Verifiable Data Streaming

Dominique Schröder
Saarland University, Germany
University of Maryland, USA

Heike Schröder
Technical University of Darmstadt, Germany

ABSTRACT

In a *verifiable data streaming* protocol, the client streams a long string to the server who stores it in its database. The stream is verifiable in the sense that the server can neither change the order of the elements nor manipulate them. The client may also retrieve data from the database and update them. The content of the database is publicly verifiable such that any party in possession of some value s and a proof π can check that s is indeed in the database.

We introduce the notion of verifiable data streaming and present an efficient instantiation that supports an exponential number of values based on general assumptions. Our main technique is an authentication tree in which the leaves are not fixed in advanced such that the user, knowing some trapdoor, can authenticate a new element on demand *without* pre- or re-computing all other leaves. We call this data structure *chameleon authentication tree* (CAT). We instantiate our scheme with primitives that are secure under the discrete logarithm assumption. The algebraic properties of this assumption allow us to obtain a very efficient verification algorithm. As a second application of CATs, we present a new transformation from any one-time to many-time signature scheme that is more efficient than previously known solutions.

Categories and Subject Descriptors

F.0 [**Theory of Computation**]: General

General Terms

Security, Theory

Keywords

Outsourcing, streaming, verifiable delegation

1. INTRODUCTION

In a verifiable data streaming protocol (VDS), the client $\mathcal{C}$ streams a long string $S = s[1], \dots, s[m]$ to the server $\mathcal{S}$ who stores the string in its database DB. The length of S exceeds the client's memory such that $\mathcal{C}$ cannot read or store the entire string at once. Instead, $\mathcal{C}$ reads some substring $s[i] \in \{0,1\}^k$ and sends it to the server. The stream is verifiable in the sense that the server cannot change the order of the elements or manipulate them. The entries in the database are publicly verifiable such that any party in possession of $s[i]$ and a proof $\pi_{s[i]}$ can check that $\mathcal{C}$ stored $s[i]$ in DB hold by $\mathcal{S}$.

The client has also the ability to retrieve and update any element in the database. Whenever the client wishes to get some value $s[j]$ from DB, then the server appends a proof $\pi_{s[j]}$ that shows the authenticity of $s[j]$ with respect to some verification key PK. To update some element i in DB, the client retrieves $(s[i], \pi_{s[i]})$ from $\mathcal{S}$, checks its validity, and sends the updated value $s'[i]$ back to the server.

1.1 Trivial Approaches

It seems that a VDS can easily obtained by letting the client sign all values it streams to the server. This solution indeed works for append-only databases that do not consider the order of the elements (clearly, a stateful solution encodes the position into each element). What makes the problem interesting is that the client can update the elements in the database. In this setting, the trivial solution does not work anymore, because all previous entries in DB would still verify under the public key and it is unclear how to revoke previous signatures. One could let the client store all previous elements locally in order to keep track of all changes. This approach, however, would not only lead the problem *ad absurdum*, but it simply is impossible due to the limited storage capacities of the client.

1.2 Applications

Several companies offer already storage in the Internet such as, e.g., Google drive, Dropbox, Apple's iCloud, and many more. The basic idea is that users can outsource most of their data to a seemingly unbounded storage. In some cases, such as Google's Chromebook, the complete data are stored in the cloud while the client keeps only a small portion of the data. Google provides for the users of the Chromebook some free storage, but they have to pay a yearly fee for any additional space.

From a high-level point of view, this can be seen as data streaming, where a weak client streams a huge amount of data to a very powerful server. A crucial point here is the authenticity of the data. How can the client make sure that the server is not charging the client for space it is not using (e.g., by adding random data to the user's database)? Furthermore, the client needs to verify that the server keeps the current version of the data without modifying it, or switching back to an old version.

Taking the order of data into account is a very natural requirement in computer science. As an example consider a server that stores the DNA sequences for a health insurance. A difference in the sequence of the DNA usually means some sort of mutation which effects, e.g., in a disease. Thus, if a malicious server manages to change the order in a patient's DNA sequence, then the client might have to pay a higher fee due to some medical risks (such as genetic disease) that might be implied by this mutation. The website of the Museum of Paleontology of UC Berkley describes the affect of mutations [oCMoP12].

