: In order to construct the strongest version of a server memory, i.e., a secure and auditable server-memory resource, one can apply a sequence of simpler modular steps as follows: (1) construct an authentic from a basic server memory, (2) construct a confidential from an authentic server memory, (3) construct a secure from a confidential server memory, and (4) construct an auditable secure server memory from a secure server memory. Combining these four steps, the composition theorem directly assures the security of the overall construction, which is a protocol that realizes an auditable, secure server-memory resource from a basic server-memory resource.

Beyond the fact that modular steps are often simpler to analyze than an entire protocol, this approach has a further benefit: it allows to identify sources of inefficiency and to improve single steps in isolation without the need to re-prove the security of the overall construction: for example, developing a more efficient ORAM scheme—that in addition meets our stronger requirements demanded by step (3)—directly gives an improved overall construction. As such, we put forward this approach to provide an interface to specialized works focusing on the individual steps, such that they can directly contribute to the development of outsourcing schemes.

Any protocol built this way provides stronger security than typical protocols in this realm: it is resilient against any number of errors, hides the content and access pattern, does not allow targeted attacks under any circumstances, and provides an audit function.

1.2 On the Practical Importance of Composition and Robustness

In this section, we provide a comparison between the most important aspects of this work and related literature. In the full version of this paper [4] we further elaborate on additional relevant work in the broad area of outsourced storage to provide a more complete picture on this topic. Our setting has similarities with previous works that devise outsourcing schemes secure against active tampering adversaries and which build upon the foundational work by Goldreich and Ostrovsky [9] on software protection. There is, however, a subtle and fundamental difference between the context of outsourced storage and the context of software protection of [9] that seems to have gone unnoticed. In this paragraph, we show how this difference necessarily leads to strictly stronger security requirements for outsourcing schemes and even gives rise to novel security-relevant questions, which we answer in this work.

The context of [9] is software protection, where the goal is to prevent that an experimenter can analyze the CPU-program and learn something he could not deduce from the program specification alone. Technically, a simulator must generate an indistinguishable transcript of any experiment, solely based on the known program specification. If such a simulator exists, this means that the program effectively defeats experiments that try to figure out secret details on “how the program internally works”. Following this motivation, as soon as the program encounters an error when reading a memory location, it should abort, as the error is a sign that the software is running in a tampering experiment. In the corresponding simulation, the simulator also aborts. Overall, this behavior makes perfectly sense to defeat experiments since in any honest execution, no error is expected to occur.

The context in this work is outsourcing schemes and several of the above aspects do change in this realm. We present outsourcing schemes and the idealization they can achieve, like the secure server-memory resource, as a low-level primitive that exports the interface of a consistent storage with certain additional guarantees. We do not allow our primitives to abort in case an access to a location returns an error. It must stay operational for the remaining part of the memory. The decision to abort is left to the calling protocol or application that uses the memory abstraction. In our context, we want and should react to errors and not stop when detecting them. This is the first important point that makes the problem more difficult and gives rise to the question of what level of security we can achieve in this setting. Our most secure abstraction, the secure server-memory, answers this question in a strong way: a protocol that achieves the secure server-memory not only remains operational (and efficient) when tampering is detected (a simulator cannot “abort on error” in a simulation in our model), it also makes sure that the subsequent behavior does not reveal which logical locations the client accesses, and furthermore prevents that tampering can be targeted at specific logical locations.

We illustrate two possible security issues which are overcome by using a secure outsourcing scheme (as part of a larger system) that fulfills our strongest notion.

Example 1: Information leakage due to errors. Assume that a client application stores some control information on an outsourced storage using a secure outsourcing scheme that achieves a secure server-memory as defined in our work. Clearly, there is no attack by which the adversary could learn when the client accesses the control information, even if the attacker knew at which logical location the control information is stored. And since the attacker can only introduce failures that are equally likely for all logical locations, the occurrence of an error during an access does not allow to infer which logical memory location was accessed. In contrast, several existing schemes based on the notion of software protection, do not guarantee this level of security and allow an attacker to obtain side-channel information about the access pattern through the observed error-pattern. This holds for example if one can approximately estimate which logical addresses are targeted by tampering with the memory, or if errors are correlated. Turned around, an observed error pattern can be a good indication on which logical locations have been accessed as further discussed in Sect. 5.1.

Example 2: Implementing secure audits. Let us focus on a protocol by Cash et al. [8] that implements a proof of retrievability using a software protection scheme S that aborts on error. The following argument is independent on what security notion S fulfills exactly, the important thing is that if S aborts then the entire execution aborts. Their protocol invokes S to store the encoded data redundantly on the server, which should improve the resilience, i.e., not detecting a few errors on the server should not let the protocol fail in retrieving the data. However, since S aborts when detecting even a single (e.g., physical) error, this desired resilience practically becomes ineffective and leads to weak guarantees: consider a very weak tampering adversary that chooses just a single, physical location on the server-memory and only tampers with this single physical location. Then, the audit is passed with high probability (the data is actually still there due to the encoding). However, the client protocol will abort before the client can actually retrieve all his data, since the error is detected beforehand, namely during a rebuild phase of S , and the execution is aborted. And as shown above in the first example, simply letting the protocol continue its actions can reveal information on the access pattern. This indicates that such a patch is actually non-trivial. In particular, if S was proven to realize a secure server-memory, then this issue is avoided.

