Composable and Robust Outsourced Storage[⋆]

Christian Badertscher and Ueli Maurer

Department of Computer Science, ETH Zurich, 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. (Eurocrpyt 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 investigate the (in)applicability of hash-function constructions in this setting.

[⋆] This is the full version of the article due to appear at CT-RSA 2018. The final publication will be available at link.springer.com.

Table of Contents

1 Introduction 3
1.1 Summary of our Results and Contributions of this Work 3
1.2 On the Importance of Composition and Robustness 6
1.3 Further Related Work 7
2 Preliminaries 9
2.1 Notation for Systems and Algorithms 9
2.2 Discrete Systems 9
2.3 Constructive Cryptography 10
2.4 Definitions of Cryptographic Primitives 11
3 Basic Server-Memory Resource 12
4 Security Guarantees for Server-Memory Resources 13
5 Constructions among Server-Memory Resources 16
5.1 Authentic Server-Memory Resources from Basic Server-Memory Resources . . . 16
5.2 Confidential from Authentic Server-Memory Resources 22
5.3 Secure from Confidential Server-Memory Resources 24
5.4 Do all ORAM Schemes realize a Secure Server-Memory Resource? 31
6 Auditable Server-Memory Resources 33
7 Constructing Auditable Server-Memory Resources 34
7.1 Making Authentic Server-Memory Resources Auditable 34
7.2 Making Secure Server-Memory Resources Auditable 37
7.3 Revisiting the Hash-Based Challenge-Response Approach 39
References 42
A Further Details of Section 1 46
A.1 Traditional PoR Game 46
B Further Details of Section 2 46
B.1 Discrete Resources and the World Interface 46

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 would like 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) [36,4]. 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 [37,34]. 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 an outsourcing scheme in modular steps that provably achieves stronger security than existing protocols.

We formulate our model in the language of the constructive cryptography framework (CC) [43,44]. Our results are not specific to the CC framework itself and choosing another definitional framework like Canetti's Universal Composition (UC) framework [17] would yield closely related findings [35]. 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

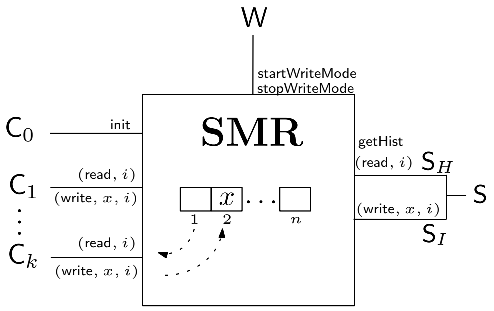

Fig. 1. The basic server-memory resource.

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 lets 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. We consider this line of research in a future work.

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 attacker yields more general statements, as it also allows us to capture scenarios where the server does indeed not have the active choice to do so, but where any external event can provoke the server memory to be corrupted.

We present more secure variants of the basic server memory. In particular, in Sect. 4, we introduce the following resources:

- 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.

We show how to construct each of these resources in Sect. 5. We then present in Sect. 6 the *auditable* versions of the above resources. An auditable server memory additionally gives the client the capability to check whether the memory has been modified or deleted, without the need to read the entire memory. We again give protocols that achieve auditable server memories in Sect. 7.

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 [2,19]. We give a protocol that realizes the secure server-memory resource and is an extension of the well-known Path ORAM protocol [57].

We also show that the existing definitions for access-pattern hiding 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 [33]), or failures among memory locations are correlated (as in [56]) and explain why this is problematic in practice.

A novel notion for audit schemes. The *auditable* server memory resources are server-memory resources with the additional client-side capability of being able to ask 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. Our new definition stipulates that a protocol implements a proof of retrievability if it realizes an auditiable server-memory resource from an ordinary one by possibly using additional resources like a random oracle.

To the best of our knowledge, this is the first composable security definition for audit schemes. We thereby rectify two drawbacks of existing definitions. First, our definition and protocols guarantee that the client can download the data, if (a) the audit succeeds and (b) the adversary does not corrupt further locations after the audit. This guarantee is not required by existing definitions and not fulfilled by certain schemes such as [19]. Second, existing definitions are based on the concept of knowledge-extractors: The extractor needs the client secrets and the server strategy to recover the data. As outlined in more detail in Sect. 1.3, this is not a scenario which is suitable in practice, since the client and the server would not reveal this information to each other or a third party. Our formulation does not use extractors.

For each of our server-memory resources, we show how to implement secure audits. 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 protocol resembles the protocol by Cash et al.[19], 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 again needed to ensure that the data can be retrieved after a successful audit. For example, encountering just a small number of failures after an audit must not harm this guarantee.

A formal analysis of hash-based challenge-response audits. A composable formalization of storage audits in the spirit of indifferentiability and constructive cryptography [45, Section 7] has been envisioned in [51] 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

[51], 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 [51]. This further implies that replacing the random oracle by any provably secure iterated hash-function construction like NMAC [22,8], does not have to yield secure audits. In particular, there is no contradiction to the composition theorem as claimed in [51]. Note that Demay et al. [23] provide another way to resolve this contradictory situation of [51].[1] 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.

A provably secure and robust outsourcing scheme. We show how to construct an authentic server-memory resource from a basic (and thus insecure) server-memory resource in Sect. 5.1. We subsequently show how to get a confidential and authentic server-memory resource from an authentic one in Sect. 5.2, and finally achieve a secure server-memory resource from a confidential and authentic one in Sect. 5.3. Finally, in Sect. 7.2, we realize an audit mechanism as outlined above on top of the secure server-memory resources. By combining all of the above steps, the composition theorem guarantees us that the composed protocol achieves an auditable, secure server-memory resource from a basic server-memory resource (and local memory). The composed protocol is thus an efficient outsourcing scheme that provably achieves strictly stronger security than existing protocols in this realm. The protocol 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.

Development of more efficient, provably secure schemes. Beyond the fact that modular steps are often simpler to analyze than an entire protocol, our modular 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) 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.

1.2 On the Importance of Composition and Robustness

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 [32] on software protection. There is, however, a subtle and fundamental difference between the context of outsourced storage and the context of software protection of [32] 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 [32] 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

[1] They prove that any simulator in the construction of a random oracle from ideal compression functions needs to maintain an internal state linear to the total size of all queries. This implies that the server cannot save space due to the simulator's memory consumption.

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. 5.4.

Example 2: Implementing secure audits. Let us focus on a protocol by Cash et al. [19] 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. The audit protocol in Sect. 7.2 is of this type.

1.3 Further Related Work

Models for outsourced storage. The security of services outsourced to untrusted third-parties has received much attention in the literature and is a prevalent topic in cloud computing. Regarding the special case of untrusted outsourced storage, Mazieres and Shasha [46] and later works by Androulaki et al. [1] and Cachin et al. [15,15,13,14,11,12] formalize untrusted storage as read-write registers, where multiple clients can write to and read from the (shared) register by issuing *read and write requests*. Each request and its answer is considered an event and the view of the client is a sequence of such invocation and response events. Integrity is then defined in a property-based way as conditions on the view of the client. This model is appealing as it is expressive and the interface is simple. Our model keeps this simple structure. At each client interface a read or write request can be input. The response by the server is defined as the current value of the location (or register) which is susceptible to adversarial write operations which means that the responses may be adaptively chosen by the attacker. Our model can be seen as an universally composable variant of the above (and where a set of cooperating clients try to obtain strong security guarantees). The benefits are stronger security guarantees and that our server-memory functionalities make the adversarial influence explicit. This allows to compare different functionalities according to their strength. It is also desirable to have a composable security definition for outsourced storage since they are naturally part of larger systems, such as distributed file systems [55] or storage systems [54,40].

Composable notions in the realm of secure outsourced storage are unfortunately still rare. Recent examples include the works by Atteniese et al. [5], Camenisch et al.[16], Liu et al. [41], or Apon et al.[2] that illustrate the importance of filling these gaps. For example, in [5] the authors formalize the security guarantees of entangled storage in the UC framework. In order for their scheme to be secure, they work in the $\mathcal{F}_{mem}$-hybrid model, where the functionality $\mathcal{F}_{mem}$ is a memory functionality that models a server storage, where clients can upload and retrieve their values. $\mathcal{F}_{mem}$ is a quite strong assumption. For example, the adversary does not see the values uploaded to the storage. Hence, $\mathcal{F}_{mem}$ can be seen as a special case of our server memory functionality where the adversarial access is limited. Such a functionality could be obtained, for example, by defining a "wrapper" functionality along the lines of [28] that wraps an ordinary memory resource to restrict the adversarial access. This makes the assumptions on the capabilities of the attacker again explicit in order to compare different protocols.

In [16], the authors investigate the security of protocols that improve the leakage resilience of imperfectly erasable memory and use the constructive cryptography framework. Imperfect erasures leak certain information to a passive adversary even after the client instructed the memory to delete the contents. This setting is fundamentally different from ours in that we are interested in security guarantees against active attacks on untrusted server storages and erasability of (local) memory is not considered. The resources in [16] also make use of a free interface (also denoted world interface) to assign certain capabilities to the environment instead of the adversary to achieve general and meaningful security statements.

ORAM. Oblivious RAM is a cryptographic primitive originally introduced by Goldreich and Ostrovsky [32] and has become a standard approach to hiding the access pattern when accessing a cloud storage. A sequence of fundamental results have led to important security and performance improvements such as [33,19,39,24,49,30,21,57,31,47,58,27,56]. Previous works define the security of ORAM schemes by requiring that different sequences of client read and write operations lead to indistinguishable sequences of accesses to the server storage. Protection against active adversaries is typically achieved by detecting malicious behavior, for example by using Merkle-Trees or authenticators [56,50,2], and aborting upon detection [32,19].

PoR. The first formal security models for proofs of retrievability (PoR) were given by Juels and Kaliski [36] and in a similar spirit also in previous works, for example by Naor et al. [48] on *sublinear authentication*. The definition in [36] is tailored to the problem of storing static data on a server, i.e., a large file like a backup that is unlikely to be changed frequently. Roughly, an important key idea of many PoR schemes is that a redundant encoding of the file makes sure that any too small data loss is tolerable and thus need not be detected. On the other hand, by downloading some file locations at random and performing some integrity checks, a client can detect a significant amount of data loss. Subsequent publications [10,25,53] present new and more efficient schemes as well as generalized adversarial models. In these works, the initial definition has been further carried over to the case of dynamic data. One major obstacle in the case of dynamic data is to force the server not to discard updates to (possibly a small number of) memory locations. Cash et al. [19] propose a scheme based on ORAM and showed that hiding the access structure of reads and writes to the server allows for efficient PoR for dynamic data. Another solution, proposed by Chandra et al. [20], shows that the problem of constructing locally decodable and locally updatable codes is strongly related to the construction of PoR schemes. Roughly, being able to update an encoding of a file F to an encoding of F' by only changing a small number of file blocks allows to reduce the problem of dynamic PoR to static PoR. Using a related guiding idea, Shi et al. [53] recently proposed an (efficient) PoR scheme for dynamic data as well. Another closely related research branch deals with models and applications for provable data possession (PDP), proposed by Ateniese et al. [4,6,3]. PDP and PoR are related in spirit, but PDP is essentially a weaker definition than PoR. While the goal of proofs of retrievability is to guarantee that a file remains retrievable in full, the goal of PDP is to test if most of the file is still retrievable. Subsequent models have been proposed that deal with dynamic data, for which several protocols have been designed [6,38,26]. The security definitions of PoR (and PDP) schemes are extractor-based and are usually formalized as a game between a challenger and an adversary. A PoR scheme consists of four interactive sub-protocols executed between a stateful client and a stateful server: $\mathsf{init}(1^\nu, \Sigma, n)$, $\mathsf{read}(i)$, $\mathsf{write}(i, v_i)$ carry their usual intended meaning, where ν is the security parameter, Σ is the alphabet and n is the size of the memory. The fourth protocol, audit, is executed to verify if the server possesses all the client's data, in which case the client returns accept.

The PoR security definition is threefold and consists of correctness, authenticity and retrievability. In this paragraph, we focus on the third security property and on the dynamic PoR game ExtGame as found in [19,20] which we state in Fig. 18 in Appendix A for completeness. For more details and variations, we refer to [36,52,19,20]. A scheme is said to provide retrievability, if there exists an efficient probabilistic extractor $\mathcal{E}$,such that, for every efficient server $\bar{S}$, every polynomial $p(\nu)$ it holds that $\mathsf{Pr}\left[\mathsf{ExtGame}_{\bar{S},\mathcal{E}}(\nu, p(\nu)) = 1\right]$ is negligible in the security parameter. Intuitively, the game formalizes that from a cheating server strategy that successfully passes an audit with good probability, it is possible to extract the correct storage content that the client uploaded. In general, the extractor is provided with the client private keys and the server strategy. The concept of a knowledge-extractor emerged from proofs of knowledge: The reasoning in proofs of knowledge is that if a dishonest prover passes the test with good probability using some arbitrary strategy, then this strategy could be used by the prover himself to effectively calculate the witness by virtue of the extractor algorithm, which is an algorithm that extracts the witness by executing and rewinding the prover's strategy.

Although the extractor-based approach to PoR includes a meaningful thought-experiment, it has a major drawback concerning client-side security guarantees: If an audit is successful, the availability of the data, which is of major concern to the client, is only guaranteed through the execution of the extractor, which needs to access the server strategy and the secret state of the client. Both parties are unlikely to disclose this sensitive information. No server would reveal its entire state and no client would reveal its secret keys.

2 Preliminaries

2.1 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, $x \twoheadleftarrow \mathcal{D}$ denotes sampling x according to $\mathcal{D}$. For a finite set X, $x \twoheadleftarrow X$ 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., $(\texttt{write}, i, v)$ to write the value v at location i of a storage). We ignore keywords in writing the domains of arguments, e.g., $(\texttt{write}, i, v) \in [n] \times \Sigma$ 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.

2.2 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 [42], 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

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

where $\mathsf{Pr}\,[\mathbf{DR} = 1]$ 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.

2.3 Constructive Cryptography

The central object in constructive cryptography is that of a resource available to parties, and the resources we discuss in this work are modeled by reactive discrete systems. As in general the same resource may be accessible to multiple parties, such as a communication channel that allows a sender to input a message and a receiver to read it, we assign inputs to certain *interfaces* that correspond to the parties: the sender's interface allows to input a message to the channel, and the receiver's interface allows to read what is in the channel. More generally, a resource is a discrete system with a finite set of interfaces $\mathcal{I}$ via which the resource interacts with its environment.

Converters model protocols used by parties and can attach to an interface of a resource to change the inputs and outputs at that interface. This composition, which for a converter π, interface I, and resource $\mathbf{R}$ is denoted by $\pi^I\mathbf{R}$, again yields a resource. In this work, a converter π is modeled as a systems with two interfaces: the *inner interface* $\texttt{in}$ and the outer interface $\texttt{out}$. The inner interface can be connected to an interface I of a resource $\mathbf{R}$ and the outer interface then becomes the new interface I of resource $\pi^I\mathbf{R}$. For a vector of converters $\pi = (\pi_{I_1}, \dots, \pi_{I_n})$ with $I_i \in \mathcal{I}$, and a subset of interfaces $\mathcal{P} \subseteq \{I_1, \dots, I_n\}$, $\pi_{\mathcal{P}}\mathbf{R}$ denotes the resource where π_I is connected to interface I of $\mathbf{R}$ for every $I \in \mathcal{P}$. For $\mathcal{I}$-resources $\mathbf{R}_1, \dots \mathbf{R}_m$ the *parallel composition* $[\mathbf{R}_1, \dots, \mathbf{R}_m]$ is again an $\mathcal{I}$-resource that provides at each interface access to the corresponding interfaces of all subsystems.

In this paper, we make statements about resources with interface sets of the form $\mathcal{I} = \mathcal{P} \cup \{\mathsf{S}, \mathsf{W}\}$, where $\mathcal{P} := \{\mathsf{C}_0, \dots, \mathsf{C}_k\}$. $\mathcal{P}$ is the set of honest (client) interfaces. A *protocol* is a vector $\pi = (\pi_{I_1}, \dots, \pi_{I_{|\mathcal{P}|}})$ that specifies one converter for each interface $I \in \mathcal{P}$. Intuitively, $\mathcal{P}$ can be thought of as the interfaces that honestly apply the specified protocol π. On the other hand, S is the potentially dishonest (server) interface and is assigned a converter that describes the default behavior at that interface in case it is honest. Intuitively, interface S is the interface where the prescribed behavior is not necessarily applied. Dishonest behavior is thus captured by replacing the protocol by an arbitrary, adversarial strategy. The interface W is the free interface, also denoted to as the world interface, and models the direct influence of a distinguisher on a resource. This interface is particularly useful to model capabilities which are not a priori assigned to a party or the attacker and hence allows more general statements. We refer the interested reader to Appendix B.1 for a more detailed exposition.

A constructive security definition then specifies the goal of a protocol in terms of *assumed* and *constructed* resources. We directly state the central definition of a construction of [43] and briefly explain the relevant conditions.

Definition 1. *Let* $\mathbf{R}$ *and* $\mathbf{S}$ *be resources with interface set* $\mathcal{I} = \mathcal{P} \cup \{\mathsf{S}, \mathsf{W}\}$ *with* $\mathcal{P} := \{\mathsf{C}_0, \ldots, \mathsf{C}_k\}$. *Let* ε *be a function that maps distinguishers to a value in* $[0, 1]$ *and let* sim *be a converter (the simulator). Let* $\pi = (\pi_{\mathsf{C}_0}, \ldots, \pi_{\mathsf{C}_k})$ *be a protocol and let* $\mathsf{srv}_{\mathbf{R}}$ *and* $\mathsf{srv}_{\mathbf{S}}$ *be converters that describe the default behavior at interface* S *when it is honest. The protocol* π *constructs resource* $\mathbf{S}$ *from resource* $\mathbf{R}$ *(with potentially dishonest* S*) within* ε *and with respect to the simulator* sim *and the pair* $(\mathsf{srv}_{\mathbf{R}}, \mathsf{srv}_{\mathbf{S}})$*, if for all distinguishers* $\mathbf{D}$*,*

$$\Delta^{\mathbf{D}}(\mathsf{srv}_{\mathbf{R}}{}^{\mathsf{S}} \pi_{\mathcal{P}} \mathbf{R}, \mathsf{srv}_{\mathbf{S}}{}^{\mathsf{S}}\, \mathbf{S}) \leq \varepsilon(\mathbf{D}) \qquad \textit{(Correctness)}$$
$$\Delta^{\mathbf{D}}(\pi_{\mathcal{P}} \mathbf{R},\ \mathsf{sim}^{\mathsf{S}}\, \mathbf{S}) \leq \varepsilon(\mathbf{D}). \qquad \textit{(Security)}$$

The first condition ensures that the protocol implements the required functionality if the server is honest. For example, for outsourced server-memory resources, all values written to the storage have to be retrievable unmodified when no attacker is present. This honest server behavior is modeled by a “dummy” converter that does not interfere with the client protocol and does not generate any output at its outer interface. The second 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. The notion of construction is composable, which intuitively means that the constructed resource can be replaced in any context by the assumed resource with the protocol attached without affecting the security. We refer to [43] for a proof. For readers more familiar with Canetti’s UC Framework [17], we refer to [35] for explanations of how the above concepts relate to similar concepts in UC.

2.4 Definitions of Cryptographic Primitives

IND-CPA secure encryption schemes. A private key cryptosystem $\mathsf{ENC} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ for message space $\mathcal{M}$, key space $\mathcal{K}$, and ciphertext space $\mathcal{C}$ consists of a (probabilistic) key generation algorithm $\mathcal{G}$ that returns a key sk, a (probabilistic) encryption algorithm $\mathcal{E}$ that given a message $m \in \mathcal{M}$ and the private key sk returns a ciphertext $c \leftarrow \mathcal{E}_{sk}(m)$, and a (possibly probabilistic) decryption algorithm $\mathcal{D}$, that given a ciphertext $c \in \mathcal{C}$ and the private key sk returns a message $m \in \mathcal{M}$. The correctness condition demands that $\mathcal{D}_{sk}(\mathcal{E}_{sk}(m)) = m$ for all keys sk in the support of $\mathcal{G}$.

A private key cryptosystem has indistinguishable ciphertexts under a chosen plaintext attack if the following two systems $\mathbf{G}_0^{\mathsf{CPA}}$ and $\mathbf{G}_1^{\mathsf{CPA}}$, defined with respect to ENC, are indistinguishable for efficient distinguishers $\mathbf{D}$: Both systems start by choosing a key $sk \in \mathcal{K}$ according to algorithm $\mathcal{G}$. Next, they act as encryption oracle by accepting messages $m \in \mathcal{M}$ and providing the corresponding ciphertexts $\mathcal{E}_{sk}(m) \in \mathcal{C}$. At any point, a distinguisher $\mathbf{D}$ can submit a *challenge query* that consists of two messages $\tilde{m}_0$ and $\tilde{m}_1$ (with $|\tilde{m}_0| = \tilde{m}_1|$) where the system $\mathbf{G}_i^{\mathsf{CPA}}$ returns the encryption of $\tilde{m}_i$. The security condition requires that no efficient distinguisher $\mathbf{D}$ can distinguish the two systems better than with negligible advantage.