1.3 Our Contribution

We introduce the notion of verifiable data streaming and present an instantiation that supports an exponential number of values based on general assumptions. Our solution has efficient updates and the data in the database are publicly verifiable. Moreover, our construction is secure in the standard model. We summarize our contributions as follows:

- Our main technical contribution is an authentication tree that authenticates an exponential number of values, but where the leaves are not defined in advance. The owner of a trapdoor can add elements to the tree without pre- or re-computing all other elements. We call such a tree a chameleon authentication tree (CAT).
- We show the generality of our technique by applying it to two different problems: Firstly, we build a verifiable data streaming protocol based on CATs. This scheme supports an exponential number of elements, efficient updates, and the items in the database are publicly verifiable. The second application of CATs is a new transformation from any one-time to many-time signature scheme in the standard model that is more efficient than all previous approaches.
- We instantiate our construction with primitives that are secure under the discrete logarithm assumption in the standard model. This assumption is not only very mild, but the algebraic structure allows us to obtain a more efficient verification algorithm. The basic idea is to apply batch verification techniques to our verification algorithm.

1.4 Related Work

Verifiable data streaming is related to verifiable databases (VDB) by Benabbas, Gennaro, and Vahlis [BGV11]. The main difference to their work is that the data during the setup phase in a streaming protocol are unknown. Moreover, our notion has an algorithm that allows adding elements to the database that consists of a single message from the user to the server (which does not change the verification key of the database). One might wonder if VDBs can be used to simulate verifiable data streaming protocols by generating a database of exponential size and adding the entries via the update algorithm. This idea, however, does not work because the update procedure usually requires interaction and the server updates the verification key afterwards. Another difference is that the data in a VDB are usually unordered. That is, the element d_i in the database DB is associated to some key x_i, i.e., $DB(x_i) = d_i$. But there is no explicit ordering of the elements.

The problem of VDBs has previously been investigated in the context of accumulators [Ngu05, CKS09, CL02] and authenticated data structures [NN00, MND$^+$01, PT07, TT10]. These approaches, however, often rely on non-constant assumptions (such as the q-Strong Diffie-Hellman assumption) as observed in [BGV11]. Recent works, such as [BGV11] or [CF11], focus on storing specific values (such as polynomials) instead of arbitrary ones and they usually only support a polynomial number of values (instead of exponentially many). Moreover, the scheme of [BGV11] is not publicly verifiable.

Proofs-of-retrievability are also similar in the sense that the server proves to the client that it is actually storing all of the client's data [SW08, FB06, SM06]. The interesting research area of memory delegation [CKLR11] is also different, because it considers verifiable computation on (streamed) data. A more efficient solution has been suggested by Cormode, Mitzenmacher, and Thaler in [CMT12].

2. VERIFIABLE DATA STREAMING

In a verifiable data streaming protocol (VDS), a client $\mathcal{C}$ reads some long string $S = s[1], \ldots, s[m] \in \{0,1\}^{mk}$ that $\mathcal{C}$ wishes to outsource to a server $\mathcal{S}$ in a streaming manner. Since the client cannot store and read the entire string at once, $\mathcal{C}$ reads a substring $s[i] \in \{0,1\}^k$ of S and sends $s[i]$ to the server who stores the value in its database DB. We stress that we are interested in a streaming protocol, i.e., the communication between the client and the server at this stage is unidirectional and the string is ordered. The data must be publicly verifiable in the sense that the server holds some public key PK and everybody in possession of some data $s[i]$ and a proof $\pi_{s[i]}$ can verify that $s[i]$ is stored in DB. Whenever the client wishes to retrieve some data $s[i]$ from the database DB, $\mathcal{C}$ sends i to the server who returns $s[i]$ together with a proof $\pi_{s[i]}$. This proof shows that s is the i^{th} element in DB and its authenticity with respect to PK. In addition, the client has the ability to update any value $s[i]$ to a new string $s'[i]$ which leads to a new verification key PK'. More formally:

Definition 1. A verifiable data streaming protocol $\mathcal{VDS} = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Update})$ is a protocol between two PPT algorithms: a client $\mathcal{C}$ and a server $\mathcal{S}$. The server $\mathcal{S}$ can store an exponential number n of elements in its database DB and the client keeps some small state $\mathcal{O}(\log n)$. The scheme consists of the following PPT algorithms:

$\mathsf{Setup}(1^\lambda)$: The setup algorithm takes as input the security parameter 1^λ. It returns a verification key PK and a secret key SK. The verification key PK is given to the server $\mathcal{S}$ and the secret key to the client $\mathcal{C}$. W.l.o.g., SK always contains PK.