2 Preliminaries

In this section, we succinctly present the language needed to understand the main results of this paper. In the full version of this work [4], we additionally provide a more detailed technical introduction.

Notation for Systems and Algorithms. We describe our systems with pseudocode using the following conventions: We write $x \leftarrow y$ for assigning the value y to the variable x . For a distribution \mathcal{D} over some set, $x \leftarrow \mathcal{D}$ denotes sampling x according to \mathcal{D} . For a finite set X , $x \leftarrow X$ denotes assigning to x a uniformly

random value in X . Typically queries to systems consist of a suggestive keyword and a list of arguments (e.g., (write, i, v) to write the value v at location i of a storage). We ignore keywords in writing the domains of arguments, e.g., $(\text{write}, i, v) \in [n] \times \Sigma$ indicates that $i \in \{1, \dots, n\}$ and $v \in \Sigma$. The systems generate a return value upon each query which is output at an interface of the system. We omit writing return statements in case the output is a simple constant whose only purpose is to indicate the completion of an operation.

Discrete Systems. The security statements in this work are statements about reactive discrete systems that can be queried by their environment: Each interaction consists of an input from the environment and an output that is given by the system in response. Discrete reactive systems are modeled formally by random systems [15], and an important similarity measure on those is given by the distinguishing advantage. More formally, the advantage of a distinguisher \mathbf{D} in distinguishing two discrete systems, say \mathbf{R} and \mathbf{S} , is defined as

$$\Delta^{\mathbf{D}}(\mathbf{R}, \mathbf{S}) = |\Pr[\mathbf{DR} = 1] - \Pr[\mathbf{DS} = 1]|,$$

where $\Pr[\mathbf{DR} = 1]$ denotes the probability that \mathbf{D} outputs 1 when connected to the system \mathbf{R} . More concretely, \mathbf{DR} is a random experiment, where the distinguisher repeatedly provides an input to one of the interfaces and observes the output generated in reaction to that input before it decides on its output bit.

Construction statements. The security statements proven in this work express that clients realize or construct a desired resource (or ideal functionality), for example a secure server-memory resource denoted \mathbf{sSMR} , from assumed resources (or hybrid functionalities), such as an authentic and confidential server-memory resource, denoted \mathbf{cSMR} , and a small (local) client storage, denoted \mathbf{L} . They construct the desired resource by running a protocol. This situation is depicted in Fig. 2. We briefly discuss the involved formal expressions for this main example.² For a construction, we need to prove the security condition of constructive cryptography. This condition ensures that whatever a dishonest server can do with the assumed resource, he could do as well with the constructed resource by using the simulator sim . Turned around, if the constructed resource is secure by definition, there is no successful attack on the protocol. For readers more familiar with the UC framework, this essentially corresponds to the notion of *UC secure realization* (we refer to [12] for a more detailed comparison). Formally, we show that the distinguishing advantage of the two worlds is small (in this case even zero), i.e.,

$$\Delta^{\mathbf{D}}(\text{sec}_{\mathcal{P}}[\mathbf{L}, \mathbf{cSMR}], \text{sim}^{\mathbf{S}}\mathbf{sSMR}) = 0,$$

where $\text{sec}_{\mathcal{P}}[\mathbf{L}, \mathbf{cSMR}]$ denotes that the client converters are attached at their respective interfaces. The set \mathcal{P} denotes the set of client interfaces and $\text{sec}_{\mathcal{P}}$ is

² We present here the particular instantiation of the construction notion of constructive cryptography that is necessary to understand this work. We refer to the full version for a general definition.

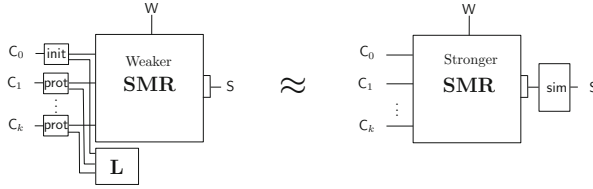


Fig. 2. Illustration of the security condition in our setting.

formally a vector of converters to succinctly express the situation in Fig. 2. The simulator is attached at the dishonest server interface.

Second, we need to satisfy the correctness (or non-triviality) condition of constructive cryptography that ensures that the protocol implements the required functionality in case the server is honest. This condition is typically easy to achieve for well-designed protocols and always ensured in this work. Formally, honest server behavior is represented by a special protocol converter honSrv attached at interface S . For outsourced server-memory resources this is a server protocol that does not interfere with the client protocol. For implementing audits, the (honest) server protocol might do some computation to support the client, such as computing a hash. In this case, we prove

$$\Delta^D(\text{honSrv}^S \text{ sec}_{\mathcal{P}}[\mathbf{L}, \text{cSMR}], \text{honSrv}^S \text{sSMR}) = 0.$$

A special role in our model takes the interface W , also denoted to as the world interface, which models the direct influence of a distinguisher on a resource. Hence, no converter is attached at that interface.³ This interface is our method of choice to model capabilities which should not be assigned to a party or the attacker and which allows to derive more general statements as explained in the introduction and already used in [6] in a different context.