Message authentication codes. We consider a MAC function f with message space $\mathcal{M}$, tag space $\mathcal{T}$, and key space $\mathcal{K}$ (with an associated distribution). The security condition for a MAC function f states that no efficient adversary $\mathbf{A}$ can win the following game $\mathbf{G}^{\mathsf{MAC}}$ (defined with respect to f) better than with negligible probability. $\mathbf{G}^{\mathsf{MAC}}$ first chooses a key $sk \twoheadleftarrow \mathcal{K}$. Then it acts as a signing oracle, receiving messages $m \in \mathcal{M}$ at its interface and responding with $f_{sk}(m)$. At any point, $\mathbf{A}$ can undertake a forging attempt by providing a message m' and a tag t' to $\mathbf{G}^{\mathsf{MAC}}$. The game is won if and only if $f_{sk}(m') = t'$ and m' was never queried before by $\mathbf{A}$. The probability of adversary $\mathbf{A}$ in winning the game is succinctly written as $\Gamma^{\mathbf{A}}(\mathbf{G}^{\mathsf{MAC}})$.

Digital signature schemes. A *digital signature scheme* $\mathsf{SIG} = (K, S, V)$ for a message space $\mathcal{M}$ and signature space Ω consists of a (probabilistic) key generation algorithm K that returns a key pair (sk, vk), a (possibly probabilistic) signing algorithm S, that given a message $m \in \mathcal{M}$ and the signing key sk returns a signature $s \leftarrow S_{sk}(m)$, and a (possibly probabilistic, but usually deterministic) verification algorithm V, that given a message $m \in \mathcal{M}$, a candidate signature $s' \in \Omega$, and the verification key vk returns a bit $V_{vk}(m, s')$. The bit 1 is interpreted as a successful verification and 0 as a failed verification. It is required that $V_{vk}(m, S_{sk}(m)) = 1$ for all m and all (vk, sk) in the support of K. A digital signatures scheme is existentially unforgeable under chosen message attacks if no efficient adversary $\mathbf{A}$ can win the following game $\mathbf{G}^{\mathsf{EU\text{-}CMA}}$ (defined with respect to SIG) better than with negligible probability. $\mathbf{G}^{\mathsf{EU\text{-}CMA}}$ first chooses a key pair $(sk, vk) \twoheadleftarrow K$. Then it acts as a signing oracle, receiving messages $m \in \mathcal{M}$ at its interface and responding with $S_{sk}(m)$. At any point, $\mathbf{A}$ can undertake a forging attempt by providing a message m' and a candidate signature s' to $\mathbf{G}^{\mathsf{EU\text{-}CMA}}$. The game is won if and only if $V_{vk}(m', s') = 1$ and m' was never queried before by $\mathbf{A}$. The probability of adversary $\mathbf{A}$ in winning the game is succinctly written as $\Gamma^{\mathbf{A}}(\mathbf{G}^{\mathsf{EU\text{-}CMA}})$.

Erasure codes. An (n, k, d) erasure code over the alphabet Σ with error symbol $\perp \notin \Sigma$, is a pair of (efficient) algorithms $(\mathsf{enc}, \mathsf{dec})$ that satisfy the following requirement: For all $F \in \Sigma^k$, let $\bar{F} := \mathsf{enc}(F) \in \Sigma^n$ and define the set

$$\mathcal{C}_{\bar{F}} := \{\bar{F}' \in (\Sigma \cup \{\perp\})^n \mid \forall i : \bar{F}'_i \in \{\bar{F}_i, \perp\} \wedge \text{at most } d{-}1 \text{ positions of } \bar{F}' \text{ are equal to } \perp\}.$$

Then, for all $\bar{F}' \in \mathcal{C}_{\bar{F}}$, it holds that $\mathsf{dec}(\bar{F}') = F$.

3 Basic Server-Memory Resource

Our 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 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 $\mathbf{SMR}^{k}_{\Sigma,n}$, in detail in Fig. 2.

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

```
Resource $\mathrm{SMR}^k_{\Sigma,n}$

Initialization
  INIT, ACTIVE, INTRUSION $\leftarrow$ false
  HIST $\leftarrow$ [ ]

Interface $\mathsf{C}_0$
Input: init
  if not INIT then
      for $i = 1$ to $n$ do
          $\mathbb{M}[i] \leftarrow \lambda$
      HIST $\leftarrow$ HIST$||(0, \mathtt{init})$
      INIT $\leftarrow$ true

Input: $(\mathtt{read}, i) \in [n]$
  if INIT and not ACTIVE then
      HIST $\leftarrow$ HIST$||(0, \mathtt{R}, i)$
      return $\mathbb{M}[i]$

Input: $(\mathtt{write}, i, x) \in [n] \times \Sigma$
  if INIT and not ACTIVE then
      HIST $\leftarrow$ HIST$||(0, \mathtt{W}, i, x)$
      $\mathbb{M}[i] \leftarrow x$

Input: initComplete
  ACTIVE $\leftarrow$ true

Interface $\mathsf{S}_H$
Input: getHist
  return HIST

Input: $(\mathtt{read}, i) \in [n]$
  return $\mathbb{M}[i]$

Interface $\mathsf{S}_I$
Input: $(\mathtt{write}, i, x) \in [n] \times \Sigma$
  if INTRUSION then
      $\mathbb{M}[i] \leftarrow x$

Interface W
Input: startWriteMode
  if ACTIVE then
      INTRUSION $\leftarrow$ true

Input: stopWriteMode
  if ACTIVE then
      INTRUSION $\leftarrow$ false

Interfaces $\mathsf{C}_t, t \in \{1, \ldots, k\}$
Input: $(\mathtt{read}, i) \in [n]$
  if ACTIVE and not INTRUSION then
      HIST $\leftarrow$ HIST$||(t, \mathtt{R}, i)$
      return $\mathbb{M}[i]$

Input: $(\mathtt{write}, i, x) \in [n] \times \Sigma$
  if ACTIVE and not INTRUSION then
      HIST $\leftarrow$ HIST$||(t, \mathtt{W}, i, x)$
      $\mathbb{M}[i] \leftarrow x$
```

Fig. 2. Description of the insecure server-memory resource.

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. Also note that for the sake of simplicity we only formalize in our specifications client requests that occur when the server-write mode is turned off. The complementary case simply follows along the lines of [2]: In case of a currently active intrusion, we would just let the adversary define the return value (i.e., define the cell content before the answer to the client is generated). All our protocols are easily seen to be secure with respect to this model: if a client access to a cell happened during an active intrusion, then this can be mapped to our model by (1) inserting an additional active intrusion step before the client access in which the distinguisher can define all return values (via interface S_I), (2) de-activating the server write mode and execute the client access, (3) activate the server write mode again.

The basic server-memory resource constitutes the core element of our model and serves as the fundamental building block for numerous applications in the realm of cloud storage as discussed in the previous sections. In this work, we elaborate along the lines of securing the memory resource against a malicious server. Other possible directions include the formalization of distributed file-systems, proof of ownership, access control mechanisms, or entangled storage that are part of future and ongoing work and not covered in this paper.

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.

Authentic server-memory resource. 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

Resource $\mathbf{aSMR}^{k}_{\Sigma,n}$

The definition of **aSMR** is identical to **SMR** *except* for the adversarial influence at interface S_I and the reaction on writing to a corrupted memory location:

Interfaces $\mathsf{C}_t, t \in \{1, \dots, k\}$

```
Input: $(\texttt{read}, i) \in [n]$
  if ACTIVE and not INTRUSION then
      $\textsc{Hist} \leftarrow \textsc{Hist}||(t, \texttt{R}, i)$
      return $\mathbb{M}[i]$

Input: $(\texttt{write}, i, x) \in [n] \times \Sigma$
  if ACTIVE and not INTRUSION then
      if $\mathbb{M}[i] \neq \epsilon$ then
          $\textsc{Hist} \leftarrow \textsc{Hist}||(t, \texttt{W}, i, x)$
          $\mathbb{M}[i] \leftarrow x$
      else
          $\textsc{Hist} \leftarrow \textsc{Hist}||(t, \texttt{Fail}, i, x)$
          return $\epsilon$
```

Interface S_I

```
Input: $(\texttt{delete}, i) \in [n]$
  if INTRUSION then
      $\mathbb{M}[i] \leftarrow \epsilon$

Input: $(\texttt{restore}, i) \in [n]$
  if INTRUSION then
    if $\exists k, x, t : \textsc{Hist}[k] = (t, \texttt{W}, i, x)$ then
        $k_0 \leftarrow \max\{k \mid \exists t, x : \textsc{Hist}[k] = (t, \texttt{W}, i, x)\}$
        Parse $\textsc{Hist}[k_0]$ as $(t, \texttt{W}, i, x_0)$
        $\mathbb{M}[i] \leftarrow x_0$
    else
        $\mathbb{M}[i] \leftarrow \lambda$
```

Fig. 3. The authentic server-memory resource (only differences to **SMR** shown).

Resource $\mathbf{cSMR}^{k}_{\Sigma,n}$

The definition of **cSMR** is identical to **SMR** *except* for the information the server and the adversary learn about the stored data:

Interface S_H

```
Input: getHist
  $\textsc{Hist}' \leftarrow [\,]$
  for $j = 1$ to $|\textsc{Hist}|$ do
      $q \leftarrow \textsc{Hist}[j]$
      if $q = (t, \texttt{W}, i, x)$ for some $t, x, i$ then
          $\textsc{Hist}' \leftarrow \textsc{Hist}'||(t, \texttt{W}, i)$
      if $q = (t, \texttt{Fail}, i, x)$ for some $t, x, i$ then
          $\textsc{Hist}' \leftarrow \textsc{Hist}'||(t, \texttt{Fail}, i)$
      if $q = (t, \texttt{R}, i)$ for some $t, i$ then
          $\textsc{Hist}' \leftarrow \textsc{Hist}'||(t, \texttt{R}, i)$
  return $\textsc{Hist}'$

Input: $(\texttt{read}, i) \in [n]$
  return $\lambda$
```

Interface S_I

```
Input: $(\texttt{delete}, i) \in [n]$
  if INTRUSION then
      $\mathbb{M}[i] \leftarrow \epsilon$

Input: $(\texttt{restore}, i) \in [n]$
  if INTRUSION then
    if $\exists k, x, t : \textsc{Hist}[k] = (t, \texttt{W}, i, x)$ then
        $k_0 \leftarrow \max\{k \mid \exists t, x : \textsc{Hist}[k] = (t, \texttt{W}, i, x)\}$
        Parse $\textsc{Hist}[k_0]$ as $(t, \texttt{W}, i, x_0)$
        $\mathbb{M}[i] \leftarrow x_0$
    else
        $\mathbb{M}[i] \leftarrow \lambda$
```

Fig. 4. The authentic and confidential server-memory resource (only differences to **SMR** shown).

input $(\texttt{delete}, i)$ at S_I, or restore previously deleted data blocks, via input $(\texttt{restore}, i)$ 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. We formally describe the authentic server-memory resource $\mathbf{aSMR}^{k}_{\Sigma,n}$ in Fig. 3.

Confidential server-memory resource. The confidential and authentic server-memory resource, denoted $\mathbf{cSMR}^{k}_{\Sigma,n}$, is formally specified in Fig. 4. 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.

Secure (oblivious) server-memory resource. We present the secure (and oblivious) server-memory resource in Fig. 5. This resource 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"

Resource $\mathrm{sSMR}_{\Sigma,n}^{k,t_{\mathrm{rep}}}$

```
Initialization
  INIT, ACTIVE, INTRUSION ← false
  $\alpha \leftarrow 0$; HIST ← [ ]
  $c_i \leftarrow 1$ for all $i \in [\ell]$
```

```
Interface $\mathsf{C}_0$
Input: init
  if not INIT then
     for $i = 1$ to $n$ do
        for $j = 1$ to $t_{\mathrm{rep}}$ do
           $\mathbb{M}[i, j] \leftarrow \lambda$
     HIST ← HIST$||(0, \texttt{init})$
     INIT ← true

Input: $(\texttt{read}, i) \in [n]$
  if INIT and not ACTIVE then
     HIST ← HIST$||(0, \texttt{R}, i)$
     return $\mathbb{M}[i, 1]$

Input: $(\texttt{write}, i, x) \in [n] \times \Sigma$
  if INIT and not ACTIVE then
     for $j = 1$ to $t_{\mathrm{rep}}$ do
        HIST ← HIST$||(0, \texttt{W}, i, x)$
        $\mathbb{M}[i, j] \leftarrow x$

Input: initComplete
  ACTIVE ← true
```

```
Interface $\mathsf{S}_H$
Input: getHist
  HIST′ ← [ ]
  for $j = 1$ to |HIST| do
     $q \leftarrow$ HIST$[j]$
     if $q \in \{(t, \texttt{W}, i, x), (t, \texttt{R}, i)\}$ then
        HIST′ ← HIST′$||(t, \texttt{Access})$
     else
        HIST′ ← HIST′$||q$
  return HIST′

Input: $(\texttt{read}, i) \in [n]$
  return $\lambda$
```

```
Interface $\mathsf{S}_I$
Input: $(\texttt{pollute}, \rho) \in [0, 1]$
  if INTRUSION then
     $\alpha \leftarrow \rho$

Input: $(\texttt{reducePollution}, \delta) \in [0, \alpha]$
  if INTRUSION then
     $\alpha \leftarrow \alpha - \delta$
```

```
Interface W
Input: startWriteMode
  if ACTIVE then
     INTRUSION ← true

Input: stopWriteMode
  if ACTIVE then
     INTRUSION ← false
```

```
Interfaces $\mathsf{C}_t, t \in \{1, \ldots, k\}$
Input: $(\texttt{read}, i) \in [n]$
  if ACTIVE and not INTRUSION then
     if $\mathbb{M}[i, c_i] \neq \epsilon$ then
        $Z \twoheadleftarrow \mathsf{Bernoulli}(\alpha)$
        if $Z = 0$ then
           HIST ← HIST$||(t, \texttt{R}, i)$
           return $\mathbb{M}[i, c_i]$
        else
           HIST ← HIST$||(t, \texttt{Failed})$
           $\mathbb{M}[i, c_i] \leftarrow \epsilon$
           if $c_i < t_{\mathrm{rep}}$ then
              $c_i \leftarrow c_i + 1$
           return $\epsilon$
     else
        $Z \twoheadleftarrow \mathsf{Bernoulli}(\alpha)$
        if $Z = 0$ then
           HIST ← HIST$||(t, \texttt{Access})$
        else
           HIST ← HIST$||(t, \texttt{Failed})$
        if $c_i < t_{\mathrm{rep}}$ then
           $c_i \leftarrow c_i + 1$
        return $\epsilon$

Input: $(\texttt{write}, i, x) \in [n] \times \Sigma$
  if ACTIVE and not INTRUSION then
     RET$_j$ ← ok for $j = 1 \ldots t_{\mathrm{rep}}$
     for $j = 1$ to $t_{\mathrm{rep}}$ do
        if $\mathbb{M}[i, j] \neq \epsilon$ then
           $Z \twoheadleftarrow \mathsf{Bernoulli}(\alpha)$
           if $Z = 0$ then
              HIST ← HIST$||(t, \texttt{W}, i, x)$
              $\mathbb{M}[i, j] \leftarrow x$
           else
              HIST ← HIST$||(t, \texttt{Failed})$
              $\mathbb{M}[i, j] \leftarrow \epsilon$
              RET$_j$ ← $\epsilon$
        else
           RET$_j$ ← $\epsilon$
           $Z \twoheadleftarrow \mathsf{Bernoulli}(\alpha)$
           if $Z = 0$ then
              HIST ← HIST$||(t, \texttt{Access})$
           else
              HIST ← HIST$||(t, \texttt{Failed})$
     return (RET$_1, \ldots,$ RET$_{t_{\mathrm{rep}}}$)
```

Fig. 5. Description of the secure server-memory resource.

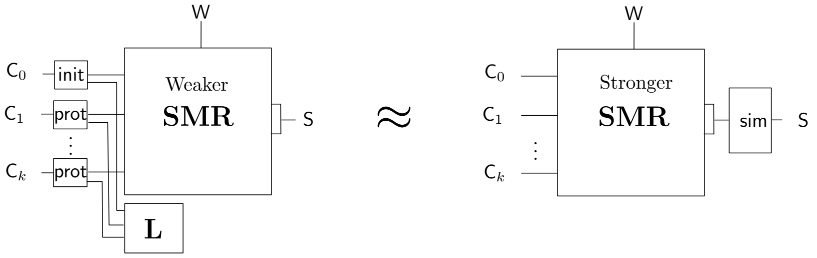

Fig. 6. Illustration of the security condition of constructions among server-memory resources.

parameter α. On each client read or write operation $(\mathtt{read}, i)$ or $(\mathtt{write}, i, x)$ 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 formulation of this resource, which we denote by $\mathbf{sSMR}_{\Sigma,n}^{k,t_{\mathrm{rep}}}$ and describe in Fig. 5, 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 $t_{\mathrm{rep}} = 1$. This guarantee, although quite strong, seems appealing in practice and is realizable as we prove in the next section.[2]

It further seems to be a desriable 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.

5 Constructions among Server-Memory Resources

In this section, we show how to construct stronger server-memory resources from weaker ones. For each construction, 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 Definition 1. In this section, the default behavior of the potentially dishonest server is specified by the dummy converter honSrv that does not answer any query at its outer interface and does not give any input to the server-memory resource.

The protocols we present make use of a local memory $\mathbf{L}$ shared among all clients. 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. An illustration is shown in Fig. 6. 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 using the shared memory $\mathbf{L}$.

[2] One could again complement the specification with the behavior upon a client request during an active intrusion phase. As in [2], one could let the adversary decide on the success or failure of a client request in that case. More technically, following the general approach to map such cases into our model, we would simply allow the adversary to adjust the corruption parameter α in Fig. 5 during such a request before calling the sampling function. The security of the protocols is not be affected by such a change.

5.1 Authentic Server-Memory Resources from Basic Server-Memory Resources

Following Blum et al. [9], we build a tree structure on top of the outsourced data blocks to protect their authenticity (and freshness). Assume the size of the memory is ℓ, then the tree is a binary search tree with ℓ leaves, where each leaf corresponds to a data block. For simplicity, and without loss of generality, we assume that ℓ is a power of 2. In [9], each leaf is associated with a timestamp indicating the number of times the block was updated. The timestamp of an internal node is defined as the sum of the timestamps of its two children. We refer to this condition on the timestamps as the *tree invariant.* The timestamp of the root of the tree corresponds to total number of times the client has accessed the server-memory resource and is stored in a reliable local memory.

Protocol and notation. We denote the full binary tree for a memory of size ℓ as $T^{(\ell)}$. The tree has $2\ell - 1$ nodes which are mapped to the linear storage of **SMR** (of size $2\ell - 1$) and a local reliable memory **L** as follows[3]: The leaf node at location $\ell+i-1$ of **SMR** stores the value x_i of the (logical) memory. Internal nodes only contain a timestamp (and an authentication tag) and are stored in **SMR**. We denote the node at location r of **SMR** by N_r (for $r > 0$) and denote the root as N_0. For a node N_r, we denote by t_r its timestamp, and for a leaf node we additionally denote by $x_i \in \Sigma$ its associated data block ($i = r - \ell + 1$). To bind the contents of a node to its actual location on the server, we make use of a MAC function $f_{sk}(\cdot)$. The root node N_0 is not stored in **SMR** but on a reliable, local memory **L**. In summary, the format of the nodes are as follows:

$$N_r = \begin{cases} (x_i, t_r, f_{sk}(r, x_i, t_r)) & \textbf{if } r \geq \ell \quad (i = r - \ell + 1) \\ (t_r, f_{sk}(r, t_r)) & \textbf{if } r < \ell \\ t_0 & \textbf{if } r = 0. \end{cases}$$

To read a value x_i of the (logical) memory, the client retrieves all nodes on the path from the root to the leaf node $N_{\ell+i-1}$ and all their children. This is sometimes denoted to as the siblings path from the root to that leaf. We denote this sub-tree (consisting of $2\log\ell$) nodes by $T_{\ell+i-1}$ to make the index i appear explicitly. On each access to the logical data block i, the client verifies all authentication tags and checks the invariant in $T_{\ell+i-1}$, i.e., that for each node N_r, the timestamp is the sum of its children's timestamps, i.e., that $t_r = t_{2r+1} + t_{2r+2}$. If all checks succeed the tree $T_{\ell+i-1}$ is said to be *valid.* To write a new value to a leaf node $N_{\ell+i-1}$, one first retrieves $T_{\ell+i-1}$ and verifies that it is valid. Then, one updates the value of the leaf, its timestamp and the authentication tag and subsequently updates the timestamps and the authentication tags of all nodes on the path to the root to restore the invariant of tree $T_{\ell+i-1}$. Finally, all nodes are written back to their original location. It is straightforward to cast this protocol as a converter for the clients, denoted by $\mathsf{auth}_{\mathrm{RW}}$ and specified in Fig. 8. The generation of cryptographic keys and the initial setup of the tree are formally specified in the initialization converter $\mathsf{init}_{\mathrm{auth}}$ found in Fig. 7.