$\mathsf{Append}(\boldsymbol{SK}, s)$: This algorithm appends the value s to the database DB hold by the server. The client sends a single message to the server who stores the element in DB. Adding elements to the database may change the private key to SK', but it does not change the verification key PK.

$\mathsf{Query}(\boldsymbol{PK}, \boldsymbol{DB}, i)$: The interactive query protocol is executed between $\mathcal{S}(PK, DB)$ and $\mathcal{C}(i)$. At the end of the protocol, the client either outputs the i^{th} entry $s[i]$ of DB together with a proof $\pi_{s[i]}$, or $\perp$.

$\mathsf{Verify}(\boldsymbol{PK}, i, s, \pi_{s[i]})$: The verification algorithm outputs $s[i]$ if $s[i]$ is the i^{th} element in the database DB, otherwise it returns $\perp$.

$\mathsf{Update}(\boldsymbol{PK}, \boldsymbol{DB}, \boldsymbol{SK}, i, s')$: The interactive update protocol $\langle \mathcal{S}(PK, DB), \mathcal{C}(SK, i, s') \rangle$ takes place between the server $\mathcal{S}(PK, DB)$ and the client $\mathcal{C}(SK, i, s')$ who wishes to update the i^{th} entry of the database DB to s'. At the end of the protocol the server sets $s[i] \leftarrow s'$ and both parties update PK to PK'.

A verifiable data streaming protocol must fulfill the usual completeness requirements.

2.1 Efficiency and Security Evaluation of VDS

Verifiable data streaming protocols should fulfill both "system" and "crypto" criteria. System criteria usually require that a scheme must be as efficient as possible. In our setting, efficiency should be evaluated w.r.t. computational complexity, storage complexity, and communication complexity. The server in a VDS must be able to store an exponential number of elements and we require that there is no a-priori bound on the number of queries to the server. The verifiers in the system should be stateless with public verifiability. Everybody in possession of a data s and a proof π_s should be able to verify that s is stored at position i and that s is valid w.r.t. the verification key PK.

The most important crypto criteria are the following: A malicious server $\mathcal{A}$ should not be able to add elements to the database outsourced by the client without its help. This means that $\mathcal{A}$ might ask the client to add q elements to its database DB (where q is adaptively determined by $\mathcal{A}$), but he is unable to add any further element that verifies under PK. A verifiable streaming protocol is order-preserving, i.e., the malicious server $\mathcal{A}$ cannot change the order of any element in the database. Furthermore, $\mathcal{A}$ should not be able to change any element in the database. Again, this property must hold even if $\mathcal{A}$ has the ability to ask the client to update q elements of its choice. Finally, the server should only be able to issue proofs that allow to recover the stored file. These criteria follow the ones that have been suggested in the context of proofs-of-retrievability [SW08].

2.2 Security of VDS

The security notion of VDS is similar to the one of verifiable databases [BGV11], but differs in many aspects: First, our model considers the case of public verifiability, while the one of [BGV11] does not. Second, we are dealing with a stream of data that has an explicit ordering. In contrast, the model of [BGV11] fixes the size of DB during the setup and guarantees authenticity only for these data. In particular, the adversary breaks the security in our model if he manages to output a data $s[i]$ with a valid proof $\pi_{s[i]}$, but where $s[i]$ is *not* the i^{th} value in DB.

We model this intuition in a game between a challenger and an adversary $\mathcal{A}$ that adaptively adds and updates elements to resp. in the database. At the end of the game, $\mathcal{A}$ tries to compute a false statement saying that a *different* data $s[i]$ is the i^{th} value in the database. This covers the different attack scenarios we have discussed so far. A successful attacker could (1) change the order of an element; (2) add an element to the database without the help of the user; (3) break the update mechanism. More formally:

Setup: The challenger runs $\mathsf{Setup}(1^\lambda)$ to generate a private key SK and a public key PK. It sets up an initially empty database DB and gives the public key PK to the adversary $\mathcal{A}$.