3 Basic Server-Memory Resource

The basic server-memory resource allows clients to read and write data blocks, where each block is encoded as an element v of some alphabet Σ (a finite non-empty set). An element of Σ is considered a data block. At the server interface, denoted S , the resource provides the entire history of accesses made by the clients (modeling the information leakage via a server log file in practice), and allows the server to overwrite existing data blocks. To syntactically separate the former capability (modeling data leakage), from the latter, capability (modeling active influence), we formally divide interface S into two sub-interfaces which we denote by S_H (for honest but curious) and S_I (for intrusion). The server can only overwrite data blocks if the resource is set into a special write mode. The distinguisher (or environment) is given the capability to adaptively enable and

³ Such a direct influence could be modeled in UC along the lines of [3] using an additional incorruptible party.

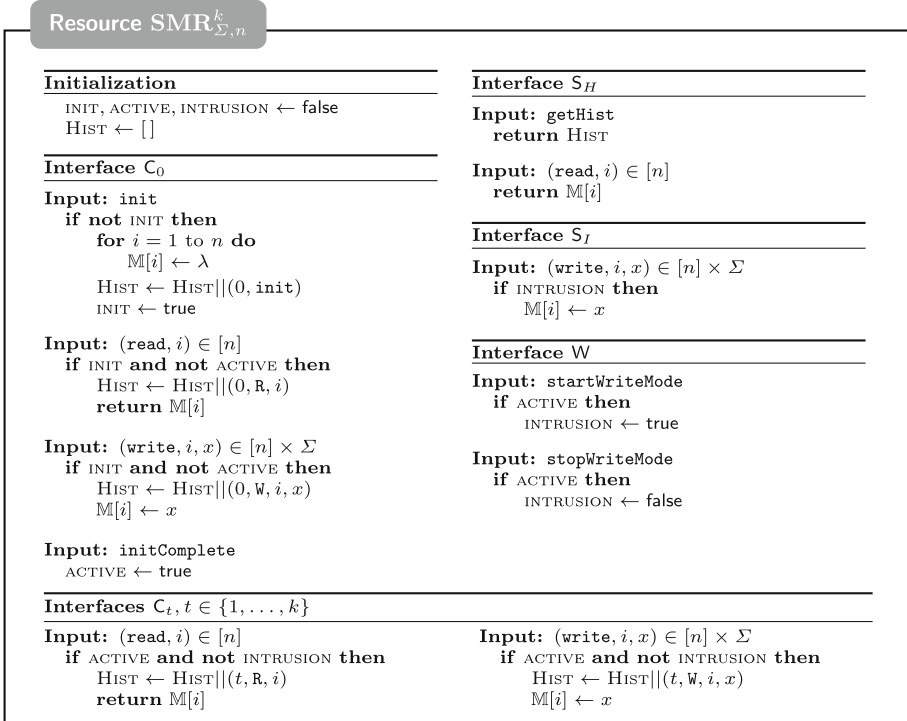


Fig. 3. Description of the insecure server-memory resource.

disable this write mode at the free interface W . The combination of capabilities at interfaces W and S_I allows our model to capture different types of adversarial influence, including adaptively setting return values of client read operations, or to model phases in which no server write access is possible at all. We present the basic server-memory resource, called $\text{SMR}_{\Sigma, n}^k$, in detail in Fig. 3. Our formalization is more general than the simple client-server setting in that it takes into account several clients that access the resource, each via their interface C_i . The parameters of the resource are the number of clients k , the alphabet Σ , and the number of blocks. The interface C_0 is the initialization interface and is used to set up the initial state of the resource (for example as a first step in the protocol). Only after the resource is initialized, indicated by the input `initComplete` at C_0 , the client interfaces become active and can update the state. We assume that (adversarial) server write operations only happen after the initialization is complete. Interface C_0 can be thought of as being assigned to a special party or simply to a dedicated client whose first actions are to initialize the resource. The basic server-memory resource constitutes the core element of our model and is general enough to serve as the building block for numerous applications in the realm of cloud storage. In this work, we are particularly interested in securing the storage against a malicious server.

4 Security Guarantees for Server-Memory Resources

In this section, we present server-memory resources that offer more security guarantees for the clients in that they restrict the capabilities of the server. The formal descriptions of these systems as pseudo-code is given in the full version of this work [4].

Authenticity and confidentiality. An *authentic* server-memory resource enhances the basic server-memory resource by restricting the capabilities at the active interface S_I . Instead of being capable to modify existing data blocks, the server can either delete data blocks, via input (`delete`, i) at S_I , or restore previously deleted data blocks, via input (`restore`, i) at S_I . A deleted data block is indicated by the special symbol ϵ . A client accessing the location of a deleted data block simply receives ϵ as an answer. The authentic server-memory resource is denoted by $\mathbf{aSMR}_{\Sigma,n}^k$. This memory resource can be realized from the insecure server-memory resource by standard techniques such as Blum’s protocol [5], which we formally show in the full version [4].

A further enhancement to this is defined by the *confidential and authentic* server-memory resource, denoted $\mathbf{cSMR}_{\Sigma,n}^k$. It enhances the authentic server-memory resource by restricting the access at the server interface S_H in that each server read operation simply returns $\lambda \in \Sigma$. Furthermore, the history of client accesses only reveal the location, but not the value that was read or written. In the full version of this work [4] we show that standard encryption can be used to construct this resource from authentic server memories.