Intuitively, this protocol is secure since no adversary can inject a new value at any memory location, as each node is bound to a memory location on the server. Additionally, replaying an older value is not possible as an older value has a smaller timestamp and if the client verifies the tree's invariant, its reliably stored value N_0 is too large. Formally, we prove the following theorem.

Theorem 1. *Let $k, \ell \in \mathbb{N}$ and let $\Sigma_1 = \Sigma \times Z_q \times \mathcal{T}$ for some alphabet Σ. The protocol* $\mathsf{auth} := (\mathsf{init}_{\mathrm{auth}}, \mathsf{auth}_{\mathrm{RW}}, \ldots, \mathsf{auth}_{\mathrm{RW}})$ *(with k copies of* $\mathsf{auth}_{\mathrm{RW}}$*) described above based on a MAC function f with tag space $\mathcal{T}$ constructs the authentic server memory* $\mathbf{aSMR}^{k}_{\Sigma,\ell}$ *from*

[3] Note that the number of storage locations of **SMR** is about two times as large as the logical locations we want to protect. This, however, does not imply a storage overhead of a factor of two in practice, since the information stored in internal nodes is only a timestamp and not an entire data block.

```
Converter $\mathsf{init}_{\mathrm{auth}}$

Interface out
Input: init :
  $sk \twoheadleftarrow \mathcal{K}$; $N_0 \leftarrow 0$;
  output $(\mathtt{write}, 1, N_0)$ at in to $\mathbf{L}$
  output $(\mathtt{write}, 2, sk)$ at in to $\mathbf{L}$
  output init at in to $\mathbf{SMR}$
  for $i = 2$ to $\ell - 1$ do
      $N_i \leftarrow (0, f_{sk}(i, 0))$
      output $(\mathtt{write}, i, N_i)$ at in to $\mathbf{SMR}$
  for $i = \ell$ to $2\ell - 1$ do
      $N_i \leftarrow (\lambda, 0, f_{sk}(i, \lambda, 0))$
      output $(\mathtt{write}, i, N_i)$ at in to $\mathbf{SMR}$

Input: $(\mathtt{read}, i)$ or $(\mathtt{write}, i, x)$:
  Defined the same as for $\mathsf{auth}_{\mathrm{RW}}$ in Fig. 8

Input: initComplete :
  output initComplete at in to $\mathbf{SMR}$
```

Fig. 7. The initialization protocol for the realization of an authentic server memory.

the basic server memory $\mathbf{SMR}^k_{\Sigma_1,2\ell}$ *and a local memory* $\mathbf{L}$ *(of constant size), with respect to the simulator* $\mathsf{sim}_{\mathrm{auth}}$ *as defined in Fig. 9 and the pair* $(\mathsf{honSrv}, \mathsf{honSrv})$. *More specifically, we construct a reduction* $\mathbf{C}$ *such that for all distinguishers* $\mathbf{D}$,

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\ \mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,2\ell}], \mathsf{honSrv}^{\mathsf{S}}\mathbf{aSMR}^k_{\Sigma,\ell}) = 0$$
$$\text{and} \qquad \Delta^{\mathbf{D}}(\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,2\ell}], \mathsf{sim}^{\mathsf{S}}_{\mathrm{auth}}\mathbf{aSMR}^k_{\Sigma,\ell}) \leq \Gamma^{\mathbf{DC}}(\mathbf{G}^{\mathsf{MAC}}).$$

Proof. The correctness condition is obvious and we only give a proof of the security condition. We analyze the input-output behavior of both systems involved. To this end, we consider the possible inputs at each interface.

On input init, initComplete at interface C_0**:** Upon the `init`-query, the protocol $\mathsf{init}_{\mathrm{auth}}$ of the real system $\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,2\ell}]$ generates a secret MAC key sk and initializes the basic memory resource. Subsequently, the protocol writes the nodes of $T^{(\ell)}$ to the server memory, except for the root N_0 which is stored in the local memory. This initialization adds $2\ell - 2$ entries to the history $\textsc{Hist}$ of $\mathbf{SMR}$. After this initialization phase, $\textsc{Hist}$ reads $(0, \mathtt{init})||(0, \mathtt{W}, 1, N_1)|| \ldots ||(0, \mathtt{W}, 2\ell - 1, N_{2\ell-1})$. Any subsequent `read` query will return the fixed value $\lambda \in \Sigma$.
In the ideal system $\mathsf{sim}^{\mathsf{S}}_{\mathrm{auth}}\mathbf{aSMR}^k_{\Sigma,\ell}$, the query initializes the memory to the fixed value $\lambda \in \Sigma$ and adds the entry $(0, \mathtt{init})$ to $\textsc{Hist}$. Since this is the first entry of $\textsc{Hist}$, the simulator will replace this entry by its locally simulated list L_{init} that consists of the $2\ell - 2$ entries as above (where the key MAC key sk is chosen locally by the simulator).
Finally, on input `initComplete`, both systems deactivate interface C_0 and the other client interfaces are operational from this point onwards.

On input $(\mathtt{read}, i)$ **at interface** C_k**:** On this query, both protocols $\mathsf{init}_{\mathrm{auth}}$ (in case $k = 0$) and $\mathsf{auth}_{\mathrm{RW}}$ (in case $k > 0$) retrieve all the nodes of the sub-tree $T_{\ell+i-1}$. This sub-tree contains the leaf node $N_{\ell+i-1} = (x_i, t, f_{sk}((\ell + i - 1, t, x_i)))$ of $T(\ell)$ which stores the value of memory location i. Furthermore, $T_{\ell+i-1}$ consists of all nodes on the path from that leaf to the root together with their children. To retrieve this sub-tree, the protocol issues $2 \log \ell - 1$ `read`-queries. The history $\textsc{Hist}$ hence is increased by the list of $2 \log \ell - 1$ value $(k, \mathtt{R}, r_1)||...||(k, \mathtt{R}, r_{2\ell-1})$, where the indices r_j are ordered in decreasing order according to their location in $\mathbf{SMR}$. Afterwards, the protocol checks the validity of each authentication tag and checks the tree's invariant (i.e., that the sum of the children's timestamps is equal to the parent's timestamp). If all checks succeed, x_i is output (which is the last value written to this location), and otherwise ϵ is output.
In system $\mathsf{sim}^{\mathsf{S}}_{\mathrm{auth}}\mathbf{aSMR}^k_{\Sigma,\ell}$, system $\mathbf{aSMR}$ answers the query with the current memory content of cell i. If the cell is not corrupted, the last value written is output at interface

```
Converter auth_RW

Interface out
Input: (read, $i$)
  $T \leftarrow$ GetValidSubtree($\ell + i - 1$)
  if $T \neq \emptyset$ then
      Parse node $N_{\ell+i-1}$ (of tree $T$) as $(x, t, f_{sk}(\ell + i - 1, x, t))$
      return $x$
  else
      return $\epsilon$

Input: (write, $i, x$)
  $T \leftarrow$ GetValidSubtree($\ell + i - 1$)
  if $T \neq \emptyset$ then
      $N_{\ell+i-1} \leftarrow (x, t_{\ell+i-1} + 1, f_{sk}(\ell + i - 1, x, t_{\ell+i-1} + 1))$
      for each internal node $N_i \in T$ do
          $N_i \leftarrow (t_i + 1, f_{sk}(i, t_i + 1))$
      for each node $N_i \in T$ with $i > 0$ do
          output (write, $i, N_i$) at in to SMR
      $N_0 \leftarrow N_0 + 1$
      output (write, $1, N_0$) at in to L
  else
      return $\epsilon$

  function GetValidSubtree($i$)
      Retrieve $T_{\ell+i-1}$, i.e., all nodes $N_r$ on the path from leaf node $N_{\ell+i-1}$
      to the root $N_0$ (and their siblings) in descending order of index via:
          -output (read, 1) at in to L to read $N_0$
          -output (read, $r$) at in to SMR to read $N_r$
      for each node $N_r \in T_{\ell+i-1}$ with $r > 0$ do
        Case $N_r = (t_r, tag)$:
          if $f_{sk}(r, t_r) \neq tag$ then
              return $\emptyset$
        Case $N_r = (x, t_r, tag)$:
          if $f_{sk}(r, x, t_r) \neq tag$ then
              return $\emptyset$
      for each node $N_r \in T_{\ell+i-1}$ with $r > 0$ do
          par $\leftarrow \lceil \frac{i}{2} \rceil - 1$                ▷ index of parent
          ch$_1 \leftarrow 2 \cdot$ par $+ 1$                     ▷ index of 1st child
          ch$_2 \leftarrow 2 \cdot$ par $+ 2$                     ▷ index of 2nd child
          if $t_{\text{par}} \neq t_{\text{ch1}} + t_{\text{ch2}}$ then
              return $\emptyset$
      return $T_{\ell+i-1}$
```

Fig. 8. The converter for the clients to realize an authentic server memory from a basic server memory.

C_k. If the cell is corrupted, ϵ is output. In order for the simulator $\mathsf{sim}_{\text{auth}}$ to emulate this view, it internally simulates the tree T^ℓ and, after each adversarial query at interfaces S_I and S_H keeps track of which sub-trees $T_{\ell+r-1}$ are valid, and if not, corrupts the cell r of **aSMR** (cf. behavior at interface S_H and S_I below). This enforces the consistency between successful reads and valid subtrees exactly as in the real system. Additionally, the next time the simulator is activated, it will update its simulated history L accordingly: if this read-request $(k, \mathtt{R}, i)$ is the qth entry in Hist of **aSMR**, then, in procedure UpdateLog, this qth entry will lead to the increase of $2\log\ell - 1$ read-query entries in L. Hence, the history L is increased by the list of $2\log\ell - 1$ value $(k, \mathtt{R}, r_1)||...||(k, \mathtt{R}, r_{2\ell-1})$, where the indices r_j are ordered in decreasing order according to their location in the simulator's emulated tree. This perfectly mimics the real world behavior.

On input $(\mathtt{write}, i, x)$ **at interface** C_k**:** Each write request can be divided into two two phases: (1) the retrieval of sub-tree $T_{\ell+i-1}$ and its validity check and (2) writing the updated sub-tree $T_{\ell+i-1}$ back to the server memory (except for the root N_0) in case the validity check was passed. Phase (1) is simulated as before by $\mathsf{sim}_{\text{auth}}$ and thus we focus on the second phase.

In the real system $\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^k_{\Sigma_1,2\ell}]$, the tree $T_{\ell+i-1}$ is updated locally and each node is subsequently written back to **SMR**. The history Hist of the resource is thus increased by the following list of $2\log\ell - 1$ values, namely $(k, \mathtt{W}, r_1, N_{r_1})||...||(k, \mathtt{W}, r_{2\ell-1}, N_{2\ell-1})$.

In the ideal system, the procedure UpdateLog will replace this entry in the history by the appropriate sequence of write-requests only if the write-request was successful.

```
Converter sim_auth

Initialization
  $sk \twoheadleftarrow \mathcal{K}$
  Initialize $2\ell - 1$ nodes $N_i$ as in Fig. 7
  $L_{init} \leftarrow (0, \texttt{init})||(0, \texttt{W}, 1, N_1)|| \ldots ||(0, \texttt{W}, 2\ell - 1, N_{2\ell-1})$
  $pos \leftarrow 1$

Interface $\mathsf{S}_H$

Input: getHist :
  UpdateLog
  return $L$

Input: $(\texttt{read}, r) \in [2\ell - 1]$ :
  UpdateLog
  return $N_r$

Interface $\mathsf{S}_I$   (intrusion = true)

Input: $(\texttt{write}, r, x) \in [2\ell - 1] \times (\Sigma \times Z_n \times \mathcal{T})$
  UpdateLog
  Determine the last entry in $L$ that wrote value $N$ in to location $i$
  if $N \neq x$ then
      $N_r \leftarrow x$
      for each leaf node $N_{\ell+i-1}$ whose sub-tree $T_{\ell+i-1}$ is not valid do
          output $(\texttt{delete}, i)$ at in to aSMR
  else
      for each leaf node $N_{\ell+i-1}$ whose sub-tree $T_{\ell+i-1}$ is valid do
          output $(\texttt{restore}, i)$ at in to aSMR
      $N_r \leftarrow N$

  procedure UpdateLog
      output getHist at in to aSMR
      Let Hist be the returned value
      for $j = pos$ to $|\textsc{Hist}|$ do
          if $\textsc{Hist}[j] = (0, \texttt{init})$ then
              $L \leftarrow L_{init}$
          else if $\textsc{Hist}[j] = (k, \texttt{R}, i)$ then
              Let $T_{\ell+i-1}$ be the sub-tree of simulated leaf node $N_{\ell+i-1}$
              for each node $N_r \in T_{\ell+i-1}$ in decreasing order of index do
                  $L \leftarrow L||(k, \texttt{R}, r)$
          else if $\textsc{Hist}[j] = (k, \texttt{W}, i, x)$ then        ▷ Successful write operation
              Let $T_{\ell+i-1}$ be the (valid) sub-tree of simulated leaf node $N_{\ell+i-1}$
              for each node $N_r \in T_{\ell+i-1}$ in decreasing order of index do
                  $L \leftarrow L||(k, \texttt{R}, r)$
              for each node $N_r \in T_{\ell+i-1}$ in decreasing order of index do
                  Update $N_r$ according to Fig. 8
                  $L \leftarrow L||(k, \texttt{W}, r, N_r)$
          else if $\textsc{Hist}[j] = (k, \texttt{Fail}, i, x)$ then        ▷ Failed write operation
              Let $T_{\ell+i-1}$ be the sub-tree of simulated leaf node $N_{\ell+i-1}$
              for each node $N_r \in T_{\ell+i-1}$ in decreasing order of index do
                  $L \leftarrow L||(k, \texttt{R}, r)$
      $pos \leftarrow |\textsc{Hist}| + 1$
```

Fig. 9. The simulator for the construction of an authenticated memory.

Note that the simulator $\mathsf{sim}_{\text{auth}}$ is informed whether a client write resulted in a successful update (in which case $\textsc{Hist}[q] = (k, \mathtt{W}, i, x)$) or whether the update failed (in which case $\textsc{Hist}[q] = (k, \mathtt{Fail}, i, x)$).

On input getHist at interface S_H: In the real system, the output is the entire history of **SMR**. By the above analysis, a straightforward inductive argument shows that in system $\mathsf{sim}^{\mathsf{S}}_{\text{auth}}\mathbf{aSMR}^{k}_{\Sigma,\ell}$, the simulator's simulated history L, which is output upon this query, emulates the real-world view perfectly.

On input $(\mathtt{write}, r, x)$ at interface S_I: An adversarial write request in the real world is a simple replacement of the memory cell r of **SMR**. If the value x corresponds to the last honest value written to this cell, then this operation might provoke that now certain sub-trees $T_{\ell+i-1}$ become valid again (and hence be involved in successful read and write requests).

This is simulated in the ideal world in that simulator $\mathsf{sim}_{\text{auth}}$ checks, for $i = 0$ to $\ell - 1$, whether any sub-tree $T_{\ell+i-1}$ in its simulated server memory became valid again and issues $(\mathtt{restore}, i)$ to **aSMR** in this case.

In the other case, if the value x is unequal to the value N_r being replaced, this might lead to a couple of corrupted (logical) memory cells since certain sub-trees become invalid in the real system $\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^{k}_{\Sigma_1,2\ell}]$.

In the ideal world $\mathsf{sim}^{\mathsf{S}}_{\text{auth}}\mathbf{aSMR}^{k}_{\Sigma,\ell}$, the simulator $\mathsf{sim}_{\text{auth}}$ first updates its internal storage up to the current point by invoking $\textsc{UpdateLog}$ to get the actual value of N_r. If $N_r \neq x$, $\mathsf{sim}_{\text{auth}}$ issues a $(\mathtt{delete}, i)$-query to **aSMR** for each location i whose tree $T_{\ell+i-1}$ got invalid due to this update.

This update, however, only simulates the real world perfectly if the change $N_r \leftarrow x$ led to an invalid sub-tree for the case $N_r \neq x$. This is the case, if the authentication of x is invalid or if timestamps do not satisfy the invariant. The bad event, denoted by BAD occurs if the adversary manages to write a value x, that has never been written to location r and which nevertheless results in a valid sub-tree. We different two cases:

1. If location r stores a leaf node, this implies that x has the format (v', t'_r, tag') for which $t'_r > t_r$ *or* $(v' \neq v$ *and* $t'_r \geq t_r)$ holds, since for $t'_r < t'_r$ a valid sub-tree gets invalid since the overall invariant cannot longer hold (since the root value N_0 would be too large.) Hence, tag' corresponds to a valid forgery for the message (r, v', t_r) of $f_{sk}(.)$.
2. If location r stores an internal node, this implies that x has the format (t'_r, tag) with $t'_r > t_r$ (as otherwise it would constitute a reduction of the sum of the timestamps which will then eventually be smaller than N_0). In this case, tag corresponds to a forgery for the message (r, t'_r) of $f_{sk}(.)$.

Hence, we conclude that the real and ideal system are identical until event BAD occurs.

On input $(\mathtt{read}, r)$ at interface S_H: In the real system, this query returns the current value at location r of **SMR**. This is either the last value written by any client interface or the last value written by the adversary. In the ideal system, the simulator updates its internal simulation of the server memory on each activation and hence, returns either the last value written to r according to its simulated history L or the value that was written by an adversarial write.

On inputs startWriteMode and stopWriteMode at interface W: First, in the real system $\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^{k}_{\Sigma_1,2\ell}]$, the first input allows the adversary to access and modify the server storage until the input stopWriteMode is input. The same holds for the the ideal system $\mathsf{sim}^{\mathsf{S}}_{\text{auth}}\mathbf{aSMR}^{k}_{\Sigma,\ell}$, since the simulator does not react on adversarial queries at interface S_I in case $\textsc{intrusion} = \mathsf{false}$ and is allowed to access interface S_I of resource **aSMR** if and only if $\textsc{intrusion}$ is set.

This concludes the analysis of the behavior. We see that the real system system and the ideal system are identical until event BAD occurs. In particular, the occurrence of BAD implies a successful forgery against the MAC function $f_{sk}(.)$. We now design the straightforward reduction from a distinguisher $\mathbf{D}$ to an adversary $\mathbf{A} := \mathbf{DC}$ against $\mathbf{G}^{\mathsf{MAC}}$. $\mathbf{C}$ simulates the real system, but evaluates the MAC-function using oracle queries to the game $\mathbf{G}^{\mathsf{MAC}}$. If $\mathbf{D}$ issues a write-query at interface S_I that provokes event BAD, $\mathbf{C}$ issues this value as a forgery to $\mathbf{G}^{\mathsf{MAC}}$. Hence, we can conclude the proof by noting that

$$\Delta^{\mathbf{D}}(\mathsf{auth}_{\mathcal{P}}[\mathbf{L}, \mathbf{SMR}^{k}_{\Sigma_1,2\ell}], \mathsf{sim}^{\mathsf{S}}_{\mathrm{auth}}\mathbf{aSMR}^{k}_{\Sigma,\ell}) \\ \leq \mathsf{Pr}^{\mathbf{D}(\mathsf{auth}_{\mathcal{P}}[\mathbf{L},\mathbf{SMR}^{k}_{\Sigma_1,2\ell}])}[\mathsf{BAD}] \leq \Gamma^{\mathbf{DC}}(\mathbf{G}^{\mathsf{MAC}}).$$

□

Replacing the MAC by a digital signature scheme

An alternative way to message authentication codes are digital signatures. They offer a way to relax the security requirements on the local storage, as the public key does not have to remain private. Furthermore, using a digital signature scheme allows to technically separate write access from read access. A party can write to the memory if and only if it possesses the secret key. Everyone possessing the public key can read the values authentically. Looking ahead, this further implies that the audit scheme of Sect. 7.1, if based on digital signatures, is an audit scheme that an external party can execute. Such schemes are known as publicly verifiable proofs of storage.