Queries: The challenger provides two interfaces for $\mathcal{A}$ that $\mathcal{A}$ may query adaptively and in an arbitrary order. If the adversary queries the *append interface* on some data s, then the challenger will run $\mathsf{Append}(SK, s)$ to append s to its database DB. Subsequently, it returns the corresponding proof π_s to $\mathcal{A}$. The second interface is an *update interface* that takes as input an index j and an element $s'[j]$. Whenever $\mathcal{A}$ invokes this interface, the challenger will run the protocol $\mathsf{Update}(PK, DB, SK, i, s')$ with $\mathcal{A}$. Notice that each call to this interface will update the verification key as well. By $DB = s[1], \ldots, s[q]$ we denote the state of the database after $\mathcal{A}$'s last query and PK^* is the corresponding verification key stored by the challenger.

Output: Eventually, the adversary outputs $(i^*, s^*, \pi^*_{s^*})$ and let $\hat{s} \leftarrow \mathsf{Verify}(PK^*, i^*, s^*, \pi^*_{s^*})$. The adversary is said to win the game if $\hat{s} \neq \perp$ and $\hat{s} \neq s[i^*]$.

We define $\mathbf{Adv}^{\mathsf{os}}_{\mathcal{A}}$ to be the probability that the adversary $\mathcal{A}$ wins in the above game.

Definition 2. A verifiable data streaming protocol is secure if for any efficient adversary $\mathcal{A}$ the probability $\mathbf{Adv}^{\mathsf{os}}_{\mathcal{A}}$ is negligible (as a function of λ).

3. PRELIMINARIES

Before describing our construction formally, we introduce the following basic notations for binary trees (c.f. consider the tree depicted in Figure 1). The algorithms using this tree will be described in the subsequent sections. Let CAT be a binary tree consisting of a root node ρ, a set of inner nodes v, and some leaf nodes ℓ. The depth of the tree is defined by $D = \mathsf{poly}(\lambda)$ and the level of a node in the tree is denoted by $h = 0, \ldots, D-1$, where leaf nodes have level $h = 0$ and the root node in turn has level $h = D-1$. At each level h the nodes are defined by $v_{h,i}$, where $i = 0, \ldots, 2^{D-h}$ is the position of the node in the tree counted from left to right. Furthermore, inner nodes of the tree are computed according the following rule: $v_{h,i} \leftarrow \mathsf{Ch}(v_{h-1,\lfloor i/2^{h-1} \rfloor} || v_{h-1,\lfloor i/2^{h-1}-2 \rfloor}; r_{h,\lfloor i/2^h+1 \rfloor})$, if $\lfloor i/2^h \rfloor \equiv 1 \mod 2$ and $v_{h,i} \leftarrow H(v_{h-1,\lfloor i/2^{h-1} \rfloor -2} || v_{h-1,\lfloor i/2^{h-1}-1 \rfloor})$, if $\lfloor i/2^h \rfloor \equiv 0 \mod 2$. Notice that H is a hash function, Ch a chameleon hash function, and r some randomness. By ℓ_i we denote the i^{th} leaf counted from left to right. The authentication path aPath of a leaf ℓ consists of all siblings of the nodes on the path from ℓ to ρ. If a parent node is computed by a chameleon hash, then it also stores the corresponding randomness in a list R. E.g., the authentication path $\mathsf{aPath}_{\ell_{15}}$ of the leaf ℓ_{15} is $\mathsf{aPath}_{\ell_{15}} = (\ell_{14}, v_{1,6}, r_{2,3}, v_{2,2}, r_{3,1}, v_{3,0})$ and $R_{\ell_{15}} = (r_{1,7}, r_{2,3}, r_{3,1}, r_\rho)$.

3.1 Chameleon Hash Functions and their Security

A chameleon hash function is a randomized hash function that is collision-resistant but provides a trapdoor [KR00].