Secure (oblivious) server-memory resource. The secure (and oblivious) server-memory resource is again a strengthening of the above and offers the strongest guarantees for the clients. First, the access pattern does not leak to the server apart from the number of accesses made. Second, the adversarial influence is now limited to setting a corruption or “pollution” parameter α . On each client read or write operation (`read`, i) or (`write`, i, x) the operation fails with probability α and the cell i is considered deleted. This expresses the inability of an intruder to mount a targeted attack on chosen blocks. His influence pollutes the entire memory in the specific way of increasing (or decreasing) the probability of a failure. In particular, our ideal functionality demands that each cell or block fails independently and with the same probability (if it had not failed before). Our concrete formulation of this resource, which we denote by $\mathbf{sSMR}_{\Sigma,n}^{k,t_{\text{rep}}}$, is slightly more general than just described: it is parameterized as before by the number of clients k , the alphabet Σ , the size n , and additionally by a tolerance t_{rep} (considered as the replication factor) that formalizes the resilience against failures. Intuitively, only after t_{rep} read or write operations for location i have failed, i is considered as deleted, which of course includes the standard case $t_{\text{rep}} = 1$. This guarantee, although quite strong, seems appealing in practice and is realizable as we prove in the next section. It further seems to be a desirable abstraction on its own, for example in the context of data replication where the assumption that blocks fail independently is crucial. It further allows for straightforward statistical predictions of this error parameter. One could imagine to weaken this resource by considering correlations among

failures, or to allow different cells to fail with different probabilities. We only consider the strongest variant in this paper and show how to achieve it.

4.1 Auditable Storage

For each server-memory resource one can naturally specify how an ideal storage audit should behave.

Basic, authenticated, and confidential auditable server memory. We observe that an audit can only provide a reasonable guarantees in a phase where an intruder is not active. In this case, the check reveals whether the current memory blocks are indeed the newest version that the client wrote to the storage. This is formally specified in Fig. 4 for the case of a basic server-memory resource. In particular, if a single data block has changed, the ideal audit will detect this and output an error to the client. It is obvious that in case of a successful audit, this guarantee only holds up to the point where the server gains write-access to the storage again, in which case a new audit has to reveal whether modifications have been made. The goal of a scheme providing a proof of storage is to implement this additional capability. In the full version [4], we give standard constructions that implement the audit capability for each of the presented server-memory resources.

Secure and auditable server memory. For the secure server-memory, it is more difficult to specify a realistic capability to capture secure audits. Due to the probabilistic nature of resource **sSMR**, the ideal retrievability guarantee for secure memory resources has to be a probabilistic one: based on an additional

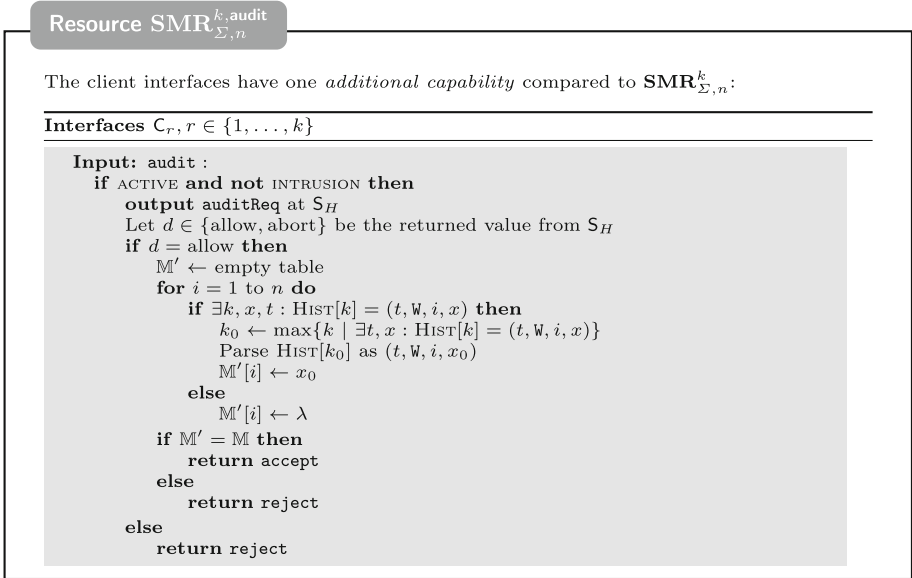


Fig. 4. Specification of an ideal audit guarantee for the basic memory.

parameter τ , the ideal audit of resource $\mathbf{sSMR}_{\Sigma, n}^{k, t_{\text{rep}}, \tau, \text{audit}}$ reveals whether the probability that the entire memory cannot be retrieved anymore is below the threshold τ . Stated differently, if an audit is successful, this means that the entire memory can be retrieved with probability at least $1 - \tau$. Hence, the smaller the parameter τ is, the stronger is the retrievability guarantee. We give the formal specification in the full version of this paper [4].