The protocol. We replace the MAC function f_{sk} in the protocol converters $\mathsf{init}_{\mathrm{auth}}$ and $\mathsf{auth}_{\mathrm{RW}}$ by a digital signature scheme (K, S, V) and denote the new protocol converters as $\mathsf{init}^{sig}_{\mathrm{auth}}$ and $\mathsf{auth}^{sig}_{\mathrm{RW}}$, respectively. These protocols are essentially identical to the protocols in Fig. 7 and Fig. 8 except for the obvious changes: In particular, the initialization protocol $\mathsf{init}^{sig}_{\mathrm{auth}}$ is defined as $\mathsf{init}_{\mathrm{auth}}$ except that the instructions $sk \leftarrow \mathcal{K}$ and $(\texttt{write}, 1, sk)$ to generate and store the secret key for the MAC function are replaced by the generation of a key pair $(sk, vk) \leftarrow K$ and storing the secret key sk in $\mathbf{L}$ and storing the public key in $\mathbf{L}'$, where $\mathbf{L}'$ is a local memory which has an additional for the server to read the contents. Furthermore, for both converters $\mathsf{init}^{sig}_{\mathrm{auth}}$ and $\mathsf{auth}^{sig}_{\mathrm{RW}}$, the format of the nodes takes the following form:

$$N_r = \begin{cases} (x_i, t_r, S_{sk}((r, x_i, tr))) & \textbf{if } r \geq \ell \quad (i = r - \ell + 1) \\ (t_r, S_{sk}((r, t_r))) & \textbf{if } r < \ell \\ t_0 & \textbf{if } r = 0. \end{cases}$$

Similarly, verification of an authentication tag tag of a node is accomplished by evaluating $V_{vk}((r, x_i, t_r), tag)$ (for a leaf) and $V_{vk}((r, t_r), tag)$ (for an internal node), respectively.

Theorem 2. *Let $k, \ell \in \mathbb{N}$ and let $\Sigma_1 = \Sigma \times Z_q \times \mathcal{T}$ for some finite (alphabet) set Σ. The protocol $\mathsf{auth}^{sig} := (\mathsf{init}^{sig}_{\mathrm{auth}}, \mathsf{auth}^{sig}_{\mathrm{RW}}, \ldots, \mathsf{auth}^{sig}_{\mathrm{RW}})$ described above based on a digital signature scheme (K, S, V) with signature space $\mathcal{T}$ constructs the authentic server-memory resource $\mathbf{aSMR}^{k}_{\Sigma,\ell}$ from the basic server-memory resource $\mathbf{SMR}^{k}_{\Sigma_1,2\ell}$ and a local private memory $\mathbf{L}$ (of constant size) and a local non-private memory $\mathbf{L}'$ (where the verification key is stored). More specifically, there is a simulator sim and a reduction $\mathbf{C}$ such that for all distinguishers $\mathbf{D}$,*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\ \mathsf{auth}^{sig}_{\mathcal{P}}[\mathbf{L}, \mathbf{L}', \mathbf{SMR}^{k}_{\Sigma_1,2\ell}], \mathsf{honSrv}^{\mathsf{S}}\mathbf{aSMR}^{k}_{\Sigma,\ell}) = 0$$

$$\text{and} \qquad \Delta^{\mathbf{D}}(\mathsf{auth}^{sig}_{\mathcal{P}}[\mathbf{L}, \mathbf{L}', \mathbf{SMR}^{k}_{\Sigma_1,2\ell}], \mathsf{sim}^{\mathsf{S}}\mathbf{aSMR}^{k}_{\Sigma,\ell}) \leq \Gamma^{\mathbf{DC}}(\mathbf{G}^{\mathsf{EU-CMA}}).$$

Proof. The simulator, the reduction, and the proof are analogous to Theorem 1 and hence omitted. □

5.2 Confidential from Authentic Server-Memory Resources

The protocol. We again specify two converters, which we call $\mathsf{init}_{\mathrm{priv}}$ (for initialization) and priv_{RW} (for the clients). Let $\mathsf{ENC} = (\mathcal{G}, \mathcal{E}, \mathcal{D})$ be a (CPA-secure) private-key encryption

Converter $\mathsf{init}_{\mathrm{priv}}$

```
Interface out
Input: init :
    $sk \leftarrow \mathcal{G}$
    output (write, 1, $sk$) at in to L
    output init at in to aSMR
    for $i = 1$ to $\ell$ do
        (write, $i$, $\mathcal{E}_{sk}(\lambda)$)

Input: (read, $i$)
    Defined the same as for $\mathsf{priv}_{RW}$ in Fig. 10

Input: (write, $i$, $x$) :
    Defined the same as for $\mathsf{priv}_{RW}$ in Fig. 10

Input: initComplete :
    output initComplete at in to aSMR
```

Converter priv_{RW}

```
Interface out
Input: (read, $i$)
    output (read, 1) at in to L
    Let $sk$ be the returned value
    output (read, $i$) at in to aSMR
    Let $c$ be the returned value
    if $c \neq \epsilon$ then
        return $\mathcal{D}_{sk}(c)$
    else
        return $\epsilon$

Input: (write, $i$, $x$)
    output (read, 1) at in to L
    Let $sk$ be the returned value
    output (write, $i$, $\mathcal{E}_{sk}(x)$) at in to aSMR
    if the write operation returns $\epsilon$ then
        return $\epsilon$ at out
```

Fig. 10. The initialization protocol (left) and the converter for the clients (right) to realize a confidential server memory from an authentic server memory.

scheme with message space Σ, ciphertext space $\mathcal{C}$, and key space $\mathcal{K}$: To initialize, $\mathsf{init}_{\mathrm{priv}}$ executes $\mathcal{G}$ to get a key sk and stores the key in the local memory $\mathbf{L}$. To read and write to the authentic server-memory resource, the converters behave as follows: On input $(\mathtt{write}, i, x)$ at the outer interface, encrypt x and output $(\mathtt{write}, i, \mathcal{E}_{sk}(x))$ to **aSMR**. If the write-operation returns ϵ (indicating an error), output ϵ at the outer interface. On input $(\mathtt{read}, i)$ at the outer interface, output $(\mathtt{read}, i)$ to **aSMR**. If the received ciphertext is $c \neq \epsilon$, output $\mathcal{D}_{sk}(c)$ at the outer interface and ϵ otherwise. The protocol is described in detail in Fig. 10.

Theorem 3. *Let $k, \ell \in \mathbb{N}$ and let Σ be an alphabet. The described protocol, i.e., the tuple of converters* priv $:= (\mathsf{init}_{\mathrm{priv}}, \mathsf{priv}_{RW}, \ldots, \mathsf{priv}_{RW})$ *(with a private-key encryption scheme* ENC *with ciphertext space $\mathcal{C}$) constructs the confidential (and authentic) server-memory resource* $\mathbf{cSMR}^k_{\Sigma,\ell}$ *from the authentic server-memory resource* $\mathbf{aSMR}^k_{\mathcal{C},\ell}$ *and a local private memory* $\mathbf{L}$ *(of constant size), with respect to the simulator* $\mathsf{sim}_{\mathrm{priv}}$ *as defined in Fig. 11 and the pair* $(\mathsf{honSrv}, \mathsf{honSrv})$. *More specifically, we construct a reduction* $\mathbf{C}_I$ *such that for all distinguishers* $\mathbf{D}$,

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\ \mathsf{priv}_{\mathcal{P}}[\mathbf{L}, \mathbf{aSMR}^k_{\mathcal{C},\ell}], \mathsf{honSrv}^{\mathsf{S}}\mathbf{cSMR}^k_{\Sigma,\ell}) = 0$$

$$\textit{and} \qquad \Delta^{\mathbf{D}}(\mathsf{priv}_{\mathcal{P}}[\mathbf{L}, \mathbf{aSMR}^k_{\mathcal{C},\ell}], \mathsf{sim}^{\mathsf{S}}_{\mathrm{priv}}\mathbf{cSMR}^k_{\Sigma,\ell}) = q \cdot \Delta^{\mathbf{DC}_I}(\mathbf{G}^{\mathsf{CPA}}_0, \mathbf{G}^{\mathsf{CPA}}_1),$$

where q is the total number of write operations at the client interfaces.

Proof (Sketch.). The correctness condition is again easy to verify. For the security condition, consider the simulator $\mathsf{sim}_{\mathrm{priv}}$ in Fig. 11 that generates an encryption key by its own and simulates the content for each write operation to be the encryption of the fixed value $\lambda \in \Sigma$. Furthermore, $\mathsf{sim}_{\mathrm{priv}}$ simply forwards deletion-operations to **aSMR**. To argue about the security, a simple hybrid argument follows.

Let q be an upper bound on the number of write-queries at the client interfaces. For $i \in \{0, \ldots, q\}$, we define the system $\mathbf{H}_i$ that behaves as $\mathsf{priv}_{\mathcal{P}}[\mathbf{L}, \mathbf{aSMR}^k_{\Sigma,\ell}]$ for the first i write-queries. However, for subsequent write-queries, not the real encrypted value is written to **aSMR**, but the encryption of λ. Hence, $\mathbf{H}_q$ is equivalent to $\mathsf{priv}_{\mathcal{P}}[\mathbf{L}, \mathbf{aSMR}^k_{\Sigma,\ell}]$ and H_0 is equivalent to $\mathsf{sim}^{\mathsf{S}}_{\mathrm{priv}}\mathbf{cSMR}^k_{\Sigma,\ell}$. Intuitively, since two adjacent hybrid systems $\mathbf{H}_{i-1}$ and $\mathbf{H}_i$ only differ in the way the ith write-query is encrypted (either the real value or λ), the overall security follows from the indistinguishability of ciphertexts.

To complete this last step, we define the reduction system $\mathbf{C}_i$ that behaves like $\mathbf{H}_i$, but instead of computing the encryptions and decryptions by itself, it queries the encryption and decryption oracles of game $\mathbf{G}^{\mathsf{CPA}}_b$. In particular, on the jth write-query input $(\mathtt{write}, i, x)$,

```
Converter $\mathsf{sim}_{\mathrm{priv}}$

Initialization
  $sk \twoheadleftarrow \mathcal{G}$
  $L_{init} \leftarrow (0, \mathtt{W}, 1, \mathcal{E}_{sk}(\lambda))||\ldots||(0, \mathtt{W}, \ell, \mathcal{E}_{sk}(\lambda))$
  $pos \leftarrow 1$

Interface $\mathsf{S}_H$
Input: getHist :
  UpdateLog
  return $L$

Input: $(\mathtt{read}, i), i \in [\ell]$ :
  UpdateLog
  if $S[i] = \epsilon$ then
      return $S[i]$
  else
      Determine the last entry in $L$ that wrote a value $c_i$ at location $i$
      return $c_i$

Interface $\mathsf{S}_I$   (intrusion = true)
Input: $(\mathtt{delete}, i) \in [\ell]$
  UpdateLog
  $S[i] \leftarrow \epsilon$
  output $(\mathtt{delete}, i)$ at in to cSMR

Input: $(\mathtt{restore}, i)$ :
  $S[i] \leftarrow \bot$
  output $(\mathtt{restore}, i)$ at in to cSMR

  procedure UpdateLog
      output getHist at in to cSMR
      Let Hist be the returned value
      for $j = pos$ to $|\textsc{Hist}|$ do
          if $\textsc{Hist}[j] = (0, \mathtt{init})$ then
              $L \leftarrow L_{init}$
          else if $\textsc{Hist}[j] = (k, \mathtt{R}, i)$ then
              $L \leftarrow L||(k, \mathtt{R}, i)$
          else if $\textsc{Hist}[j] = (k, \mathtt{W}, i)$ then
              $L \leftarrow L||(k, \mathtt{W}, i, \mathcal{E}_{sk}(\lambda))$
          else if $\textsc{Hist}[j] = (k, \mathtt{Fail}, i)$ then
              $L \leftarrow L||(k, \mathtt{Fail}, i, \mathcal{E}_{sk}(\lambda))$
      $pos \leftarrow |\textsc{Hist}| + 1$
```

Fig. 11. The simulator for the construction of a confidential memory.

ask $\mathbf{G}_b^{\mathsf{CPA}}$ for the encryption of x if $j < i$, ask $\mathbf{G}_b^{\mathsf{CPA}}$ for the encryption of λ if $j > i$, and, in case $j = i$, challenge game G_b with input (x, λ) to receive the ciphertext. We immediately see that

$$\mathbf{H}_i = \mathbf{C}_i \mathbf{G}_0^{\mathsf{CPA}} = \mathbf{C}_{i+1} \mathbf{G}_1^{\mathsf{CPA}}. \tag{1}$$

Let $\mathbf{C}_I$ be the system that first chooses $i \in \{1, \ldots, q\}$ uniformly at random and then behaves as $\mathbf{C}_i$ and let us define the distinguisher $\mathbf{D'} := \mathbf{D}\mathbf{C}_I$. We have that

$$\mathsf{Pr}[\mathbf{D'}(\mathbf{G}_0^{\mathsf{CPA}}) = 1] = \frac{1}{q} \cdot \sum_{i=1}^{q} \mathsf{Pr}[\mathbf{D}(\mathbf{C}_i \mathbf{G}_0^{\mathsf{CPA}}) = 1]$$

and

$$\mathsf{Pr}[\mathbf{D'}(\mathbf{G}_1^{\mathsf{CPA}}) = 1] = \frac{1}{q} \cdot \sum_{i=1}^{q} \mathsf{Pr}[\mathbf{D}(\mathbf{C}_i \mathbf{G}_1^{\mathsf{CPA}}) = 1] = \frac{1}{q} \cdot \sum_{i=0}^{q-1} \mathsf{Pr}[\mathbf{D}(\mathbf{C}_i \mathbf{G}_0^{\mathsf{CPA}}) = 1],$$

where the last equality follows from Equation 1.

Finally, we compute the distinguishing advantage by

$$\begin{aligned}
&\Delta^{\mathbf{D}}(\underbrace{\mathsf{priv}_{\mathcal{P}}[\mathbf{L}, \mathbf{aSMR}^{k}_{\Sigma,\ell}]}_{\mathbf{H}_q=\mathbf{C}_q\mathbf{G}_0^{\mathsf{CPA}}}, \underbrace{\mathsf{sim}^{\mathsf{S}}_{\mathrm{priv}}\mathbf{cSMR}^{k}_{\Sigma,\ell}}_{\mathbf{H}_0=\mathbf{C}_0\mathbf{G}_0^{\mathsf{CPA}}})\\
&= |\mathsf{Pr}[\mathbf{D}(\mathbf{C}_q\mathbf{G}_0^{\mathsf{CPA}}) = 1] - \mathsf{Pr}[\mathbf{D}(\mathbf{C}_0\mathbf{G}_0^{\mathsf{CPA}}) = 1]|\\
&= |\sum_{i=1}^{q} \mathsf{Pr}[\mathbf{D}(\mathbf{C}_i\mathbf{G}_0^{\mathsf{CPA}}) = 1] - \sum_{i=0}^{q-1} \mathsf{Pr}[\mathbf{D}(\mathbf{C}_i\mathbf{G}_0^{\mathsf{CPA}}) = 1]|\\
&= q \cdot |\mathsf{Pr}[\mathbf{D}'(\mathbf{G}_0^{\mathsf{CPA}}) = 1] - \mathsf{Pr}[\mathbf{D}'(\mathbf{G}_1^{\mathsf{CPA}}) = 1]| = q \cdot \Delta^{\mathbf{D}'}(\mathbf{G}_0^{\mathsf{CPA}}, \mathbf{G}_1^{\mathsf{CPA}}).
\end{aligned}$$

This concludes the proof. □

5.3 Secure from Confidential Server-Memory Resources

We present an enhanced version of the Path ORAM protocol. The original Path ORAM protocol is due to Stefanov et al. [57]. In particular, we complement the original protocol with a proper error handling such that the protocol realizes the secure server-memory resource from an authentic and confidential server-memory resource.

Overview and notation. The protocol maintains a tree structure on the 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 [57]). 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 ith 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 $\mathsf{position}$, 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 $\mathsf{position}[i] = x$, 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 [57].

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., $\mathsf{position}[i] \twoheadleftarrow \{1, \ldots \ell\}$. 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. We give the formal description of converter $\mathsf{init}_{\mathrm{sec}}$ in Fig. 12. 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 $x \leftarrow \mathsf{position}[i]$ and reassigns $\mathsf{position}[i]$ 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

```
Converter $\mathsf{init}_{\mathsf{sec}}$

Interface out
Input: init :
  Prepare $2\ell - 1$ nodes $N_r \leftarrow (0, \lambda)^Z$
  $S \leftarrow \emptyset$
  $\mathsf{position} \leftarrow$ empty list with a capacity of $\ell$ items
  for $i = 1$ to $\ell$ do
      $S \leftarrow S \cup \{(i, \lambda)\}$
      $\mathsf{position}[i] \twoheadleftarrow \{1, \ldots \ell\}$
  for $x = 1$ to $\ell$ do
      for $lv = L, L - 1, \ldots, 0$ do
          Let $r$ be the memory address of node $\mathcal{P}(x, lv)$
          Let $d$ be the number of dummy blocks in node $N_r$
          $S' \leftarrow \{(i', v') \in S \mid \mathcal{P}(x, lv) = \mathcal{P}(\mathsf{position}[i'], lv)\}$
          $m \leftarrow \min\{|S'|, Z, d\}$
          Truncate $S'$ to $m$ blocks
          $S \leftarrow S \setminus S'$
          Denote the blocks in $S'$ by $b_i$, $i = 1 \ldots m$
          for $k = 1$ to $Z$ do
              if $N_r[k] = (0, \lambda)$ then
                  $N_r[k] \leftarrow b_k$
  for all nodes $N_r$ in decreasing order of index do
      output $(\mathtt{write}, r, N_r)$ at in to cSMR

Input: $(\mathtt{read}, i)$ or $(\mathtt{write}, i, x)$:
  Defined the same as for $\mathsf{sec}_{\mathrm{RW}}$ in Fig. 13

Input: initComplete :
  output initComplete at in to cSMR
```

Fig. 12. The initialization protocol for the realization of a secure server memory.

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 $\mathsf{position}[i] \leftarrow \epsilon$. 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 re-writes 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 can be found Fig. 13. We denote this client converter by $\mathsf{sec}_{\mathrm{RW}}$. The security of the protocol is assured by the following theorem. It implies that the above error-handling for Path ORAM is sufficient to realize the secure server-memory resource and to ensure strong security guarantees.

Theorem 4. *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 $\mathsf{sec} := (\mathsf{init}_{\mathsf{sec}}, \mathsf{sec}_{\mathrm{RW}}, \ldots, \mathsf{sec}_{\mathrm{RW}})$ (with k copies of $\mathsf{sec}_{\mathrm{RW}}$) constructs the secure server-memory resource $\mathbf{sSMR}^{k,1}_{\Sigma,\ell}$ from the confidential (and authentic) server-memory resource $\mathbf{cSMR}^{k}_{\Sigma_1,2\ell}$ and a local memory, with respect to the simulator $\mathsf{sim}_{\mathsf{sec}}$ described in Fig. 14 and the pair $(\mathsf{honSrv}, \mathsf{honSrv})$. More specifically, for all distinguishers $\mathbf{D}$*

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

Proof. We prove the security condition and again analyze the input-output behavior of both systems involved. To this end, we consider the possible inputs at each interface.

On input init, initComplete at interface C_0: On input init to the real system, i.e., to $\mathsf{sec}_{\mathcal{P}}[\mathbf{L}, \mathbf{cSMR}^{k}_{\Sigma_1,2\ell}]$, the converter $\mathsf{init}_{\mathsf{sec}}$ first initializes the position map $\mathsf{position}$ by

Converter $\mathsf{sec}_{\mathrm{RW}}$

```
Interface out
Input: $(\mathtt{read}, i)$
  $v \leftarrow$ UpdatePath$(\mathtt{R}, i, \perp)$
  return $v$

Input: $(\mathtt{write}, i, v)$
  $v' \leftarrow$ UpdatePath$(\mathtt{W}, i, v)$
  if $v' = \perp$ then
      return $\epsilon$
  else
      return ok

function UpdatePath$(op, i, v')$
    Res $\leftarrow \perp$
    Retrieve the stash $S$ and the position table position from $\mathbf{L}$
    $x \leftarrow$ position$[i]$
    if $x \neq \epsilon$ then
        position$[i] \twoheadleftarrow \{1, \ldots, \ell\}$
        for $lv = L, L-1, \ldots, 0$ do
            Let $r$ be the memory address of node $\mathcal{P}(x, lv)$
            output $(\mathtt{read}, r)$ at in to cSMR
            Store the returned value as $N_r$
        if all fetched nodes $N_r \neq \epsilon$ and $\mathcal{P}(x)$ is not marked as invalid then
            for each node $N_r$ do
                Parse $N_r$ as a list of $Z$ blocks $b_i \in \{(i, v) \mid i \in \mathbb{N}\} \cup \{\lambda\}$
                for $i = 1$ to $Z$ do
                    if $b_i \neq (0, \lambda)$ then
                        $S \leftarrow S \cup \{b_i\}$
            Retrieve block $b$ from $S$ such that $b = (i, v)$ for some $v \in \Sigma$
            Res $\leftarrow v$
            if $op = \mathtt{W}$ then
                Replace $(i, v)$ in $S$ by $(i, v')$
            for $lv = L, L-1, \ldots, 0$ do
                Let $r$ be the memory address of node $\mathcal{P}(x, lv)$
                $N \leftarrow [\,]$
                $S' \leftarrow \{(i', v') \in S \mid \mathcal{P}(x, lv) = \mathcal{P}(\text{position}[i'], lv)\}$
                $m \leftarrow \min\{|S'|, Z\}$
                Truncate $S'$ to $m$ blocks
                $S \leftarrow S \setminus S'$
                Denote the blocks in $S'$ by $b_i$, $i = 1 \ldots m$
                for $k = 1$ to $Z$ do
                    if $k \leq m$ then
                        $N \leftarrow N || b_k$
                    else
                        $N \leftarrow N || (0, \lambda)$
                output $(\mathtt{write}, N, r)$ at in to cSMR
                if the write query returns $\epsilon$ then
                    Mark all paths containing node $\mathcal{P}(x, lv)$ as invalid[a]
        else
            position$[i] \leftarrow \epsilon$
            for $lv = L, L-1, \ldots, 0$ do
                Let $r$ be the memory address of node $\mathcal{P}(x, lv)$
                output $(\mathtt{write}, r, N_r)$ at in to cSMR
    else              ▷ Simulate dummy accesses if logical address $i$ is marked invalid.
        $x \twoheadleftarrow \{1, \ldots, \ell\}$
        for $lv = L, L-1, \ldots, 0$ do
            Let $r$ be the memory address of node $\mathcal{P}(x, lv)$
            output $(\mathtt{read}, r)$ at in to cSMR
            Store the returned value as $N_r$
        for $lv = L, L-1, \ldots, 0$ do
            Let $r$ be the memory address of node $\mathcal{P}(x, lv)$
            output $(\mathtt{write}, r, N_r)$ at in to cSMR
    Store the stash $S$ and the position table position in $\mathbf{L}$
    return Res
```

[a] Encoded in the position table

Fig. 13. The converter for the clients to realize a secure server memory from a confidential and authentic server memory.