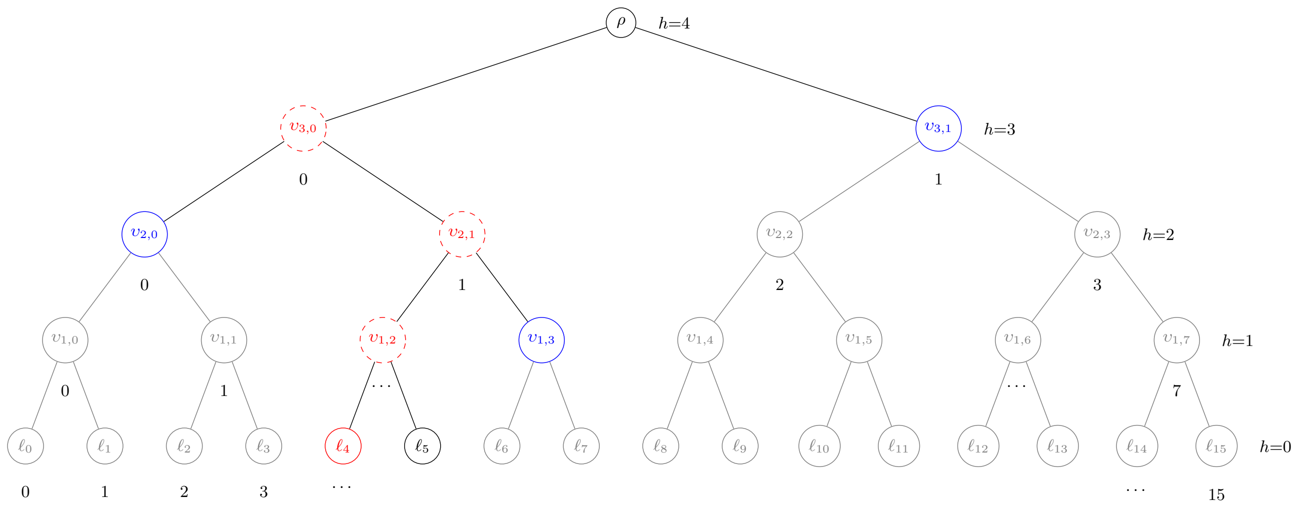


Figure 1: Given a binary tree CAT which consists of a root node ρ, a set of inner nodes v, and some leaf nodes ℓ. The blue nodes define the authentication path of leaf ℓ_4. The gray nodes right-handed of ℓ_5 do not exist at this stage.

Given the trapdoor *csk*, a message x with some randomness r, and any additional message x', it is possible to efficiently compute a value r' such that the chameleon hash algorithm Ch maps to the same y, i.e., $\mathsf{Ch}(x;r) = \mathsf{Ch}(x';r') = y$.

Definition 3. A chameleon hash function is a tuple of PPT algorithms $\mathcal{CH} = (\mathsf{Gen}, \mathsf{Ch}, \mathsf{Col})$:

Gen(1^λ)**:** The key generation algorithm returns a key pair (csk, cpk) and we set $\mathsf{Ch}(\cdot) := \mathsf{Ch}(cpk, \cdot)$.

Ch$(x; r)$**:** The input of the hash algorithm is a message $x \in \{0,1\}^{\mathsf{in}}$ and some randomness $r \in \{0,1\}^\lambda$ (which is efficiently sampable from some range $\mathcal{R}_{cpk}$). It outputs a hash value $h = \mathsf{Ch}(x;r) \in \{0,1\}^{\mathsf{out}}$.

Col$(\boldsymbol{csk}, x, r, x')$**:** The collision-finding algorithm returns a value r' such that $\mathsf{Ch}(x;r) = y = \mathsf{Ch}(x';r')$.

Uniform Distribution: The output of Ch is uniformly distributed, i.e., it also holds that for any cpk, x, r, x' the distribution of $\mathsf{Col}(csk, x, r, x')$ (over the choice of r) is the same as the distribution of r itself, also implying that a hash value $\mathsf{Ch}(x;r)$ (over the choice of r) is distributed independently of x.

A chameleon hash function must be collision-resistant. This means that any malicious party should not be able to find two pairs (x_0, r_0) and (x_1, r_1) that map to the same image. More precisely is the following definition.