5 Implementing Secure and Robust Outsourced Storage

In this section, we show that our strongest notion is achievable. We first construct a secure server storage from an authentic and confidential server storage and make it auditable in a second step. We need to specify the protocol for the clients by means of a converter which every client attaches to its interface. We further have to provide a converter that describes the initialization step (generating cryptographic keys etc.) and which is attached at interface C_0 . To show that a protocol achieves a construction, we have to prove both conditions of Constructive Cryptography as explained in Sect. 2. The protocol uses a local memory \mathbf{L} shared among all clients.⁴ We assume that client accesses to the resources are sequential (which is trivially true in the single client setting). If this is not guaranteed, the clients could establish mutual exclusion by running Dekker’s or Peterson’s algorithm via \mathbf{L} .

Basic structure. We employ the Path ORAM protocol by Stefanov et al. [22] and complement it with a proper error handling such that the protocol realizes the secure server-memory resource from an authentic and confidential server-memory resource: The protocol maintains a tree structure on the assumed server-memory resource. For a logical memory with ℓ positions (assume ℓ is a power of two), the binary tree has height $L = \log(\ell)$ (and thus ℓ leaves). Each node N_r of the tree can hold Z memory blocks (where Z is a small constant greater or equal to 4 [22]). As usual, the tree is stored in the server memory in linear ordering from 1 to $2\ell - 1$, where in location 1 the root node N_1 is stored and where the leaves are located at addresses ℓ to $2\ell - 1$. We refer to the leaf node at address $\ell + i - 1$ as the i th leaf node. For such a leaf node, the unique path to the root of the tree is denoted $\mathcal{P}(i)$ and by $\mathcal{P}(i, lv)$ we denote the node at level lv on this path. The total number of blocks stored on the server is thus $Z \cdot (2\ell - 1)$.

The client stores a position map **position**, which is a table of size $L \cdot \ell$ bits and maps all logical addresses to the index of its associated leaf node. At any time during protocol execution, the invariant holds that for any logical address $i \in [\ell]$, if $\text{position}[i] = x$, then the correct data block (i, v) is contained in a node on the path $\mathcal{P}(x)$ or in the stash S . The stash is a local buffer maintained by the client that stores data blocks that overflow during the protocol execution. A data block overflows if all suitable nodes in the tree are already occupied by real memory blocks. The number of overflowing blocks is proven to be small in [22].

⁴ More specifically, at each interface C_i of \mathbf{L} , the usual read and write capabilities are available. The server does not have access to this resource.

Protocol. Initially, the tree is initialized to contain ℓ empty blocks of the form (i, λ) for each address $i \in [\ell]$. Upon initialization, the tree is built to contain these empty blocks. In addition, the position table and the stash are stored in the shared memory \mathbf{L} and to each address i , a uniformly random leaf node is assigned, i.e., $\text{position}[i] \leftarrow \{1, \dots, \ell\}$. Since each node of the tree should be a list of exactly Z elements, each node is complemented with the necessary amount of dummy elements which we encode as $(0, \lambda)$ (as opposed to real elements that contain the normal addresses and the associated data block). The entire tree is then written to the server storage. The formal description as pseudo-code of the converter init_{sec} that implements the above steps can be found in the full version of this paper.

To access a logical address i to either read or update the corresponding value v , the client reads the associated index of the leaf node $x \leftarrow \text{position}[i]$ and reassigns $\text{position}[i]$ to a new uniformly random leaf. Next, the client retrieves all nodes on the path $\mathcal{P}(x)$ from the server memory (from leaf to root) and all found real elements (j, v) ($j > 0$) are added to the stash. In case the value at position i is to be updated, it is assigned a new value at this point. Finally, the nodes of $\mathcal{P}(x)$ are newly built and written back to the server. In this write-back phase, as many blocks as possible from the local stash are “pushed” onto this path. To deal with failures on a read or write-access to a logical address i , the protocol behaves as follows: if during the above execution, a read request to the server is answered by ϵ , indicating that a node is deleted, then the logical address i is marked as invalid in the local position table $\text{position}[i] \leftarrow \epsilon$. To remain oblivious in this case, the protocol subsequently writes back all previously retrieved nodes without any modifications (yielding a sequence of dummy accesses). In a subsequent request to retrieve logical block i , the protocol will detect the invalid entry in the position table and just return ϵ . To remain oblivious, the protocol additionally reads a uniformly random path from the outsourced binary tree and subsequently rewrites the very same elements without modifications (again yielding a sequence of dummy accesses). If during these dummy accesses an error occurs, i.e., the server-memory resource returns ϵ upon a request, this is simply ignored.

This concludes the description of the protocol. A more precise specification as pseudo-code can be found in the full version of this work [4]. We denote this client converter by sec_{RW} . The security of the protocol is assured by the following theorem proven in the full version:

Theorem 1. *Let $k, \ell, Z \in \mathbb{N}$ and $\Sigma_1 := ((\{0\} \cup [\ell]) \times \Sigma)^Z$ for some finite non-empty set Σ . The above described protocol $\text{sec} := (\text{init}_{\text{sec}}, \text{sec}_{\text{RW}}, \dots, \text{sec}_{\text{RW}})$ (with k copies of sec_{RW}) constructs the secure server-memory resource $\text{sSMR}_{\Sigma, \ell}^{k, 1}$ from the confidential (and authentic) server-memory resource $\text{cSMR}_{\Sigma_1, 2\ell}^k$ and a local memory, with respect to the simulator sim_{sec} (described in the proof) and the pair $(\text{honSrv}, \text{honSrv})$. More specifically, for all distinguishers \mathbf{D}*

$$\begin{aligned} \Delta^{\mathbf{D}}(\text{honSrv}^{\text{S}} \text{sec}_{\mathcal{P}}[\mathbf{L}, \text{cSMR}_{\Sigma_1, 2\ell}^k], \text{honSrv}^{\text{S}} \text{sSMR}_{\Sigma, \ell}^{k, 1}) &= 0 \\ \text{and} \quad \Delta^{\mathbf{D}}(\text{sec}_{\mathcal{P}}[\mathbf{L}, \text{cSMR}_{\Sigma_1, 2\ell}^k], \text{sim}_{\text{sec}}^{\text{S}} \text{sSMR}_{\Sigma, \ell}^{k, 1}) &= 0. \end{aligned}$$

Improving the resilience by replication. There is a simple protocol that improves the resilience to losing data blocks. The protocol stores each data block t times within the secure server memory. Formally, this protocol constructs resource $\text{sSMR}_{\Sigma, \ell}^{k, t}$ from $\text{sSMR}_{\Sigma, t, \ell}^{k, 1}$. Recall that in the former resource, only failing to read (or write) a logical memory cell more than t times implies that the data block is not accessible any more. We provide more details in [4].

5.1 Do All ORAM Schemes Realize a Secure Server-Memory?

Our ideal system provides strong security guarantees. Especially, the failure probabilities are required to be independent and the same for each memory location. However, not all existing ORAM schemes satisfy this level of security: we show in [4] that in the recursive Path ORAM scheme by Stefanov et al. [21], failures among memory locations are correlated. This is dangerous in applications, where such errors can be observed because the error pattern and the access pattern are correlated. In a second case, we elaborate on the Goodrich-Mitzenmacher ORAM scheme [10], where we show that the failure probabilities are not the same for all (logical) memory locations. The problem in general with hierarchical structures of ORAM is that they allow to predict at which level in the hierarchy an element resides (recall that an element is an address-value pair). In the full version of this work [4], we describe a concrete scenario where this can be abused to lead to an error-pattern that correlates with the access pattern and hence provides a side-channel to the access pattern. The attack is structural and therefore applies also to stronger notions, for example ORAM schemes that satisfy the NRPH-property of [8].

5.2 Implementing Audits for Secure Server-Memory Resources

In this section, we show how to construct an auditable secure server-memory resource from a secure server-memory resource. We reduce the problem of auditing secure server-memory resources to the problem of estimating the corruption factor α . Each protocol chooses a tolerated threshold ρ and stores the data with replication factor t_{rep} that compensates data loss up to the corruption threshold ρ . To make sure that all values can be retrieved with a certain probability, the protocol tests t_{audit} fixed locations to estimate whether the parameter α has already reached the tolerated threshold ρ . In a first variant, the audit is successful if none of the probed locations return an error. In a second variant, we obtain similar results if the t_{audit} trials are used to obtain a sufficiently accurate estimate of α . The constructions are parameterized by the tolerated threshold ρ and by the desired retrievability guarantee τ . The values of t_{audit} and t_{rep} depend on both of these parameters. The dependency is roughly as follows: The stronger the desired retrievability guarantee should be, the higher the value of t_{rep} needs to be. However, the smaller the value of the tolerated threshold ρ is, the smaller the value of t_{rep} can be. On the other hand, a smaller value of the threshold ρ implies a higher value of t_{audit} . More specifically, the assumed resource is a secure server-memory resource with replication t_{rep} and size $\ell + t_{\text{audit}}/t_{\text{rep}}$ whose

values are determined in the theorem below. The desired resource is an auditable secure server-memory resource of size ℓ and with retrievability guarantee τ .

The protocol. As before, the protocol consists of an initialization converter `statInit`, the client converter `statAudit`, and an honest server behavior `statSrvAudit`. The server behavior is equal to the dummy behavior of the last section. So we only describe the protocol for the client. The protocol is parameterized by t_{audit} . For the sake of presentation, we do not explicitly write it as it is clear from the context. On input `init` to `statInit`, the converter calls `init` and sets `FLAG` \leftarrow 0. The variable `FLAG` records whether the protocol has ever detected an error when writing or reading to the server. If equal to one, it signals that misbehavior has been detected and will provoke subsequent audits to reject. The flag does not influence ordinary client read and write requests. On `(read, i)` to either `statInit` or `statAudit`, the converter outputs `(read, i)` to retrieve the value at memory location i or the error symbol ϵ , and outputs this returned value at its outer interface. In the case of an error, set `FLAG` \leftarrow 1. On `(write, i, v)` to either `statInit` or `statAudit`, the converter outputs `(write, i, v)` to write the value v at location i of the server. Again, if an error is observed, it sets `FLAG` \leftarrow 1. Finally, on input `audit` to converter `statAudit`, the converter immediately returns `reject` if `FLAG` = 1. If `FLAG` = 0 the audit is executed as follows:⁵, the converter issues t_{rep} read requests to each logical memory location $r = \ell + 1, \dots, \ell + \frac{t_{\text{audit}}}{t_{\text{rep}}}$. If and only if no read instruction returned the error symbol ϵ , then output `success`. Otherwise, the output is `reject` and the flag is updated to `FLAG` \leftarrow 1. The security of this scheme follows from the following theorem that is proven in [4].