Converter $\mathsf{sim}_{\mathrm{sec}}$

```
Initialization
  for $i = 1$ to $2\ell - 1$ do
      $N_i \leftarrow$ valid
  Let $T$ be the binary tree consisting of nodes $N_0, \ldots, N_{2\ell-1}$ in linear ordering
  $L_{init} \leftarrow (0, \texttt{init})||(0, \texttt{W}, 2\ell - 1)|| \ldots ||(0, \texttt{W}, 1)$
  $pos \leftarrow 1$
  $L \leftarrow [\,]$

Interface $\mathsf{S}_H$
Input: getHist :
  UpdateLog
  return $L$

Input: $(\texttt{read}, r), r \in [2\ell - 1]$ :
  UpdateLog
  return $\lambda$

Interface $\mathsf{S}_I$   (intrusion = true)
Input: $(\texttt{delete}, r) \in [2\ell - 1]$
  UpdateLog
  $N_r \leftarrow$ invalid
  $I \leftarrow \{i \in [\ell] \mid \text{Path } \mathcal{P}_T(i) \text{ contains at least one invalid node}\}$
  $\alpha \leftarrow \frac{|I|}{\ell}$
  output $(\texttt{pollute}, \alpha)$ at in to sSMR

Input: $(\texttt{restore}, r)$ :
  UpdateLog
  $I_{\text{old}} \leftarrow \{i \in [\ell] \mid \text{Path } \mathcal{P}_T(i) \text{ contains at least one invalid node}\}$
  $N_i \leftarrow$ valid
  $I_{\text{new}} \leftarrow \{i \in [\ell] \mid \text{Path } \mathcal{P}_T(i) \text{ contains at least one invalid node}\}$
  $\delta \leftarrow \frac{|I_{\text{old}}| - |I_{\text{new}}|}{\ell}$                                  ▷ $I_{\text{new}} \subseteq I_{\text{old}}$
  output $(\texttt{reducePollution}, \delta)$ at in to sSMR

  procedure UpdateLog
      output getHist at in to sSMR
      Let Hist be the returned value
      for $j = pos$ to $|\textsc{Hist}|$ do
          if $\textsc{Hist}[j] = (0, \texttt{init})$ then
              $L \leftarrow L_{init}$
              $I \leftarrow \emptyset$
          else if $\textsc{Hist}[j] = (k, \texttt{Access})$ then
              $I \leftarrow \{i \in [\ell] \mid \text{Path } \mathcal{P}_T(i) \text{ contains only valid nodes}\}$
          else if $\textsc{Hist}[j] = (k, \texttt{Failed})$ then
              $I \leftarrow \{i \in [\ell] \mid \text{Path } \mathcal{P}_T(i) \text{ contains at least one invalid node}\}$
          if $I \neq \emptyset$ then
              $x \twoheadleftarrow I$
              for $lv = L, L - 1, \ldots, 0$ do                          ▷ Simulate read access.
                  Let $r$ be the address of simulated node $N_r = \mathcal{P}_T(x, lv)$
                  $L \leftarrow L||(k, \texttt{R}, r)$
              for $lv = L, L - 1, \ldots, 0$ do                          ▷ Simulate write access.
                  Let $r$ be the address of simulated node $N_r = \mathcal{P}_T(x, lv)$
                  if $N_r =$ valid then
                      $L \leftarrow L||(k, \texttt{W}, r)$
                  else
                      $L \leftarrow L||(k, \texttt{Fail}, r)$
      $pos \leftarrow |\textsc{Hist}| + 1$
```

Fig. 14. The simulator for the construction of a secure memory.

assigning to each logical address i the corresponding leaf number uniformly at random. Then, the converter stores all initial blocks (i, λ) for $i = 1 \ldots \ell$ in the stash S. Subsequently, the binary tree T consisting of nodes N_0 to $N_{2\ell-1}$ (in the usual linear ordering) is locally built: each path from any leaf to the root is examined and as many blocks as possible are pushed from the stash S to a node in the tree. After this step, a block (i, λ) is either found in the stash S (stored in the local memory) or within the tree in any of the nodes of the path $\mathcal{P}(i)$. Finally, the whole tree is written to the server-memory resource (in decreasing order of the linear index), which adds $2\ell - 1$ entries $(0, \mathtt{W}, r)$ for $r = 2\ell - 1 \ldots 1$ to the history.

In the ideal system $\mathsf{sim}^{\mathsf{S}}_{\mathrm{sec}}\mathbf{sSMR}^{k,1}_{\Sigma,\ell}$, the command sets the value of any storage location to λ and adds the initial entry $(0, \mathtt{init})$ to the history. The simulator will replace this first entry by the list L_{init} that contains the simulated accesses that would happen in the real world. This perfectly emulates the real-world view.

Finally, on input `initComplete`, both systems deactivate interface C_0 and the other client interfaces are operational from this point onwards.

On input $(\mathtt{read}, i)$ at interface C_k: Upon this input at a client interface of the real system $\mathsf{sec}_{\mathcal{P}}[\mathbf{L}, \mathbf{cSMR}^{k}_{\Sigma_1,2\ell}]$, the protocol executes a write access to the memory resource. Inspecting the program code in Fig. 13, the function updatePath is executed in read-mode, i.e., where the operation $op = \mathtt{R}$ (and hence no other arguments need to be specified). First, the current leaf node of address i is read from the position table (line 2). The program now branches into two tracks: if a valid leaf index x is returned, then the instructions on lines 5 to 41 are executed. The other case corresponds to the event that a previous access to logical address i was invalid and lines 42 to 50 are executed instead. Let us focus on the successful branch first: the client downloads all nodes corresponding to the path from the xth leaf node to the root. The accessed path is determined in a uniformly random way, since each time a path for logical address i is successfully accessed, a new uniformly random value is written to the position and determines to be accessed the next time when address i is to be read (line 6). If all retrieved nodes are valid, i.e., if the test on line 11 is passed, all the blocks contained in the nodes are added to the local stash S (lines 12 to 16) and finally the retrieved value is read from the stash (lines 17 and 18). To conclude this operation, the updated path is written back to the confidential server memory (lines 21 to 34). The update step tries to push as many blocks as possible from the stash into the tree nodes. Only the blocks (j, v) can be inserted into a node in the intersection of $\mathcal{P}(x)$ and $\mathcal{P}(j)$ (condition on line 24). However, if the test on line 11 is not passed, i.e., if an invalid node is retrieved, then the currently read logical block is declared as invalid by setting the position table $\mathsf{position}[i] \leftarrow \epsilon$ on line 38. Furthermore, the client simply writes back the nodes it just retrieved without modification. Some of these writes might not be successful but this can safely be ignored (as nothing is changed). Overall, we conclude that this branch adds in any case $\log(\ell)$ read-requests and $\log(\ell)$ write-requests to the history of **cSMR**.

The second branch is taken if the position i is known to have failed in the past (lines 42 to 50). Then, the protocol simply sends $\log(\ell)$ read-requests to the server to retrieve a randomly chosen path and then rewrites the path unaltered. This adds another $\log(\ell)$ write-request to the history.

Overall, the probability that an access to logical address i is successful given there has not been an invalid access[4] since initialization, is exactly the ratio of the number of valid paths and all ℓ paths. Similarly, the probability that an access to logical address i is invalid given there has not been an invalid access since initialization, is exactly the ratio of the number of invalid paths and all ℓ paths. In any other case, an access to logical address i will return ϵ with probability one. Finally, we observe that on input $(\mathtt{read}, i)$ each of the ℓ paths of the tree has equal probability to be accessed.

Let us now consider the ideal system $\mathsf{sim}^{\mathsf{S}}_{\mathrm{sec}}\mathbf{sSMR}^{k,1}_{\Sigma,\ell}$. Upon a read-query, we again have two possible branches. This is seen by inspecting the program code of **sSMR** on a client-

[4] We mean an access that returned ϵ.

read request at interface C_k for $k > 0$.[5] Given that there has never been an invalid access to address i, the probability of a successful access is exactly $1 - \alpha$, and that of an invalid access is exactly α, where α is the pollution factor that can be set by the simulator. In each step of the execution, the simulator $\mathsf{sim}_{\mathrm{sec}}$ maintains the invariant that α equals the ratio of invalid paths and all ℓ possible paths. In particular, as explained below, on each deletion-query by the distinguisher, the simulator updates the parameter α accordingly.

We now look at how the simulator simulates the real-world memory access and maintains the simulated history. The simulator is informed, whether an operation was evaluated to be successful (entry $(k, \mathtt{Access})$ in the history), or whether it was evaluated to be a fail (entry $(k, \mathtt{Failed})$ in the history). In the first case, the simulator chooses a random path from all the paths that only contain valid nodes. (The statistics which nodes are valid and which are not is maintained as explained below for input $(\mathtt{delete}, r)$ to interface S_I). In the second case, the simulator simulates the accesses to a random path from the set of all paths that contain at least one invalid node. Overall, this means that on input $(\mathtt{read}, i)$ to resource **sSMR**, the probability for any fixed path to be added to the history is $\frac{1}{\ell}$. This is easily seen by a case distinction: the probability that a particular valid path is added to the history is $(1 - \alpha) \cdot \frac{\#\text{valid paths}}{\ell}$. For $\alpha = \frac{\#\text{invalid paths}}{\ell}$ this gives us a probability of $\frac{1}{\ell}$ for all valid paths. The other case is analogous and we see that on each read-request, a uniformly random path is added to the history.

We can conclude that the behavior of the simulator mimics the real world behavior. In particular, the simulated history is updated accordingly such that the failure probabilities are identical, as well as the distribution of the access pattern in the simulated history.

On input $(\mathtt{write}, i, x)$ at interface C_k: On a write-instruction to the systems, the same function UPDATEPATH is executed, but with arguments $op = \mathtt{W}$, i and v', where v' is the new value for address i. The code for this case is identical to the read case except for the instructions on lines 19 and 20. Since these two lines do not affect the observable behavior, the analysis of this case follows from the analysis of the previous analysis of the read-instructions.

On input getHist at interface S_H: In the real system, the output is the history of **cSMR**. By the above analysis, a straightforward inductive argument shows that in case of system $\mathsf{sim}^{\mathsf{S}}_{\mathrm{sec}}\mathbf{sSMR}^{k,1}_{\Sigma,\ell}$, the simulator's internally maintained history L, which is output upon this query, emulates the real-world view perfectly.

On input $(\mathtt{restore}, r)$ at interface S_I: In the real system, the restore operation makes a node, which was invalid before, become valid again. This means that the number of valid paths might increase. In fact, for all logical address i, that have not failed on any access so far, the probability thus increases that the next read or write request is successful. The already failed addresses are not affected by this change since the local position table is not affected by a restore command.

In the ideal system, the simulator updates the pollution factor α of the server memory **sSMR** accordingly by recomputing the ratio of invalid paths after the node N_r becomes valid again (note that this ratio will not increase). Hence, in both worlds, the effects of a restore command are identical.

On input $(\mathtt{delete}, r)$ at interface S_I: In the real system, the delete operation makes a node, which was valid before, become invalid. This means that the number of invalid paths increases. In fact, for all logical address i, that have not failed on any access so far, the probability thus increases that the next read or write request fails. The already failed addresses are not affected by this change since the local position table is not affected by a deletion command.

In the ideal system, the simulator updates the pollution factor α of the server memory **sSMR** by recomputing the ratio of invalid paths after the node N_r becomes invalid again. Hence, in both worlds, the effects of a deletion command are identical.

[5] Recall that for the sake of simplicity (and without loss of generality), we do not assume any failure during the initialization phase.

On input $(\texttt{read}, r)$ **at interface** S_H**:** On any command $(\texttt{read}, r)$ both systems simply return the dummy symbol λ. This holds by definition of system **cSMR** in the real world and by definition of the simulator $\mathsf{sim}_{\text{sec}}$ in the ideal world.

On inputs `startWriteMode` **and** `stopWriteMode` **at interface** W**:** In case of the real system $\mathsf{sec}_{\mathcal{P}}[\mathbf{L}, \mathbf{cSMR}^{k}_{\Sigma_1, 2\ell}]$, the first input allows the adversary to access and modify the server storage until the input `stopWriteMode` is input. The same holds for the the ideal system $\mathsf{sim}^{\mathsf{S}}_{\text{sec}}\mathbf{sSMR}^{k,1}_{\Sigma,\ell}$, since the simulator does not react on adversarial queries at interface S_I in case INTRUSION $= \mathsf{false}$ and is allowed to access interface S_I of resource **sSMR** if and only if INTRUSION is set.

This ends our analysis of the behavior. We conclude that on each input, the observable effects are identical for the real system and the ideal system. The statement follows. □

Client-side storage reduction. At first sight, the client storage overhead seems unpractical since the size of the position map is $\ell \log(\ell)$ bits, which corresponds roughly to ℓ data blocks if we assume that each data block has a size B of (at least) $\log(\ell)$ bits. There are a couple of techniques suggested to reduce this storage overhead. Stefanov et al. [56] describe that under realistic workloads such as mostly sequential accesses in file systems, the position table might consume only 0.13ℓ bytes, achieved for example by just maintaining a counter per block instead of its random position and to infer the block's position via a PRF applied to the current counter and block number. This suggests that even for an outsourced storage in the order of a couple of terabytes, the position table would not exceed one gigabyte. A second technique to reduce the client storage overhead is by outsourcing the position itself in a clever way. However, not all schemes are equally suitable as will be discussed in the next section.

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 $\mathbf{sSMR}^{k,t}_{\Sigma,\ell}$ from $\mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}$. 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 sketch the converter for initialization, denoted $\mathsf{init}_{\text{rep},t}$ and the client converter rep_t.

On input `init` at the outer interface of $\mathsf{init}_{\text{rep},t}$, output `init` to $\mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}$ and additionally store for each $i \in \ell$ the value c_i (initially zero) in the local storage $\mathbf{L}$. The value c_i denotes the number of failed accesses to logical address i. On input $(\texttt{read}, i)$ to converter $\mathsf{init}_{\text{rep},t}$ or rep_t, output $(\texttt{read}, i + \min\{c_i, t-1\})$ and return whatever is returned by resource $\mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}$. In case ϵ is returned, the converter sets $c_i \leftarrow c_i + 1$. On input $(\texttt{write}, i, x)$ to converter $\mathsf{init}_{\text{rep},t}$ or rep_t output $(\texttt{write}, i + r, x)$ to $\mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}$ for all $r = 0 \ldots t-1$ and output ϵ at the outer interface for each failed write access to the resource (and ok for the others). For this protocol, one can show the following lemma:

Lemma 1. *Let $k, \ell, t \in \mathbb{N}$. be a secure server-memory resource with the usual parameters. The above described replication protocol* $\mathsf{rep} := (\mathsf{init}_{rep,t}, \mathsf{rep}_t, \ldots, \mathsf{rep}_t)$ *(with k copies of* rep_t*) constructs the secure server-memory resource* $\mathbf{sSMR}^{k,t}_{\Sigma,\ell}$ *from the secure server-memory resource* $\mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}$*. More specifically, there is a simulator* $\mathsf{sim}_{\text{rep}}$ *such that for all distinguishers* $\mathbf{D}$*,*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\ \mathsf{rep}_{\mathcal{P}}[\mathbf{L}, \mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}], \mathsf{honSrv}^{\mathsf{S}}\mathbf{sSMR}^{k,t}_{\Sigma,\ell}) = 0$$
$$\textit{and} \qquad \Delta^{\mathbf{D}}(\mathsf{rep}_{\mathcal{P}}[\mathbf{L}, \mathbf{sSMR}^{k,1}_{\Sigma,t\cdot\ell}], \mathsf{sim}^{\mathsf{S}}_{\text{rep}}\mathbf{sSMR}^{k,t}_{\Sigma,\ell}) = 0.$$

5.4 Do all ORAM Schemes realize a Secure Server-Memory Resource?

Our formalization 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 elaborate on two popular ORAM schemes. We show that in the recursive Path ORAM scheme by Stefanov et al. [56], failures among memory locations are correlated. In the case of the Goodrich-Mitzenmacher ORAM scheme [33], we show that the failure probabilities are not the same for all (logical) memory locations. As we explain, the latter property is structural and therefore applies also to stronger notions, for example ORAM schemes that satisfy the NRPH-property of [19].

The recursive Path ORAM scheme. A beautiful technique to reduce the client storage overhead is by using the Path ORAM scheme recursively as suggested by Stefanov et al. [56]. In recursive Path ORAM, the position table itself is outsourced using another (and smaller) instance of a Path ORAM scheme. This smaller instance could itself outsource its position table to an even smaller ORAM scheme etc. The final instance (i.e., the base case), stores its position table in the local memory. Assuming a constant block size $B > \log(n)$, each recursive instance reduces the number of positions by a factor $f := \frac{B}{\log(n)} > 1$, where each block is used to store (roughly) f entries of the position table. Hence, after recursion depth in the order of $O(\log(\ell))$, the position table stored in the client storage is of size roughly $\log(\ell)$ data blocks. A formal proof of this is given in [56].

Let us first describe one (recursion) step of this procedure in our formalism: We consider the similar scenario as before, but we replace the local memory $\mathbf{L}$ by an instance of a secure server-memory resource $\mathbf{sSMR}$. This additional secure server storage memory has $\ell' < \ell$ storage locations, each of which holds a tuple of f values of the position table $\mathsf{position}$. The protocols need to be adapted only slightly: let the converter $\mathsf{init}'_{\mathrm{sec}}$ be defined as $\mathsf{init}_{\mathrm{sec}}$ except that the position table is written to secure server-memory resource instead of the private memory $\mathbf{L}$. Let further $\mathsf{sec}'_{\mathrm{RW}}$ be defined as converter $\mathsf{sec}_{\mathrm{RW}}$ but instead of the instruction $x \leftarrow \mathsf{position}[i]$, the converter computes $q \leftarrow (i-1)$ div f and sends a read instruction $(\mathtt{read}, q+1)$ to the secure memory to obtain the tuple $(\mathsf{position}[fq+1], \ldots, \mathsf{position}[f(q+1)-1])$, where the desired value x is at position $i - qf$ in the tuple. Similarly, the subsequent update step $\mathsf{position}[i] \leftarrow x$ now consists of first updating the tuple at the respective location and then sending a write instruction to the secure memory resource to write the entire tuple back to location $q+1$.

The question now is: does the protocol still realize a secure server-memory resource? Unfortunately, the answer to this question is negative. On an intuitive level, the reason is that logical memory addresses are grouped in blocks. For example, the logical memory locations $i = 1 \ldots f$, i.e., their mappings $\mathsf{position}[1] \ldots \mathsf{position}[f]$, are an atomic block in the recursive Path ORAM scheme. This, however, implies that if the lookup fails for one logical address in $\mathsf{sec}'_{\mathrm{RW}}$, then it fails for all the others in that block as well. For the overall scheme, this means that failing to access the value at location $i = 1$ is not independent of failing to access the value at location $i = 2$ etc. In contrast, failing to access the value at location $f+1$ is again independent, as it resides in a different block of the smaller ORAM scheme. It is easy to exploit this observation to design a distinguisher that distinguishes the system[6] $\mathsf{sec}'_{\mathcal{P}}[\mathbf{sSMR}, \mathbf{cSMR}]$ and its ideal goal, the desired secure memory resource, with noticeable advantage. This scheme thus only constructs a weaker variant of the resource, where failures among data blocks are correlated.