Definition 4. A chameleon hash function $\mathcal{CH} = (\mathsf{Gen}, \mathsf{Ch}, \mathsf{Col})$ is collision-resistant if the advantage function $\mathbf{Adv}^{ch\text{-}col}_{\mathcal{CH},\mathcal{A}}$ is a negligible function in λ for all PPT adversaries $\mathcal{A}$, where

$$\mathbf{Adv}^{ch\text{-}col}_{\mathcal{CH},\mathcal{A}} := \Pr\left[\begin{array}{l}\mathsf{Ch}(x;r) = \mathsf{Ch}(x';r') \\ \text{and } (x,r) \neq (x',r')\end{array} : \begin{array}{l}(csk, cpk) \leftarrow \mathsf{Gen}(1^\lambda); \\ (x, x', r, r') \leftarrow \mathcal{A}(\mathsf{Ch})\end{array}\right].$$

Observe that collision-resistance only holds as long as the adversary has not learned a collision. Indeed, some chameleon hash functions allow to recover the private key if a collision is known, such as, e.g., [KR00]. A comprehensive discussion about this problem is given in [Ad04]. Collision-resistance of hash functions is defined analogously and omitted here.

4. CATS

The central building block that we use in our VDS protocol is a technique that we call *chameleon authentication tree* (CAT). A CAT is an authentication tree that has the ability to authenticate an exponential number of 2^D leaves that are not fixed in advanced, where $D = \mathsf{poly}(\lambda)$. Instead, the owner of a trapdoor can authenticate a new element on demand *without* pre- or re-computing all other leaves.

4.1 Formal Definition of CATs

We formalize CATs as a triple of efficient algorithms: A CAT generation algorithm catGen, a path generation algorithm addLeaf that adds a leaf to the tree and returns the corresponding authentication path, and a path verification algorithm catVrfy that checks if a certain leaf is part of the tree.

Definition 5. A chameleon authentication tree is a tuple of PPT algorithms $\mathsf{CAT} = (\mathsf{catGen}, \mathsf{addLeaf}, \mathsf{catVrfy})$:

$\mathsf{catGen}(1^\lambda, D)$**:** The CAT generation algorithm takes as input a security parameter λ and an integer D that defines the depth of the tree. It returns a private key sp and verification key vp.

$\mathsf{addLeaf}(\mathsf{sp}, \ell)$**:** The path generation algorithm takes as input a private key sp and a leaf $\ell \in \mathcal{L}$ from some leaf space $\mathcal{L}$. It outputs a key sp′, the index i of ℓ in the tree, and the authentication path aPath.

$\mathsf{catVrfy}(\mathsf{vp}, i, \ell, \mathsf{aPath})$**:** The verification algorithm takes as input a public key vp, an index i, a leaf $\ell \in \mathcal{L}$, and a path aPath. It outputs 1 iff ℓ is the i^{th} leaf in the tree, otherwise 0.

A CAT must fulfill the usual completeness requirements.

4.2 Security of CATs

We identify two security properties that a CAT should support. Loosely speaking, an adversary $\mathcal{A}$ should not be able to change the structure of the CAT. In particular, changing the sequence of the leaves, or substitute any leaf should be a successful attack. We call this property *structure-preserving*. Furthermore, an adversary should not be able to add further leaves to a CAT. We refer to this property as *one-wayness*.

STRUCTURE-PRESERVING. We formalize the first property by an interactive game between the challenger and an adversary $\mathcal{A}$. The challenger generates a key pair $(\mathsf{sp}, \mathsf{vp})$ and hands the verification key vp over to the adversary $\mathcal{A}$. The attacker may then send q leaves $\ell_1, \ldots, \ell_{q(\lambda)}$ (adaptively) to the challenger who returns the corresponding authentication paths $(\mathsf{aPath}_1, \ldots, \mathsf{aPath}_{q(\lambda)})$. Afterwards, the adversary $\mathcal{A}$ tries to break the structure of the CAT by outputting a leaf that has not been added to the tree at a particular position. More formally:

Setup: The challenger runs the algorithm $\mathsf{catGen}(1^\lambda, D)$ to compute a private key sp and a verification key vp. It gives vp to the adversary $\mathcal{A}$.

Streaming: Proceeding adaptively, the attacker $\mathcal{A}$ sends a leaf $\ell \in \mathcal{L}$ to the challenger. The challenger computes $(\mathsf{sp}', i, \mathsf{aPath}) \leftarrow \mathsf{addLeaf}(\mathsf{sp}, \ell)$ and returns (i, aPath) to $\mathcal{A}$. Denote by $Q := \{(\ell_1, 1, \mathsf{aPath}_1), \ldots, (\ell_{q(\lambda)}, q(\lambda), \mathsf{aPath}_{q(\lambda)})\}$ the ordered sequence of query-answer pairs.