Theorem 2. *Let Σ be an alphabet, let $\ell, \kappa, t_{\text{rep}}, t_{\text{audit}}, d \in \mathbb{N}$ such that $d = \frac{t_{\text{audit}}}{t_{\text{rep}}}$, and let $\rho, \tau \in (0, 1)$ such that*

$$t_{\text{rep}} > \frac{\log(\tau) - \log(\ell)}{\log(\rho)}, \quad t_{\text{audit}} > \frac{-\kappa}{\log(1 - \rho)}. \tag{1}$$

The above described protocol `statCheck` := (statInit, statAudit, . . . , statAudit) (with k copies of `statAudit`) parameterized by t_{audit} , constructs the auditable secure server-memory resource $\mathbf{sSMR}_{\Sigma, \ell}^{k, t, \tau, \text{audit}}$ from the secure server-memory resource $\mathbf{sSMR}_{\Sigma, \ell+d}^{k, t_{\text{rep}}}$ and a local memory (which stores the variable `FLAG`), with respect to the simulator `simstat` (described in the proof) and the pair $(\text{honSrv}, \text{honSrv})$. More specifically, for all distinguishers \mathbf{D} performing at most q audits,

$$\Delta^{\mathbf{D}}(\text{honSrv}^S \text{statCheck}_{\mathcal{P}}[\mathbf{L}, \mathbf{sSMR}_{\Sigma, \ell+d}^{k, t_{\text{rep}}}], \text{honSrv}^S \mathbf{sSMR}_{\Sigma, \ell}^{k, t_{\text{rep}}, \tau, \text{audit}}) = 0$$

and

$$\Delta^{\mathbf{D}}(\text{statCheck}_{\mathcal{P}}[\mathbf{L}, \mathbf{sSMR}_{\Sigma, \ell+d}^{k, t_{\text{rep}}}], \text{sim}_{\text{stat}}^S \mathbf{sSMR}_{\Sigma, \ell}^{k, t_{\text{rep}}, \tau, \text{audit}}) \leq q \cdot 2^{-\kappa}.$$

⁵ From a statistical point of view, if `FLAG` = 0, we have t_{audit} independent samples to estimate the parameter α .

We refer to the full version [4] for a numerical example of the above audit mechanism. In [4], we also give a second audit mechanism based on a direct estimation of the corruption factor α using Chernoff-Bounds.

6 Assessment of Hash-Based Challenge-Response Audits

Our model allows for a formal assessment of the security of a very simple and widely-used hash-based challenge-response protocol. To the best of our knowledge, this scheme lacks a formal security analysis. In a nutshell, during an audit, the server has to provide the correct hash value of the current storage content concatenated with a uniform random challenge provided (and precomputed) by the client. The expected security claim is that the server cannot have modified or deleted the content before answering the challenge. As we outline here, this intuition, although seemingly convincing, is not correct in general and the scheme requires much stronger assumptions in order to be provably secure. We consider the setting where one client stores a single file F (sequence of bits) on an insecure server memory and later audits this file once.

Assumed and constructed resource. We assume an (ideal) hash function, i.e., a random oracle, $H : \{0, 1\}^* \mapsto \{0, 1\}^r$. The random oracle is made available to the parties by means of a system \mathbf{H} that has an interface for the client and one for the server: On input (eval, x) at any of its interfaces \mathbf{H} returns $H(x)$ at the same interface. We further assume a small local storage and a bidirectional communication channel [7, 16] between client and server and denote it by \mathbf{Ch} . Last but not least, we assume an ordinary insecure memory resource $\mathbf{SMR}_{\Sigma, \ell + \kappa}^1$, where $\Sigma = \{0, 1\}$ and κ being the size of the challenge c (note that we assume more space here than simply ℓ : the client will at some point append the challenge to the file). The desired functionality we want to achieve is the auditable insecure memory resource $\mathbf{SMR}_{\Sigma, \ell}^{1, \text{audit}}$.

The protocol. As usual, we specify an initialization converter hashInit , a client converter hashAudit , and the protocol for the honest server behavior srvHash . On input init to hashInit , the converter simply calls init of its connected resource. On $(\text{write}, 1, F)$ to either hashInit or hashAudit , where F is an ℓ -bitstring, the converter writes F to the server storage. It then chooses a uniform random challenge $c \in \{0, 1\}^\kappa$ and computes $y \leftarrow H(F||c)$ and stores c and y in the local storage. On $(\text{read}, 1)$ to either hashInit or hashAudit , the converter retrieves the content of the memory and outputs the first ℓ bits of the received content. Finally, on a query (audit) to converter hashAudit , if there is a challenge stored in local memory, the protocol writes c to the server memory at locations $\ell + 1 \dots \ell + \kappa$ and sends a notification auditReq to the server via the bidirectional channel. On receiving a response y' on that channel from the server, the client protocol outputs success if and only if $y = y'$. In any case, the challenge c is deleted from the local storage. Last but not least, the server protocol srvHash , upon receiving an audit-request, simply evaluates H on the current memory contents and sends the result to the client.