ORAM schemes based on a hierarchical structure of hash-tables. A prominent ORAM scheme in this category is due to Goodrich and Mitzenmacher [33] which is based on cuckoo-hashing and follows the hierarchical approach envisioned by Goldreich and Ostrovsky [32]. The hierarchical approach organizes the data in levels, where each level is capable of storing two times as many elements as the level above. The first level can be thought of as an array of small size. The lowest level is capable of storing ℓ elements (and hence the number of levels is in $O(\log(\ell))$. Each level except the first is either a standard hash-table or a cuckoo

[6] We omit here the parameters of the systems for brevity.

hash-table[7] An element is encoded in the familiar form (i, v), where v denotes the value at logical address i. On a given level lv, if the pair resides in the table of that level, then it is found at location $H^{lv}(i)$, where H is a hash function.[8] To perform a search for an address i, each level lv is accessed (starting from the top level) and the location $H^{lv}(i)$ is read until a pair (i, v) is found. After the element has been found, the remaining tables are accessed at uniformly random locations. After all accesses have been performed, the pair is inserted into the top level array. We denote this random walk through the tables succinctly by $RW(i)$ and understand the above procedure. Inserting the element into the top level is a crucial step: intuitively, the access pattern does not reveal any information, since after each successful search (to random locations from the servers point of view), the element will be found in the top level in a subsequent search and the accesses to lower tables still look random. Overall, no lookup for an address i (accessing position $H^{lv}(i)$ on level lv) is performed twice for the same table. To maintain this invariant, and to prevent tables from overflowing, periodic rebuild phases occur. Such a rebuild phase servers two purposes: first, it moves data from one level to the next lower level in an oblivious way. This has the effect that infrequently accessed items are more likely to be found in lower tables. Second, new hash functions $H^{lv}(.)$ are chosen for the levels to keep the access pattern random looking. Both of these steps (regular re-hashing and moving elements downwards in the hierarchy) are crucial to prove its security in a passive setting [39,33] or in an active setting where the protocol aborts upon detecting an error.

Similar to the construction presented in Sect. 5.3, it seems that each access is essentially a random walk from the top-level to the bottom level and one would probably expect that we have equal failure probability on any search for the item with logical address i. More precisely, we say that an access to address i is successful if it has never failed in the past and the current random walk $RW(i)$ does not access any deleted cell. Thus, one could suspect that a similar adaption of the GM scheme would realize **sSMR** from a confidential and authentic memory. However, this is not the case as one can see using the following thought experiment which lets us conclude that there is a strategy that makes elements of tables higher in the hierarchy fail with significantly smaller probability than elements residing towards the bottom of the hierarchy. Imagine a hierarchy of tables and assume that the address-value pair $(1, v)$ is currently stored at some level lv and that no failure has occurred so far. We assume v to be a uniform random value of some alphabet. Consider an attacker that now deletes a uniform random location of the hash-table at level lv: with noticeable probability, this will hit exactly the pair $(1, v)$ and thus delete the information which value is associated to logical address 1. If this event happens, no search will ever be able to output v (except with negligible probability). Now, imagine that (roughly) 2^{lv} read operations for a different logical address, say 2, are performed. This number of accesses is sufficient to provoke a rebuild phase that moves all elements contained in level lv to the table at the lower lever. However, if the pair $(1, v)$ was deleted before, the lower level tables cannot contain the correct value for address 1 and hence any subsequent access to this address cannot return a consistent value. We can thus conclude: the probability that at this point in time, accessing logical address 1 returns the correct (and valid) value v is equal to the sum of the two probabilities that the actual random walk through the tables is valid *and* the probability that it has not been deleted before this last rebuild phase.

$$\begin{aligned}&\mathsf{Pr}[\text{Access to address 1 returns correct result}] = \\ &\qquad\qquad \mathsf{Pr}[\text{RW}(1) \text{ is valid}] + \mathsf{Pr}[\text{Pair } (1, v) \text{ was not deleted before at level lv}]. \qquad (2)\end{aligned}$$

Hence, the probability is not uniform for all data blocks: in fact, items stored at levels lower than lv have a significant lower probability of failing than items stored at levels greater than lv, since they do not have this additional error term on the right hand side. For example, the

[7] Usually the smaller levels are implemented using standard hash tables and larger levels are implemented using cuckoo hash-tables.

[8] H is usually modeled as a uniform random function.

probability of a read operation returning an invalid value is not the same for locations 1 and 2 for the above attacker strategy.

The above observation is likely to have an impact in practical settings, where the ORAM scheme is used as part of a larger application. Assume that the application stores some control information in the memory at a known locations (e.g., address 1) and does not frequently update this location such that the above considerations apply. Then, an attacker following the above strategy can conclude, that if the application signals an error (or any other special behavior) on any future access, then it is more likely that this access pattern corresponds to an access to logical location 1 than to any other location.

In comparison, this error signaling is not problematic if the underlying protocol fulfills our stronger security goal. The reason is that the attacker can then only introduce failures that are equally likely for all logical locations and thus there is no bias in the correlation of the error signal and the access pattern.

Remark 1. It is important to note on what core property the above argument builds in order to develop schemes that are not susceptible to the same issue. The core property of the attack, which for example also explains why stronger ORAM notions, such as the next-read-pattern-hiding (NRPH) requirement in [19], do not rule out the attack is the following: consider again Equation 2. The problem here is that the expected level at which the newest version of a location resides is quite predictable and hence this element can be somewhat accurately deleted. Under the above attack, the access to this location has to throw an error, since the data is simply not available after deletion (it was chosen uniformly at random and is not redundantly stored, so the system would have to guess it which is not possible with very high probability). While enhancing the ORAM scheme to achieve NRPH improves resilience against rewinding attacks (in particular, rewinding twice has to yield the same pseudo-random walk), it does not improve on hiding where a logical element is stored approximately. In particular, the NRPH scheme of [19] is also susceptible to the above attack.

6 Auditable Server-Memory Resources

In this section, we introduce the ideal abstraction of auditing mechanisms.

Basic, authenticated, and confidential auditable server memory. The ideal audit is described in Fig. 15. It provides security guarantees only in a phase where an intruder is not active.[9] In this case, the check reveals whether the current memory blocks are indeed the newest version that the client wrote to the storage. 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 realize $\mathbf{SMR}^{k,\text{audit}}_{\Sigma,n}$, $\mathbf{aSMR}^{k,\text{audit}}_{\Sigma,n}$, or $\mathbf{cSMR}^{k,\text{audit}}_{\Sigma,n}$ from an ordinary server-memory resource.

Secure and auditable server memory. We present the ideal audit for secure memory resources in Fig. 16. Due to the probabilistic nature of resource **sSMR**, the ideal retrievability guarantee for secure memory resources is a probabilistic one. Based on an additional parameter τ, the ideal audit of resource $\mathbf{sSMR}^{k,t_{\text{rep}},\tau,\mathsf{audit}}_{\Sigma,n}$ is successful if the probability that the entire memory cannot be retrieved is smaller than τ. The smaller τ, the stronger the retrievability guarantee.

[9] In fact, this is the only interesting case to consider, since if an intruder is active at the time of an audit, he can freely decide on the success or failure of the audit. We omit this second case in our specifications for simplicity.

Resource $\{\mathsf{a},\,\mathsf{c}\}\mathbf{SMR}^{k,\mathsf{audit}}_{\Sigma,n}$

Server-memory resources are augmented with a new client capability as follows:

Interfaces $\mathsf{C}_r, r \in \{1,\dots,k\}$

Input: $(\mathtt{read}, i) \in [n]$
Defined as in the respective resources **SMR**, **aSMR**, **cSMR**

Input: $(\mathtt{write}, i, x) \in [n] \times \Sigma$
Defined as in the respective resources **SMR**, **aSMR**, **cSMR**

```
Input: audit :
  if ACTIVE and not INTRUSION then
      output auditReq at $\mathsf{S}_H$
      Let $d \in \{\mathrm{allow}, \mathrm{abort}\}$ be the returned value from $\mathsf{S}_H$
      if $d = \mathrm{allow}$ then
          $\mathbb{M}' \leftarrow$ empty table
          for $i = 1$ to $n$ do
              if $\exists k, x, t : \textsc{Hist}[k] = (t, \mathtt{W}, i, x)$ then
                  $k_0 \leftarrow \max\{k \mid \exists t, x : \textsc{Hist}[k] = (t, \mathtt{W}, i, x)\}$
                  Parse $\textsc{Hist}[k_0]$ as $(t, \mathtt{W}, i, x_0)$
                  $\mathbb{M}'[i] \leftarrow x_0$
              else
                  $\mathbb{M}'[i] \leftarrow \lambda$
          if $\mathbb{M}' = \mathbb{M}$ then
              return accept
          else
              return reject
      else
          return reject
```

Fig. 15. Description of the auditable server-memory resources (only difference to ordinary server memory shown).

7 Constructing Auditable Server-Memory Resources

In this section, we provide constructions of auditable server-memory resources from ordinary server-memory resources. In order to show that a protocol achieves a construction, we again have to prove both conditions of Definition 1. In this section, the default behavior at interface S is possibly more complicated, especially if interaction between the server and the client is required, for example if the client requests the server to compute a hash during an audit. Still, in the simpler case, the default behavior at interface S can be described by the usual dummy converter honSrv with the addition that it always inputs "allow" to the resource upon an audit request. We do not assign it a new name as it is clear from the context.

7.1 Making Authentic Server-Memory Resources Auditable

We now describe a straightforward way to achieve an auditable and authentic (or confidential) server-memory resource from an authentic (or confidential) server-memory resource. We again denote the storage content as $F = (F_1, \dots, F_\ell)$, with $F_i \in \Sigma$. The main idea is to encode the entire storage F (for example an entire backup file) using an erasure code to tolerate a certain fraction of deleted symbols F_i. The audit consists sampling a sufficient number of random positions of the encoded version of F and to correctly decide whether the information on the server is sufficient to decode the file. This straightforward idea has been studied before, for example in [52,36], and we briefly show how this idea is implemented in our model.

Assumed and constructed resource. The assumed resource is an authenticated server-memory resource of size ℓ' (which is used to store a single file consisting of ℓ blocks) and alphabet Σ. The system achieved is an auditable and authenticated server-memory resource with alphabet Σ and ℓ locations.

Resource $\mathbf{sSMR}^{k,t_{\mathrm{rep}},\tau,\mathsf{audit}}_{\Sigma,n}$

Secure server-memory resources are augmented with a new client capability as follows:

Interfaces $\mathsf{C}_r, r \in \{1, \ldots, k\}$

```
Input: (read, $i$) $\in [n]$
  Defined as for sSMR

Input: (write, $i, x$) $\in [n] \times \Sigma$
  Defined as for sSMR

Input: audit :
  if ACTIVE and not INTRUSION then
      output auditReq at $\mathsf{S}_H$
      Let $d \in \{\mathrm{allow}, \mathrm{abort}\}$ be the returned value from $\mathsf{S}_H$
      if $d = \mathrm{allow}$ then
          $p \leftarrow 1 - \prod_{i=1}^{n}(1 - \alpha^{t_{\mathrm{rep}} - E_i})$, where $E_i := |\{j \mid j \in [t_{\mathrm{rep}}] \text{ and } \mathbb{M}[i,j] = \epsilon\}|$
          ($p$ is the probability that the value of at least one cell cannot be
          obtained based on the current state of the resource.)
          if $p \leq \tau$ then
              return accept
          else
              return reject
      else
          return reject
```

Fig. 16. Description of the auditable secure server-memory resources (only the difference to the secure server memory is shown).

The protocol. We now describe the protocol in more detail by specifying the two client converters ecInit (for initialization) and $\mathsf{ecAudit}$ (to implement the audits). We note that the default server behavior for this section equals the dummy one (that never deletes anything and allows all audit requests).

In the sequel, let $(\mathsf{enc}, \mathsf{dec})$ be an (ℓ', ℓ, d) erasure code. On input `init` to ecInit, the converter calls `init` of its connected resource and computes the encoding $F' \leftarrow \mathsf{enc}(\lambda^{\ell}) \in \Sigma^{\ell'}$, sets the counter ctr to 0. ctr can be seen as the version number or identifier of the currently stored memory content. Finally, the converter stores at each location $i \in [\ell']$ of $\mathbf{aSMR}^{k}_{\Sigma,\ell'}$ the pair $(F'_i, 0)$.

On $(\texttt{read}, i)$ to either ecInit or $\mathsf{ecAudit}$, the converter retrieves the whole memory content via $(\texttt{read}, i)$ requests and obtains for each cell either a pair (v_i, ctr') or the error symbol ϵ. If $ctr' \neq ctr$ (version mismatch) or if ϵ was returned, set $\bar{F}_i \leftarrow \perp$. Otherwise set $\bar{F}_i \leftarrow v_i$. If $|\{i \in [\ell'] \mid \bar{F}'_i = \perp\}| \geq d$, then output ϵ at the outer interface, otherwise, compute $F \leftarrow \mathsf{dec}(\bar{F}')$, where $\bar{F}' = (\bar{F}'_1, \ldots, \bar{F}'_{\ell'})$, and output F_i.

On $(\texttt{write}, i, F'_i)$, where $F'_i \in \Sigma$, to either ecInit or $\mathsf{ecAudit}$, the converter first executes the same instructions as on input $(\texttt{read}, i)$ to retrieve the currently (outsourced) storage content F. If and only if the F is successfully retrieved, the converter increments ctr, updates the single location $F_i \leftarrow F'_i$ and re-encodes the new memory content F' as $\bar{F}' \leftarrow \mathsf{enc}(F')$ and finally outputs $(\texttt{write}, i, (\bar{F}'_i, ctr))$ for all $i = 1 \ldots \ell'$.[10]

Finally, on a query $(\texttt{audit})$ to converter $\mathsf{ecAudit}$, the converter chooses a random subset $S \subseteq [\ell']$ of size t and outputs $(\texttt{read}, i)$ to $\mathbf{aSMR}$ for each $i \in S$ to retrieve the memory content at that location. If and only if all read instructions for $i \in S$ returned a pair (and not ϵ) and the counter of all pairs are equal to the locally stored value ctr, then output `success`.

The security of this scheme follows from the following theorem.

Theorem 5. *Let $\ell, \ell', d \in \mathbb{N}$. Let* $(\mathsf{enc}, \mathsf{dec})$ *be an (ℓ', ℓ, d)-erasure-coding scheme for alphabet Σ and error symbol $\perp$ and let ρ be the minimum fraction of blocks needed to recover the file, i.e., let $\rho = 1 - \frac{d-1}{\ell'}$.* [11] *Then the above protocol* $\mathsf{ecCheck} := (\mathsf{ecInit}, \mathsf{ecAudit}, \ldots, \mathsf{ecAudit})$ *(with*

[10] If the code would additionally support local updates and local decoding then reading and writing could be implemented more efficiently.

[11] For example, for $\ell' > \ell$ and alphabet $\Sigma = \mathbb{F}_{q > \ell'}$ the systematic Reed-Solomon code over Σ has $d = \ell' - \ell + 1$ and thus $\rho = \frac{\ell}{\ell'}$.

k copies of $\mathsf{ecAudit}$*) that chooses a random subset of size t during the audit, constructs the auditable server-memory resource $\mathbf{aSMR}^{k,\mathsf{audit}}_{\Sigma^{\ell},1}$ from the authentic server-memory resource $\mathbf{aSMR}^{k}_{\Sigma,\ell'}$, with respect to the simulator* sim_{ec} *(described in the proof of the theorem) and the pair* $(\mathsf{honSrv}, \mathsf{honSrv})$*. More specifically, for all distinguishers $\mathbf{D}$ performing at most q audits,*

$$\Delta^{\mathbf{D}}(\mathsf{honSrv}^{\mathsf{S}}\ \mathsf{ecCheck}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\Sigma,\ell'}, \mathsf{honSrv}^{\mathsf{S}}\mathbf{aSMR}^{k,\mathsf{audit}}_{\Sigma,\ell}) = 0$$
$$\textit{and} \qquad \Delta^{\mathbf{D}}(\mathsf{ecCheck}_{\mathcal{P}}\,\mathbf{aSMR}^{k}_{\Sigma,\ell'}, \mathsf{sim}^{\mathsf{S}}_{ec}\mathbf{aSMR}^{k,\mathsf{audit}}_{\Sigma,\ell}) \leq q \cdot \rho^{t}.$$

Proof (Sketch). Assume that a fraction α of cells of the real world authentic server memory have been deleted such that a $\beta := 1 - \alpha$ fraction is still available. A standard bound for binomial coefficients assures that the probability of selecting a subset of only good cells during an audit is $\frac{\binom{\beta \cdot m}{|S|}}{\binom{m}{|S|}} \leq \beta^{|S|}$. In the bad case where decoding would not be possible, i.e., if $\beta < \rho$, we see that for an arbitrary distinguisher $\mathbf{D}$ in the real setting, the probability that the audit succeeds is no larger than $\rho^{|S|}$.

We only prove the security condition and describe the simulator sim_{ec}. It internally maintains a simulated real-world view on the most recent memory content of size ℓ' by setting the memory content of simulated location i to ϵ on a $(\mathtt{delete}, i)$ instruction from the distinguisher (and updating the memory content to the last value written in case of a $(\mathtt{restore}, i)$ command. The simulator further maintains a simulated history, which is built based on the history of the ideal resource $\mathbf{aSMR}$ as follows: upon an audit request, the appropriate number of read-instructions are added to the simulated history (for each read request to deleted location the entry $(t, \mathtt{Fail}, i)$ is added). For each entry $(t, \mathtt{R}, i)$ or $(t, \mathtt{Fail}, i, F_i)$ in the ideal world history, the simulator replaces this entry by ℓ' read instructions in its own simulated history (for each read request to deleted location the entry $(t, \mathtt{Fail}, i)$ is added). An entry $(t, \mathtt{W}, i, F_i)$ (indicating a successful write operation) is replaced by ℓ' read-instructions (to all simulated locations) and ℓ' write instructions where each write instructions writes the pair $(\bar{F}_i, ctr)$ consisting of one symbol of the encoded and updated version of the memory content F together with the current counter ctr which is increased on each successful write operation.

If, after a deletion command, the number of simulated memory locations, that are equal to ϵ or associated with a too small counter, exceeds $d - 1$, the simulator deletes all memory locations of the ideal resource. Similarly, after a restore-command (which restores the last valid value stored by the client at that location), if the number of invalid memory locations (including wrong counter values) drops below $d - 1$, the simulator restores the entire memory content of the ideal resource.

On an audit request, the simulator simulates the random locations that are probed and evaluates if the test succeeds. If so, it allows the resource to output the right result to the client, and otherwise it instructs the resource to output $\mathtt{reject}$. The simulation is perfect up to the point where the following event BAD happens: *An audit succeeds when more than $d - 1$ locations are invalid.* The probability of distinguishing can hence be upper bounded by the probability that event BAD happens in an execution. To see this, in case $\neg\mathsf{BAD}$, the whole (logical) memory content is intact as long as there are no more than $d - 1$ deletions (or invalid counter values) to the real or simulated memory content. When the client initiates an audit, the simulator simulates the execution on its simulated memory which has the same distribution as in the real world and hence the probability to succeed is the same. In case of an unsuccessful (real or simulated) audit both, the ideal and the real system output $\mathtt{reject}$. In case the check succeeds, both resources output $\mathtt{success}$ and the whole memory can be retrieved: either the simulator has not deleted the memory contents in this case, or, in the real system, less than $d-1$ locations are invalid such that decoding is successful. The statement follows. □

7.2 Making Secure Server-Memory Resources Auditable

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 retrievablity 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}.

Assumed and constructed resource. The desired resource is an auditable secure server-memory resource of size ℓ and with retrievability guarantee τ. Recall that if an audit is successful, it means that the probability that any memory location is not accessible any more is smaller than τ. The assumed resource is a secure server-memory resource with replication t_{rep} and size $\ell + t_{\mathrm{audit}}/t_{\mathrm{rep}}$ whose values are determined below.

The protocol. As before, the protocol consists of the converters $\mathsf{statInit}$ (initialization), $\mathsf{statAudit}$ (client), and honest server behavior $\mathsf{statSrvAudit}$. 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 $\mathsf{statInit}$, the converter calls `init` and sets $\textsc{flag} \leftarrow 0$. The variable $\textsc{Flag}$ 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 $(\mathtt{read}, i)$ to either $\mathsf{statInit}$ or $\mathsf{statAudit}$, the converter outputs $(\mathtt{read}, i)$ 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 $\textsc{flag} \leftarrow 1$. [12] On $(\mathtt{write}, i, v)$ to either $\mathsf{statInit}$ or $\mathsf{statAudit}$, the converter outputs $(\mathtt{write}, i, v)$ to write the value v at location i of the server. Again, if an error is observed, it sets $\textsc{flag} \leftarrow 1$. Finally, on input `audit` to converter $\mathsf{statAudit}$, the converter immediately returns `reject` if $\textsc{flag} = 1$. If $\textsc{Flag} = 0$ the audit is executed as follows:[13], the converter issues t_{rep} read requests to each logical memory location $r = \ell + 1, \ldots, \ell + \frac{t_{\mathrm{audit}}}{t_{\mathrm{rep}}}$. 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 $\textsc{flag} \leftarrow 1$. [14] The security of this scheme follows from the following theorem.