Output: Eventually, $\mathcal{A}$ outputs $(\ell^*, i^*, \mathsf{aPath}^*)$. The attacker $\mathcal{A}$ is said to win the game if: $1 \leq i^* \leq q(\lambda)$ and $(\ell^*, i^*, \mathsf{aPath}^*) \notin Q$ and $\mathsf{catVrfy}(\mathsf{vp}, i^*, \ell^*, \mathsf{aPath}^*) = 1$.

We define $\mathbf{Adv}^{\mathsf{sp}}_{\mathcal{A}}$ to be the probability that the adversary $\mathcal{A}$ wins in the above game.

Definition 6. A chameleon authentication tree CAT, defined by the efficient algorithms $(\mathsf{catGen}, \mathsf{addLeaf}, \mathsf{catVrfy})$ with n leaves, is $\mathsf{structure\text{-}preserving}$ if for any $q \in \mathbb{N}$, and for any PPT algorithm $\mathcal{A}$, the probability $\mathbf{Adv}^{\mathsf{sp}}_{\mathcal{A}}$ is negligible (as a function of λ).

ONE-WAYNESS. We model the second property in a game between a challenger and an adversary as follows:

Setup: The challenger runs the algorithm $\mathsf{catGen}(1^\lambda, D)$ to compute a private key sp and a verification key vp. It gives vp to the adversary $\mathcal{A}$.

Streaming: Proceeding adaptively, the attacker $\mathcal{A}$ streams a leaf $\ell \in \mathcal{L}$ to the challenger. The challenger computes $(\mathsf{sp}', i, \mathsf{aPath}) \leftarrow \mathsf{addLeaf}(\mathsf{sp}, \ell)$ and returns (i, aPath) to $\mathcal{A}$.

Output: Eventually, $\mathcal{A}$ outputs $(\ell^*, i^*, \mathsf{aPath}^*)$. The attacker $\mathcal{A}$ is said to win the game if: $q(\lambda) < i^* \leq n$ and $\mathsf{catVrfy}(\mathsf{vp}, i^*, \ell^*, \mathsf{aPath}^*) = 1$.

We define $\mathbf{Adv}^{\mathsf{ow}}_{\mathcal{A}}$ to be the probability that the attacker $\mathcal{A}$ wins in the above game.

Definition 7. A chameleon authentication tree CAT, defined by the PPT algorithms $(\mathsf{catGen}, \mathsf{addLeaf}, \mathsf{catVrfy})$ with n leaves, is $\mathsf{one\text{-}way}$ if for any $q \in \mathbb{N}$, and for any PPT algorithm $\mathcal{A}$, the probability $\mathbf{Adv}^{\mathsf{ow}}_{\mathcal{A}}$ is negligible (as a function of λ).

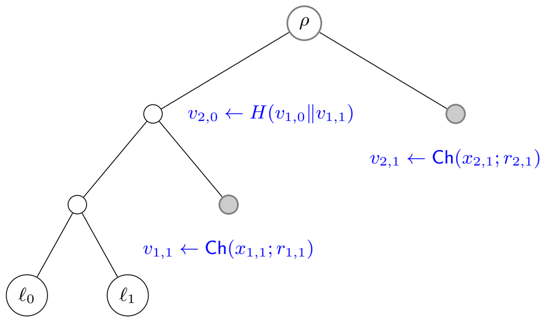
5. OUR SCHEME

The basic idea of our construction is a careful combination of hash functions H and chameleon hash functions Ch. Recall that in a chameleon hash function the owner of the trapdoor can easily find collisions, i.e., for a given string x (and randomness r) there exists an efficient algorithm that computes a value r' such that $\mathsf{Ch}(x; r) = y = \mathsf{Ch}(x'; r')$. We first discuss why obvious approaches do not seem to work. After explaining the main ideas of our construction, we define them formally and give a proof of security.