The following lemma (formally proven in the full version) says that computing the correct hash does not imply that the data is stored in the memory resource. Hence, the protocol is in general provably not secure under this assumption.

Lemma 1. *Let $\ell, \ell', \kappa, r \in \mathbb{N}$, with $\ell' = \ell + \kappa$, let $\Sigma := \{0, 1\}$, and let \mathbf{H} be a random oracle (with one interface for the client and one for the server). Then, the challenge-response protocol, specified by the client converters `hashInit`, `hashAudit` and the server converter `srvHash`, does not provide a secure proof of storage: there is a distinguishing strategy such that for any simulator `sim` it holds that*

$$\Delta^{\mathbf{D}}(\text{hashInit}^{C_0} \text{hashAudit}^{C_1}[\mathbf{L}, \mathbf{Ch}, \mathbf{SMR}_{\Sigma, \ell'}^1, \mathbf{H}], \text{sim}^{\mathbf{S}} \mathbf{SMR}_{\Sigma^{\ell}, 1}^{1, \text{audit}}) = 1.$$

In the full version of this work, we give sufficient conditions for this protocol to be provably secure. We show that the additional assumption we have to make in order for the scheme to become sound, is to restrict adversarial random oracle evaluations to inputs from the server storage only, i.e., an adversarial query consists of two indices i and j ($i < j$) to obtain $\mathbf{H}(\mathbb{M}[i] \parallel \dots \parallel \mathbb{M}[j])$. However, whether such an assumption can be made in practice relies on trust into the server software: the assumption seems reasonable if we trust the server to work correctly except that in case of failures it is simply not willing to reveal this fact (due to loss of reputation). But the scheme does not protect against a fully cheating server as proven in the above lemma. Finally, we would like to point out that another drawback of the scheme is that structural properties of the hash-function could be abused. We give an in-depth explanation of this weakness and the corresponding formal claims in the full version of this work [4].

References

1. Apon, D., Katz, J., Shi, E., Thiruvengadam, A.: Verifiable oblivious storage. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 131–148. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54631-0_8
2. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Kissner, L., Peterson, Z.N.J., Song, D.X.: Provable data possession at untrusted stores. In: ACM Conference on Computer and Communications Security, pp. 598–609 (2007)
3. Ateniese, G., Dagdelen, Ö., Damgård, I., Venturi, D.: Entangled cloud storage. *Future Gener. Comput. Syst.* **62**, 104–118 (2016)
4. Badertscher, C., Maurer, U.: Composable and robust outsourced storage. *Cryptography ePrint Archive, Report 2017/133* (2017). <https://eprint.iacr.org/2017/133>. Full version of this paper
5. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. *Algorithmica* **12**(2/3), 225–244 (1994)
6. Camenisch, J., Enderlein, R.R., Maurer, U.: Memory erasability amplification. In: Zikas, V., De Prisco, R. (eds.) SCN 2016. LNCS, vol. 9841, pp. 104–125. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44618-9_6
7. Canetti, R.: Universally composable security: a new paradigm for cryptographic protocols. In: Proceedings of the 42nd Symposium on Foundations of Computer Science, pp. 136–145. IEEE (2001)

8. Cash, D., K upc u, A., Wichs, D.: Dynamic proofs of retrievability via oblivious RAM. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 279–295. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_17
9. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. *J. ACM (JACM)* **43**(3), 431–473 (1996)
10. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22012-8_46
11. Halevi, S., Harnik, D., Pinkas, B., Shulman-Peleg, A.: Proofs of ownership in remote storage systems. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 491–500. ACM (2011)
12. Hofheinz, D., Matt, C., Maurer, U.: Idealizing identity-based encryption. In: Iwata, T., Cheon, J.H. (eds.) ASIACRYPT 2015, Part I. LNCS, vol. 9452, pp. 495–520. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_21
13. Juels, A., Kaliski, B.S.: PORs: proofs of retrievability for large files. In: ACM Conference on Computer and Communications Security, pp. 584–597 (2007)
14. Keelveedhi, S., Bellare, M., Ristenpart, T.: DupLESS: server-aided encryption for deduplicated storage. Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 2013), pp. 179–194 (2013)
15. Maurer, U.: Indistinguishability of random systems. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 110–132. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46035-7_8
16. Maurer, U.: Constructive cryptography – a new paradigm for security definitions and proofs. In: M odersheim, S., Palamidessi, C. (eds.) TOSCA 2011. LNCS, vol. 6993, pp. 33–56. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-27375-9_3
17. Maurer, U., Renner, R.: Abstract cryptography. In: Innovations in Theoretical Computer Science, pp. 1–21 (2011)
18. Maurer, U., Renner, R.: From indifferentiability to constructive cryptography (and back). In: Hirt, M., Smith, A. (eds.) TCC 2016-B. LNCS, vol. 9985, pp. 3–24. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53641-4_1
19. Naor, M., Rothblum, G.N.: The complexity of online memory checking. *J. ACM* **56**(1), 2:1–2:46 (2009)
20. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: limitations of the indifferentiability framework. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20465-4_27
21. Stefanov, E., Shi, E., Song, D.X.: Towards practical oblivious RAM. In: 19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, 5–8 February 2012 (2012)
22. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 299–310. ACM (2013)