Theorem 6. *Let Σ be an alphabet, let $\ell, \kappa, t_{rep}, t_{audit}, d \in \mathbb{N}$ such that $d = \frac{t_{audit}}{t_{rep}}$, and let $\rho, \tau \in (0,1)$ such that*

$$t_{rep} > \frac{\log(\tau) - \log(\ell)}{\log(\rho)}, \qquad t_{audit} > \frac{-\kappa}{\log(1-\rho)}. \tag{3}$$

The above described protocol $\mathsf{statCheck} := (\mathsf{statInit}, \mathsf{statAudit}, \ldots, \mathsf{statAudit})$ (with k copies of $\mathsf{statAudit}$) parameterized by t_{audit}, constructs the auditable secure server-memory resource $\mathbf{sSMR}^{k,t_{rep},\tau,\mathsf{audit}}_{\Sigma,\ell}$ from the secure server-memory resource $\mathbf{sSMR}^{k,t_{rep}}_{\Sigma,\ell+d}$ and a local memory

[12] One could relax this by introducing a tolerance *tol* and requiring that $\textsc{flag} \leftarrow 1$ only if more than *tol* of the t_{rep} copies of a memory location failed. Our results formalize zero tolerance and where t_{rep} is the minimal number required to obtain the desired retrievability guarantee.

[13] From a statistical point of view, if $\textsc{flag} = 0$, we have t_{audit} independent samples to estimate the parameter α.

[14] One could again introduce a tolerance *tol* and require that $\textsc{flag} \leftarrow 1$ only if more than *tol* samples returned an error. Our results formalize zero tolerance and t_{audit} is the minimal number of samples required to get a sufficiently accurate estimate of α.

(which stores the variable FLAG*), with respect to the simulator* sim_{stat} *(described in the proof) and the pair* $(\mathsf{honSrv}, \mathsf{honSrv})$*. More specifically, for all distinguishers* $\mathbf{D}$ *performing at most* q *audits,*

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

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

⌜ As a numerical example, let us assume we are given a secure memory that can store one tebibyte of data, with a certain replication factor t_{rep}, and where each element of Σ represents a block of size 16 kibibytes. This yields $\ell = 2^{26}$. For a security level $\kappa = 128$, $\tau = 2^{-32}$, and $\rho = 2^{-9}$ we would need a replication factor of $t_{\text{rep}} \approx 6$ and the total size of retrieved data during an audit is roughly 700 mebibytes. Different applications can adjust these parameters according to their preference in order to trade security, storage overhead, and access time. ⌟

Proof. We start by describing the simulator. sim_{stat} internally maintains a simulated history, which is identical to the history of the ideal resource but where for each audit request, the appropriate number of read-requests are added. It further maintains a value FLAG which is initially 0.

On input $(\texttt{pollute}, \alpha)$ and $(\texttt{reducePollution}, \delta)$ it forwards this query to the ideal resource $\mathbf{sSMR}^{k,t,\tau,\mathsf{audit}}_{\Sigma,\ell}$. On input `getHist` the simulator reads the history of the ideal resource and updates its simulated history appropriately and returns it to $\mathbf{D}$. If at any time, the simulated history contains an entry $(k, \texttt{Failed}$ for some k, then FLAG is set to 1.

On an audit-request, the simulator first updates its simulated history. Then, it replies abort to $\mathbf{sSMR}^{k,t,\tau,\mathsf{audit}}_{\Sigma,\ell}$ in case FLAG $= 1$. In case FLAG $= 0$, sim_{stat} simulates t_{audit} read accesses, such that each access fails with probability α. In case there are failures, sim_{stat} outputs abort to the ideal resource to provoke a reject and internally sets FLAG $\leftarrow 1$. It further adds the appropriate entries $(k, \texttt{Failed})$ and $(k, \texttt{Access})$ to its simulated history.

The remainder of the proof proceeds in two steps. First, we bound the probability that a simulated audit succeeds (if FLAG $= 0$), although α is larger than the threshold ρ, by $2^{-\kappa}$. A straightforward statistical argument shows that as long as $\alpha < \rho$ (and FLAG $= 0$), the probability that after the audit (and before the next intrusion phase) retrieving any cell would result in a failure is smaller than τ. Hence, the probability of an imperfect simulation is negligible in κ.

We first compute the probability that an audit is passed in case $\alpha > \rho$ and FLAG $= 0$. let $X = \sum_{i=1}^{t_{\text{audit}}} X_i$, where X_i are independently distributed according to $X_i \sim \mathsf{Bernoulli}(\alpha)$.

$$\mathsf{Pr}[X = 0] = (1-\alpha)^{t_{\text{audit}}} \leq (1-\rho)^{t_{\text{audit}}} \leq 2^{-\kappa},$$

where we used the assumption that $t_{\text{audit}} > \frac{-\kappa}{\log(1-\rho)}$. We conclude that except with negligible probability, the audit only succeeds if $\alpha < \rho$.

Assuming $\alpha < \rho$, we prove that the probability that any value is not recoverable (given that FLAG $= 0$) is smaller than τ for the choices in Equation 3. Again, each read request would fail independently with probability α. Using Bernoulli's inequality, we get

$$1 - (1 - \alpha^{t_{\text{rep}}})^{\ell} \leq 1 - (1 - \rho^{t_{\text{rep}}})^{\ell} \leq 1 - (1 - \ell\rho^{t_{\text{rep}}}) = \ell \cdot \rho^{t_{\text{rep}}},$$

and by the theorem assumption $t_{\text{rep}} > \frac{\log(\tau)-\log(\ell)}{\log(\rho)}$, we immediately have

$$\ell \cdot \rho^{t_{\text{rep}}} \leq \ell \cdot \rho^{\frac{\log(\tau)-\log(\ell)}{\log(\rho)}} = \ell \cdot \rho^{\frac{\log(\tau)}{\log(\rho)}} \cdot \rho^{\frac{\log(-\ell)}{\log(\rho)}} = \ell \cdot \tau \cdot \ell^{-1} = \tau.$$

This concludes the proof. □

Auditing via a direct estimate of $\boldsymbol{\alpha}$. We can replace the audit of protocol $\mathsf{statCheck}$ by a direct estimation of the parameter α. In case that the estimation $\bar{\alpha}$ is sufficiently accurate, say up to $\frac{\varrho}{2}$ with very high probability, verifying that $\bar{\alpha} < \frac{\varrho}{2}$ is sufficient to obtain the desired retrievability guarantee and the audit returns `success`. The audit itself consists of obtaining t_{audit} independent samples via read-requests to the d extra locations of the secure server-memory resource. In case it is not possible to obtain that many samples, for example because certain locations failed during the last audit and would therefore output ϵ with probability 1 instead of α, the audit returns `reject`. If t_{audit} samples can be obtained, let n_e be the number of errors that occurred and define the estimate $\bar{\alpha} \leftarrow \frac{n_e}{t_{\text{audit}}}$. From the Chernoff-Hoeffding bound, we get that for arbitrary $\theta, \delta \in (0,1)$, if $t_{\text{audit}} > \frac{2+\theta}{\theta^2} \cdot \log(\frac{2}{\delta})$ independent samples of a Bernoulli distribution with parameter α are obtained, the probability that $\bar{\alpha} \in [\alpha - \theta, \alpha + \theta]$ is at least $1 - \delta$. Hence, setting $\theta < \frac{\varrho}{2}$ and $\delta \leq 2^{-\kappa}$, we get an analogous result to the one above, but in general with higher values for t_{audit}.

On composing the previous modular steps. It is instructive to wrap up the results until this point: We have shown how to construct an auditable secure server-memory resource from a secure server-memory resource by estimating the failure probability α. In Sect. 5.3, we have shown how to construct a secure server-memory resource from an authentic and confidential server-memory resource, which itself can be constructed from an insecure server-memory as shown in Sect. 5.1 and Sect. 5.2. We can invoke the composition theorem of constructive cryptography to conclude that the composition of all protocols constructs an auditable secure server-memory resource from an insecure one (and local storage). The composed protocol has strong security guarantees and is robust against an arbitrary number of failures and is comparable to existing schemes in terms of access times to read and write single data blocks. However, the gained security comes at the price of a theoretically larger server-side memory consumption due to replication, and higher audit times.

7.3 Revisiting the Hash-Based Challenge-Response Approach

Our model allows to formalize the security guarantees of a very simple hash-based challenge-response protocol that is often given as an introductory example to proofs of retrievability, but, to the best of our knowledge, lacks a formal security statement in other models. In a nutshell, the retrievability test asks the server to deliver the correct hash value of the (current) storage content concatenated with a uniform random challenge provided (and precomputed) by the client. The intuitive security claim is that the server cannot have modified or deleted the content before answering the challenge. For the sake of concreteness, we consider the setting where one client stores a single file F (modeled as a sequence of bits in this paragraph) on an insecure server memory and would like to audit this file at a later point in time. We assume an (ideal) hash function, i.e., a random oracle, $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^r$ following the notation of [7] and denote by $x||y$ the concatenation of bitstrings x and y.

Assumed and constructed resource. We assume a random oracle, $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^r$, which is made available to the parties by means of a resource $\mathbf{H}$ that has an interface for the client and one for the server: On input $(\texttt{eval}, x)$ at any of its interfaces $\mathbf{H}$ returns $\mathsf{H}(x)$ at the same interface. We further assume a small local storage and a communication channel between client and server, which we denote by $\mathbf{Ch}$. Formally, $\mathbf{Ch}$ is a system with the two interfaces C_1 and S, such that whatever is input at one interface is output at the other interface. We refer to [43,17] for a possible channel formalization. Last but not least, we assume an ordinary insecure memory resource $\mathbf{SMR}^1_{\Sigma,\ell+\kappa}$, where $\Sigma = \{0,1\}$ and κ being the size of the challenge c. The desired functionality we want to achieve is the auditable insecure memory resource $\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma,\ell}$.

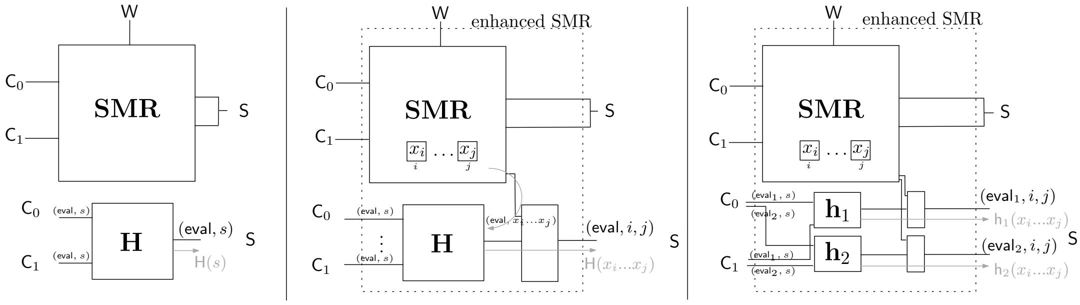

Fig. 17. *Left:* Real system with unrestricted server access to the random oracle. The challenge-response protocol, executed in this setting, is not secure. *Center:* Real system with restricted server access. Under this stronger assumption, the challenge-response protocol is secure. *Right:* Real system with restricted access to two ideal compression functions. In this setting, the challenge-response protocol, where the hash is computed using a secure iterated construction like NMAC, is not secure in general.

The protocol. We now describe the protocol in more detail: As usual, we specify an initialization converter $\mathsf{hashInit}$, a client converter $\mathsf{hashAudit}$, and the protocol for the honest server behavior $\mathsf{srvHash}$. On input $\texttt{init}$ to $\mathsf{hashInit}$, the converter simply calls $\texttt{init}$ of its connected resource. On $(\texttt{write}, 1, F)$ to either $\mathsf{hashInit}$ or $\mathsf{hashAudit}$, 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 $y \leftarrow \mathsf{H}(F||c)$ and stores c and y in the local storage. On $(\texttt{read}, 1)$ to either $\mathsf{hashInit}$ or $\mathsf{hashAudit}$, the converter retrieves the content of the memory and outputs the first ℓ bits of the received content. Finally, on a query $(\texttt{audit})$ to converter $\mathsf{hashAudit}$, if there is a challenge stored in local memory, the protocol writes c to the server memory at locations $\ell+1 \ldots \ell+\kappa$ and sends a notification $\texttt{auditReq}$ to the server via the bidirectional channel. On receiving a response y' on that channel from the server, the client protocol outputs $\texttt{success}$ if and only if $y = y'$. In any case, the challenge c is deleted from the local storage. The next audit is only possible after executing a new write-query.[15] Finally, the server protocol $\mathsf{srvHash}$, upon receiving an audit-request, simply evaluates H on the current memory contents and sends the result to the client via the bidirectional channel.

Insecurity of the approach. Unfortunately, the security of the hash-based challenge-response protocol above does not follow solely based on the random oracle assumption. The proof of this follows closely the intuition that in a composable security framework, the environment "knows" the content of the server-memory resource. Hence, if the random oracle can be queried by the distinguisher on an arbitrary input, i.e., not restricted to the actual value stored in the server memory, it can always be queried on the correct input, irrespective of the actual content of the server-memory resource. This is formalized in the following lemma.

Lemma 2. *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 protocol above (specified by the converters* $\mathsf{hashInit}, \mathsf{hashAudit}, \mathsf{srvHash}$*) does not provide a secure proof of storage. More specifically, there is a distinguishing strategy such that for any simulator* sim *it holds that*

$$\Delta^{\mathbf{D}}(\mathsf{hashInit}^{\mathsf{C}_0}\mathsf{hashAudit}^{\mathsf{C}_1}[\mathbf{L}, \mathbf{Ch}, \mathbf{SMR}^1_{\Sigma,\ell'}, \mathbf{H}], \mathsf{sim}^{\mathsf{S}}\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}) = 1.$$

Proof. To prove the statement, we describe the random experiment between a particular distinguisher $\mathbf{D}$ and the system $\mathbf{T}$, which either corresponds to the real system with the protocols attached, i.e., $\mathsf{hashInit}^{\mathsf{C}_0}\mathsf{hashAudit}^{\mathsf{C}_1}[\mathbf{L}, \mathbf{Ch}, \mathbf{SMR}^1_{\Sigma,\ell'}, \mathbf{H}]$ or to the ideal system with the simulator attached, i.e., $\mathsf{sim}^{\mathsf{S}}\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}$. First, $\mathbf{D}$ inputs $\texttt{init}$ at interface C_0 and queries,

[15] We assume that the client protocol rejects an audit if no challenge is stored in local memory. Note that one could of course prepare more challenges.

for some arbitrary file $F \neq 0^\ell$, $(\texttt{write}, 1, F)$ and inputs $\texttt{initComplete}$ at interface C_0. As the next step, $\mathbf{D}$ inputs $\texttt{startWriteMode}$ at interface W and subsequently instructs the resource to delete the file by storing the all-zero string via queries $(\texttt{write}, i, 0)$ for all locations $i \in [\ell]$ at interface S_I and finally inputs $\texttt{stopWriteMode}$ at interface W. $\mathbf{D}$ then inputs $\texttt{audit}$ at interface C_1 to receive a challenge c.[16] $\mathbf{D}$ then queries $\mathbf{H}$ on input $F||c$ to receive the value y_0 and sends y_0 back to the client. As the last step the client retrieves the actual storage content by querying $(\texttt{read}, 1)$ at interface C_1. Let the returned file be F'. Finally, the distinguisher outputs 1 if and only if the audit is successful and F' is the all-zero bitstring. It is obvious that if $\mathbf{D}$ is querying the real system (with the protocol), then its output is 1 with certainty. However, in the ideal system, if the server memory content when the audit start is $F' \neq F$, then the ideal audit, by definition cannot be successful, irrespective of the simulator's actions. The distinguisher outputs 1 with probability zero in that case. The statement follows. □

Security under stronger assumptions. In this paragraph, we show that the additional assumption we have to make in order for the scheme to be secure, is to restrict adversarial random oracle evaluations by allowing inputs from the server storage only. In particular, the server is only allowed to query the random oracle via calls $(\texttt{eval}, i, j)$, $i \leq j$, and to obtain the hash value $\mathsf{H}(\mathbb{M}[i]||\ldots||\mathbb{M}[j])$ as opposed to receiving hash values for arbitrary bitstrings. See also Fig. 17.

To turn this intuition into a formal statement, we consider the following functionality $\mathbf{SMR}^k_{\mathsf{H},\Sigma,\ell}$ which basically behaves like $\mathbf{SMR}^k_{\Sigma,\ell}$, but with two additional capabilities: Each client interface can, aside of ordinary read- and write-request, query $(\texttt{eval}, x)$ upon which the resource provides $\mathsf{H}(x)$ as output. Second, the server gets access to the random oracle via its interface, and is restricted to submit queries of the form $(\texttt{eval}, i, j)$ with $i \leq j$, and the resource returns the result of $\mathsf{H}(\mathbb{M}[i]||\ldots||\mathbb{M}[j])$ to the server. We prove the following theorem.

Theorem 7. *Let $\ell, \ell', \kappa, n \in \mathbb{N}$, with $\ell' = \ell + \kappa$, let $\Sigma := \{0, 1\}$, and let H denote a hash function modeled as a random oracle. The above described protocol* $(\mathsf{hashInit}, \mathsf{hashAudit})$ *constructs the auditable server-memory resource $\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}$ from the server-memory resource $\mathbf{SMR}^1_{\mathsf{H},\Sigma,\ell'}$, a local memory (of constant size), and a channel, with respect to the simulator sim_{hash} (described in the proof) and the pair* $(\mathsf{srvHash}, \mathsf{honSrv})$. *More specifically, for all distinguishers $\mathbf{D}$ asking at most q queries*

$$\Delta^{\mathbf{D}}(\mathsf{hashInit}^{\mathsf{C}_0}\mathsf{hashAudit}^{\mathsf{C}_1}\mathsf{srvHash}^{\mathsf{S}}[\mathbf{L}, \mathbf{Ch}, \mathbf{SMR}^1_{\mathsf{H},\Sigma,\ell'}], \mathsf{honSrv}^{\mathsf{S}}\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}) = 0$$

$$\textit{and} \qquad \Delta^{\mathbf{D}}(\mathsf{hashInit}^{\mathsf{C}_0}\mathsf{hashAudit}^{\mathsf{C}_1}[\mathbf{L}, \mathbf{Ch}, \mathbf{SMR}^1_{\mathsf{H},\Sigma,\ell'}], \mathsf{sim}^{\mathsf{S}}_{hash}\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}) \leq q \cdot 2^{-\kappa} + 2^{-r}.$$

Proof (Sketch). Since it is obvious that when the server is honest, the audit succeeds, we directly proceed to prove the security of the construction. We first describe the straightforward simulation. On any query by the distinguisher to read or write directly into the storage via server interface S_I, the simulator simply forwards this request to $\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}$. If the distinguisher inputs the query $(\texttt{eval}, i, j)$ for $1 \leq i \leq j \leq \ell + \kappa$, the simulator computes the string $s \leftarrow \mathbb{M}'[i]||\ldots||\mathbb{M}'[j]$, and if there is no internally stored pair (s, y) for this string, choose $y \twoheadleftarrow \{0, 1\}^r$ and store (s, y) internally for future reference. Finally, output y as the (simulated) random oracle output to the distinguisher.

Last but not least, when the client starts the first audit for the most recent uploaded file (note that by definition such a requests occurs only in a phase where no intruder is active), the simulator internally chooses a challenge $c \twoheadleftarrow \{0, 1\}^\kappa$. Then, the simulator retrieves the history of the resource to check which file F was written to the storage and defines $s \leftarrow F||c_1||\ldots||c_\kappa$ and checks whether there is a recorded pair (s, y). Only if none is recorded,

[16] We assume that in both worlds a challenge is output as otherwise distinguishing is trivial.

chooses $y_0 \twoheadleftarrow \{0,1\}^r$ and store the pair (s, y_0). Finally, sim_{hash} stores c at locations $\mathbb{M}'[\ell+1]$ to $\mathbb{M}'[\ell+\kappa]$ of its simulated memory. After this, it outputs the notification `auditReq` to the distingusiher (as coming from the bidirectional channel). If the distinguisher's response to this audit request equals y_0, then the simulator outputs allow and otherwise output abort to $\mathbf{SMR}^{1,\mathsf{audit}}_{\Sigma^\ell,1}$.

We now consider the an execution of a distinguisher with either the real system or the ideal system. On an audit-request by the client, which happens only in a phase where the distinguisher is not allowed to write to the server-memory resource, the client reveals the challenge c by writing it into the server storage. Let us denote the current server storage at this point as R. We observe that the server is only capable of evaluating the random oracle on $R||c$ or on $F||c$ by restoring the original memory cell. Hence, assume that the distinguisher does not restore and the memory content is $R \neq F$. Hence, in both worlds, the distinguisher does not learn receive the value $y_0 = \mathsf{H}(F||c)$ at this point. In particular, the probability of guessing the correct hash output given that $\mathbf{D}$ has never evaluated the random oracle on $s = F||c$ is 2^{-r}. Furthermore, the probability that the distinguisher has ever evaluated the random oracle on $F||c$ before the audit was initiated, is no larger than $q \cdot 2^{-\kappa}$. We observe that if none of these two events occur, then the real and ideal systems behave identically. Indeed, a retrievability check is passed in both worlds if and only if the memory content is the original file F and the distinguisher sends the correct hash value $y_0 = \mathsf{H}(F||c)$ to the client. The statement follows. □

On replacing the monolithic random oracle. We consider iterated constructions of random oracles $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^r$ from ideal compression functions $\mathsf{h}_1 : \{0,1\}^\kappa \times \{0,1\}^n \mapsto \{0,1\}^n$ and $\mathsf{h}_2 : \{0,1\}^n \mapsto \{0,1\}^r$ $(n, r, \kappa > 0)$ from [22]. We formally show that in our setting, the fact that an iterated construction realizes a random oracle does not imply that the iterated construction realizes resource $\mathbf{SMR}^1_{\mathsf{H},\Sigma,\ell'}$ from resource $\mathbf{SMR}^1_{\mathsf{h}_1,\mathsf{h}_2,\Sigma,\ell}$. In other words, even if the server access to the functions is restricted as required above by Theorem 7 and illustrated in Fig. 17, their applicability is not generally safe in the context of audits. This observation meets our intuition and has already been observed in [51]. The intuitive reason why it fails is that certain constructions (like NMAC) allow the server to compute a result in multiple stages, such that he can store an intermediate result, ignore the original memory, and still compute the correct hash value.

For simplicity, we focus on the NMAC construction that was shown to securely realize a random oracle H from two ideal compression functions h_1 and h_2 [22]. The input is a file $F = (F_1||F_2||\ldots||F_\ell)$ of $\ell = \kappa \cdot l$ bits, let us denote the ith block (having κ bits) as F^i. Let $y_0 \leftarrow 0^n$ be the initial block.[17]. Compute for each block i from 1 to l, $y_i \leftarrow \mathsf{h}_1(F^i, y_{i-1})$. Finally, compute and return $Y \leftarrow \mathsf{h}_2(y_l)$ as $\mathsf{H}(F)$. Let nmac_{cli} be the client converter that simply relays all queries and responses not concerning the random oracle evaluations and on input $(\mathtt{eval}, x)$ at its outer interface, computes the hash value according to the NMAC construction above. Similarly, the honest server converter nmac_{srv} simply relays all queries and responses not concerning the random oracle evaluations and on input $(\mathtt{eval}, i, j)$ at the outer interface evaluates the NMAC construction using the appropriate instructions to the resource.

Lemma 3. *Let $\ell, n, r, \kappa > 0$ be integers such that $\ell > n + \kappa$, and let $\Sigma := \{0,1\}$. Let H denote a random oracle and h_1 and h_2 be ideal compression functions as introduced above. Then, the client protocol $(\mathsf{nmac}_{cli}, \mathsf{nmac}_{cli})$ described above does not construct the system $\mathbf{SMR}^1_{\mathsf{H},\Sigma,\ell}$ from system $\mathbf{SMR}^1_{\mathsf{h}_1,\mathsf{h}_2,\Sigma,\ell}$ if the server is possibly dishonest. In particular, there is a distinguishing strategy such that for any simulator sim, making at most q random oracle queries, it holds that*

$$\Delta^{\mathbf{D}}(\mathsf{nmac}^{\mathsf{C}_0}\mathsf{nmac}^{\mathsf{C}_1}\mathbf{SMR}^1_{\mathsf{h}_1,\mathsf{h}_2,\Sigma,\ell}, \mathsf{sim}^{\mathsf{S}}\mathbf{SMR}^1_{\mathsf{H},\Sigma,\ell}) \geq 1 - q \cdot 2^{-\kappa} - 2^{-r}.$$

[17] We assume this initial value to be prepended to F such that the computation formally gets $F' = 0^n||F$ as the only input. This is only a syntactic simplification

Proof (Sketch). We describe a distinguisher $\mathbf{D}$ that interacts either with the ideal world $\mathsf{sim}^{\mathsf{S}}\mathbf{SMR}_{\mathsf{H}}$ or with the real world $\mathbf{SMR}^{1}_{\mathsf{h}_1,\mathsf{h}_2,\Sigma,\ell}$. The distinguisher first chooses a uniform random bitstring s of length $\ell - 1$ and stores $F := s||1$ in the memory and pre-computes $y := \mathsf{H}(F, c)$ for a uniformly random challenge c (of length κ) via an input $(\mathtt{eval}, F||c)$ at interface C_1. A a second step, $\mathbf{D}$ performs the "first stage" of the NMAC computation using the interface S: having obtained y_{i-1}, $\mathbf{D}$ writes this value back to an appropriate location, say j, of the server storage and queries $(\mathtt{eval}_1, j, j + \kappa + n)$ to receive the intermediate value y_i and proceeds to until obtaining y_ℓ. Finally, $\mathbf{D}$ sets the server memory to $y_l||0^{\ell-n}$ via a write-command at the server interface and then issues $\mathtt{stopWriteMode}$ at interface W which disallows adversarial write-access at interface S. Next, $\mathbf{D}$ writes the challenge c to the server storage via the client interface C_1 and completes the evaluation of NMAC by computing $y' \leftarrow \mathsf{h}_2(\mathsf{h}_1(1, \ldots, n||c))$ by appropriate evaluation queries at the server interface. To decide on its output bit, $\mathbf{D}$ reads the current content R of the memory at interface C and decides on 1 if and only if $R = y_l||c||0^{\ell-n-\kappa}$ and $y = y'$.

If $\mathbf{D}$ is connected to the real system then, by design of the experiment, the probability that $y = y'$ and $R \neq F$ is 1. The reason is that the memory content $y_l||c||0^{\ell-n-\kappa}$, is sufficient to compute the correct hash value $\mathsf{H}(F||c)$.

If $\mathbf{D}$ is connected to the ideal system, the probability that $y = y'$ and $R \neq F$ is significantly smaller and is based on the observation that the simulator can only compute $H(F||c)$ with non-negligible probability if it can evaluate the storage its random oracle on input $F||c$ which has to reside in memory. By the time the simulator learns c, he has already lost his write-access to the resource. Hence, the probability of $y = y'$ given that the storage content in the end of the experiment is $R \neq F||c$ is upper bounded by $q \cdot 2^{-\kappa} + \cdot 2^{-r}$. □

References

1. Androulaki, E., Cachin, C., Dobre, D., Vukolić, M.: Erasure-coded byzantine storage with separate metadata. In: International Conference on Principles of Distributed Systems. pp. 76–90. Springer (2014)
2. Apon, D., Katz, J., Shi, E., Thiruvengadam, A.: Verifiable oblivious storage. In: International Workshop on Public Key Cryptography. pp. 131–148. Springer (2014)
3. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Khan, O., Kissner, L., Peterson, Z.N.J., Song, D.: Remote data checking using provable data possession. ACM Trans. Inf. Syst. Secur. 14(1), 12 (2011)
4. 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)
5. Ateniese, G., Dagdelen, Ö., Damgård, I., Venturi, D.: Entangled cloud storage. Future Generation Computer Systems 62, 104–118 (2016)
6. Ateniese, G., Pietro, R.D., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. IACR Cryptology ePrint Archive 2008, 114 (2008)
7. Baecher, P., Fischlin, M.: Random oracle reducibility. In: Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, 2011. Proceedings. pp. 21–38. Springer (2011)
8. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Advances in Cryptology — CRYPTO '96: 16th Annual International Cryptology Conference, 1996 Proceedings. pp. 1–15. Springer (1996)
9. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12(2/3), 225–244 (1994)
10. Bowers, K.D., Juels, A., Oprea, A.: Proofs of retrievability: theory and implementation. In: CCSW. pp. 43–54 (2009)
11. Cachin, C.: Integrity and consistency for untrusted services. In: International Conference on Current Trends in Theory and Practice of Computer Science. pp. 1–14. Springer (2011)
12. Cachin, C., Dobre, D., Vukolić, M.: Separating data and control: Asynchronous bft storage with 2t+ 1 data replicas. In: Symposium on Self-Stabilizing Systems. pp. 1–17. Springer (2014)
13. Cachin, C., Geisler, M.: Integrity protection for revision control. In: International Conference on Applied Cryptography and Network Security. pp. 382–399. Springer (2009)

14. Cachin, C., Keidar, I., Shraer, A.: Fail-aware untrusted storage. SIAM Journal on Computing 40(2), 493–533 (2011)
15. Cachin, C., Shelat, A., Shraer, A.: Efficient fork-linearizable access to untrusted shared memory. In: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. pp. 129–138. ACM (2007)
16. Camenisch, J., Enderlein, R.R., Maurer, U.: Memory erasability amplification. In: International Conference on Security and Cryptography for Networks. pp. 104–125. Springer (2016)
17. 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)
18. Canetti, R., Dodis, Y., Pass, R., Walfish, S.: Universally composable security with global setup. In: Theory of Cryptography, 4th Theory of Cryptography Conference, 2007, Proceedings. pp. 61–85. Springer (2007)
19. Cash, D., Küpçü, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. In: Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2013. Proceedings. pp. 279–295. Springer (2013)
20. Chandran, N., Kanukurthi, B., Ostrovsky, R.: Locally updatable and locally decodable codes. In: Theory of Cryptography: 11th Theory of Cryptography Conference, 2014. Proceedings. pp. 489–514. Springer (2014)
21. Chung, K.M., Pass, R.: A simple oram. Tech. rep., DTIC Document (2013)
22. Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-damgård revisited: How to construct a hash function. In: Advances in Cryptology – CRYPTO 2005: 25th Annual International Cryptology Conference, 2005. Proceedings. pp. 430–448. Springer (2005)
23. Demay, G., Gaži, P., Hirt, M., Maurer, U.: Resource-restricted indifferentiability. In: Advances in Cryptology – EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2013. Proceedings. pp. 665–684. Springer (2013)
24. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion oram: A constant bandwidth blowup oblivious ram. In: Theory of Cryptography, 13th Theory of Cryptography Conference, 2016-A, Proceedings, Part II. pp. 145–174. Springer (2016)
25. Dodis, Y., Vadhan, S.P., Wichs, D.: Proofs of retrievability via hardness amplification. In: Theory of Cryptography: 6th Theory of Cryptography Conference, 2009. Proceedings. pp. 109–127. Springer (2009)
26. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: ACM Conference on Computer and Communications Security. pp. 213–222 (2009)
27. Fletcher, C., Naveed, M., Ren, L., Shi, E., Stefanov, E.: Bucket oram: single online roundtrip, constant bandwidth oblivious ram. Tech. rep., IACR Cryptology ePrint Archive, Report 2015, 1065 (2015)
28. Garay, J.A., MacKenzie, P., Prabhakaran, M., Yang, K.: Resource fairness and composability of cryptographic protocols. Journal of cryptology 24(4), 615–658 (2011)
29. Gaži, P., Maurer, U., Tackmann, B.: Environment-affected constructions. Manuscript (2016)
30. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing oram and using it efficiently for secure computation. In: International Symposium on Privacy Enhancing Technologies Symposium. pp. 1–18. Springer (2013)
31. Gentry, C., Halevi, S., Jutla, C., Raykova, M.: Private database access with he-over-oram architecture. In: International Conference on Applied Cryptography and Network Security. pp. 172–191. Springer (2015)
32. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. Journal of the ACM (JACM) 43(3), 431–473 (1996)
33. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious ram simulation. In: ICALP (2). pp. 576–587 (2011)
34. 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)
35. Hofheinz, D., Matt, C., Maurer, U.: Idealizing identity-based encryption. In: Advances in Cryptology – ASIACRYPT 2015: 21st International Conference on the Theory and Application of Cryptology and Information Security, 2015, Proceedings, Part I. pp. 495–520. Springer (2015)
36. Juels, A., Kaliski, B.S.: Pors: proofs of retrievability for large files. In: ACM Conference on Computer and Communications Security. pp. 584–597 (2007)

37. Keelveedhi, S., Bellare, M., Ristenpart, T.: Dupless: server-aided encryption for deduplicated storage. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 179–194 (2013)
38. Kupcu, A.: Efficient Cryptography for the Next Generation Secure Cloud. Ph.D. thesis (2010)
39. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious ram and a new balancing scheme. In: SODA. pp. 143–156 (2012)
40. Li, J., Krohn, M.N., Mazières, D., Shasha, D.: Secure untrusted data repository (sundr). In: OSDI. vol. 4, pp. 9–9 (2004)
41. Liu, B., Warinschi, B.: Universally composable cryptographic role-based access control. In: Provable Security: 10th International Conference, ProvSec 2016, 2016, Proceedings. pp. 61–80. Springer (2016)
42. Maurer, U.: Indistinguishability of random systems. In: Knudsen, L. (ed.) Advances in Cryptology — EUROCRYPT 2002: International Conference on the Theory and Applications of Cryptographic Techniques, 2002 Proceedings. pp. 110–132. Springer-Verlag (2002)
43. Maurer, U.: Constructive cryptography - a new paradigm for security definitions and proofs. In: TOSCA. pp. 33–56 (2011)
44. Maurer, U., Renner, R.: Abstract cryptography. In: Innovations in Theoretical Computer Science. pp. 1–21 (2011)
45. Maurer, U., Renner, R.: From indifferentiability to constructive cryptography (and back). In: Theory of Cryptography: 14th International Conference, 2016, Proceedings, Part I. pp. 3–24. Springer (2016)
46. Mazieres, D., Shasha, D.: Building secure file systems out of byzantine storage. In: Proceedings of the twenty-first annual symposium on Principles of distributed computing. pp. 108–117. ACM (2002)
47. Moataz, T., Mayberry, T., Blass, E.O.: Constant communication oram with small blocksize. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. pp. 862–873. ACM (2015)
48. Naor, M., Rothblum, G.N.: The complexity of online memory checking. J. ACM 56(1) (2009)
49. Ren, L., Fletcher, C.W., Yu, X., Kwon, A., van Dijk, M., Devadas, S.: Unified oblivious-ram: Improving recursive oram with locality and pseudorandomness. IACR Cryptology ePrint Archive 2014, 205 (2014)
50. Ren, L., Fletcher, C.W., Yu, X., Van Dijk, M., Devadas, S.: Integrity verification for path oblivious-ram. In: 2013 IEEE High Performance Extreme Computing Conference. Institute of Electrical and Electronics Engineers (IEEE) (2013)
51. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with composition: Limitations of the indifferentiability framework. In: Advances in Cryptology – EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2011. Proceedings. pp. 487–506. Springer (2011)
52. Shacham, H., Waters, B.: Compact proofs of retrievability. J. Cryptology 26(3), 442–483 (2013)
53. Shi, E., Stefanov, E., Papamanthou, C.: Practical dynamic proofs of retrievability. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. pp. 325–336. CCS '13, ACM (2013)
54. Shraer, A., Cachin, C., Cidon, A., Keidar, I., Michalevsky, Y., Shaket, D.: Venus: Verification for untrusted cloud storage. In: Proceedings of the 2010 ACM workshop on Cloud computing security workshop. pp. 19–30. ACM (2010)
55. Stefanov, E., van Dijk, M., Juels, A., Oprea, A.: Iris: a scalable cloud file system with efficient integrity checks. In: ACSAC. pp. 229–238 (2012)
56. 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, February 5-8, 2012 (2012)
57. 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 & communications security. pp. 299–310. ACM (2013)
58. Wang, X.S., Huang, Y., Chan, T.H., Shelat, A., Shi, E.: Scoram: oblivious ram for secure computation. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 191–202. ACM (2014)

A Further Details of Section 1

This first section of the supplementary material provides more details on the traditional security definition of Proofs of Retrievability (PoR).

A.1 Traditional PoR Game

Below we sketch the traditional retrievability game. The game involves the (malicious) server $\bar{S}$, the extractor $\mathcal{E}$, and the challenger and is defined as follows:

Retrievability Game $\mathsf{ExtGame}_{\bar{S},\mathcal{E}}(\nu, p)$

1.) $\bar{S}$ outputs a valid protocol sequence $P := (op_0, op_1, \ldots, op_q)$, i.e., a sequence of invocations where $op_0 = \mathsf{init}(1^\nu, \Sigma, n)$ and, for $j > 0$, $op_j \in \{\mathsf{read}(i), \mathsf{write}(i, v_i), \mathsf{audit}\}$ with $i \in [n]$ and $v_i \in \Sigma$. We denote by M the correct memory content after an ideal execution of the sequence P.

2.) The challenger creates an instance of an honest client $\mathcal{C}$ and executes the whole sequence P between $\mathcal{C}$ and $\bar{S}$. After the execution, let $\mathcal{C}_{\text{fin}}$ and $\bar{S}_{\text{fin}}$ denote the state of the client and server (including its random coins), respectively. We define the following probability over the random coins of the client during the audit:

$$\mathsf{Succ}\left(\bar{S}_{\text{fin}}\right) := \mathsf{Pr}\left[\bar{S}_{\text{fin}} \overset{\mathsf{audit}}{\longleftrightarrow} \mathcal{C}_{\text{fin}} \Rrightarrow \mathsf{accept}\right]$$

3.) Run $M' \leftarrow \mathcal{E}^{\bar{S}_{\text{fin}}}(\mathcal{C}_{\text{fin}}, 1^n, 1^p)$, where the extractor gets black-box rewinding access to the server strategy and in addition a description of the client strategy.

4.) If $\mathsf{Succ}\left(\bar{S}_{\text{fin}}\right) \geq \frac{1}{p}$ but $M' \neq M$, then output 1 (the server wins the game). Otherwise, output 0.

Fig. 18. One version of the traditional retrievability game.

B Further Details of Section 2

This section contains the details skipped in the preliminaries and provides more information on the type of resources we use in this work.

B.1 Discrete Resources and the World Interface

This section is to recap and to provide more background regarding resources, constructions and the roles of the world and attacker interfaces.

Discrete systems and interfaces. We model all resources in these work as 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 resource as a response. In general, the same resource may be accessible to multiple parties, for example a communication channel that allows a sender to write a message and a receiver to read it. In that case, we assign inputs to certain *interfaces* that correspond to the parties: the sender's interface allows to write a message to the channel, and the receiver's interface allows to read what is in the channel. More generally, an interface does not necessarily correspond to one specific particular role in a cryptographic protocol: in a security statement, a single party can be given access to multiple interfaces of a resource. In the construction statements, (protocol) converters are attached to the parties' interfaces in the "real world," whereas the environment is given direct access in the "ideal world." This formalizes that the protocol, a tuple of (protocol) converters, is supposed to implement the same interfaces for the honest parties as the ideal/desired resource.

The attacker interface and the world interface. There are two interfaces that serve different, specific purposes. The first such interface, usually referred to as the dishonest (or possibly dishonest) interface, models the capabilities that a resource may allow to a potential attacker. In the construction statements, a (simulator) converter is attached to this interface in the "ideal world," whereas the environment is given direct access in the "real world." This formalizes that the constructed resource (together with the simulator) allows the same capabilities to the attacker as the assumed resource with the protocol or, in other words, that the protocol restricts the attacker as specified by the constructed/desired resource. The second special interface is referred to as the "world" or "environment" interface, and models that the resource may be affected by its environment in ways we do not want to make explicit in the resource description. As such, the environment interface also allows to abstract from certain technical details by delegating them to the environment. In the construction statements, this means that the world interface is directly accessible in both the "real world" and the "ideal world:" The construction is valid independently of what happens at the world interface.

To give one more example, the difference between specifying capabilities at the attacker's and the world interface can also be illustrated when modeling communication: If a certain capability, such as the decision to deliver or drop a message from a sender to a receiver, is provided at the attacker's 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 dropping the message, even if the distinguisher's behavior corresponds to delivering the message in the "real world." This means that the scheme does not have to guarantee that the receiver output the message; in particular, both the receiver's protocol and the simulator may drop messages. In contrast, if the capability is provided at the world interface, then the message will indeed be delivered in the "ideal world," since that is exactly what is specified in the resource. Consequently, the protocol in the "real world" has to provide the message to the receiver as well, because otherwise the two settings were distinguishable.

Uses of the world interface. In the context of this work, the world interface is denoted by W and allows to decide when a server is allowed to overwrite the client's data. Along the same lines as the above example, this means that we can enforce that a protocol must guarantee that detecting certain errors within the server memory does not imply that the entire memory is considered as "deleted". This way, we are guaranteed that a protocol stays operational for the faultless part of the server memory. Furthermore, we can explicitly model when the resource is "under attack".

The world interface, which is introduced and described in more detail by Gaži *et al.* [29], is a more general concept that applies to more situations than the one described above: One example is to model global resources that can be used in multiple different protocols as in Generalized UC [18]. Generally, the world interface is instrumental whenever certain conditions have to be kept synchronized between the "real world" and the "ideal world," such as in security statements involving adaptive corruption or protocol resources that may fail and are, e.g., made redundant to be more resilient [29].