5.1 Naïve Approaches do not Work

The first idea would be to build a Merkle tree, where the server stores the entire tree and the client keeps the last authentication path in its state. This idea, however, does not work, because all leaves are necessary to compute the root. Using dummy nodes does not solve the problem, because the root would change whenever a new leaf is authenticated. Thus, the second approach might be to store the outputs of chameleon hash functions as the leaves with the idea that whenever the client wishes to authenticate a new value, it simply applies the trapdoor such that the new leaf authenticates under the same root. This idea, however, does still not work. One reason is that the client would have to store all leaves (together with the corresponding randomness) in order to compute a collision. One might be temped to let the server store these values, but this does not work for several reasons: First of all, the data are streamed. This means that there is no communication from the server to the client at this stage. But even if we would allow bi-directional interaction, it would immediately lead to an attack: Suppose that the client wishes to append the leaf $\hat{\ell}$. To do so, $\mathcal{C}$ asks the server to send the dummy leaf ℓ with the corresponding randomness r. Then, the client applies the trapdoor to compute the matching randomness $\hat{r}$ and sends the updated values $\hat{\ell}, \hat{r}$ to the server. The problem is that the malicious server would learn a colluding pair $(\ell, r), (\hat{\ell}, \hat{r})$ such that $\mathsf{Ch}(\ell; r) = \mathsf{Ch}(\hat{\ell}; \hat{r})$. In many schemes, this knowledge would allow the server to compute *another* pair (ℓ^*, r^*) such that $\mathsf{Ch}(\ell; r) = \mathsf{Ch}(\hat{\ell}; \hat{r}) = \mathsf{Ch}(\ell^*; r^*)$ (some schemes even allow to recover the trapdoor *csk* if one knows a collision, such as, e.g., [KR00]).

5.2 Intuition of our Construction



ρ
$v_{2,0} \leftarrow H(v_{1,0} \| v_{1,1})$
$v_{2,1} \leftarrow \mathsf{Ch}(x_{2,1}; r_{2,1})$
$v_{1,1} \leftarrow \mathsf{Ch}(x_{1,1}; r_{1,1})$
ℓ_0
ℓ_1

Figure 2: A CAT of depth 3 that authenticates the leaves ℓ_0 and ℓ_1. The root and the right nodes are computed by a chameleon hash and the left nodes by a collision-resistant hash function. The leaves $\ell_2, \ldots, \ell_7$ are unknown.

As a warm up, we first illustrate the high-level idea of our instantiation with the tree shown in Figure 2. We stress that our actual construction is slightly different, because the catGen algorithm does not know the leaves ℓ_0 and ℓ_1.

We set up the tree such that the root and every right-handed node of the tree are computed by a chameleon hash function and all left-handed nodes with a collision-resistant hash function. The first step is to set up the tree by computing the hash value of the leaves ℓ_0 and ℓ_1 as $v_{1,0} \leftarrow H(\ell_0 \| \ell_1)$. Then, the algorithm picks two random values $x_{1,1}, r_{1,1}$ to compute the dummy right-handed node $v_{1,1} \leftarrow \mathsf{Ch}(x_{1,1}; r_{1,1})$. The next step is to compute the parent node $v_{2,0} \leftarrow H(v_{1,0} \| v_{1,1})$ and to pick two additional random values $x_{2,1}, r_{2,1}$. The algorithm then sets $v_{2,1} \leftarrow \mathsf{Ch}(x_{2;1}; r_{2,1})$ and $\rho \leftarrow \mathsf{Ch}(v_{2,0} \| v_{2,1}; r_\rho)$ using some randomness r_ρ. The authentication path of the leaves ℓ_0 and ℓ_1 only consists of the nodes $v_{1,1}, v_{2,1}$ and the randomness r_ρ. It does *not* contain the pre-images $x_{1,1}, r_{1,1}$ (resp. $x_{2,1}, r_{2,1}$). We stress that this is crucial for the security proof.

To add the elements ℓ_2 and ℓ_3 to the tree, the algorithm sets $x'_{1,1} \leftarrow (\ell_2 \| \ell_3)$ and computes the collision for the node $v_{1,1}$ using its trapdoor csk and randomness $r_{1,1}$, i.e., $r'_{1,1} \leftarrow \mathsf{Col}(csk, x_{1,1}, r_{1,1}, x'_{1,1})$. This means that the chameleon hash function maps to the same value $v_{1,1} = \mathsf{Ch}(x_{1,1}; r_{1,1}) = \mathsf{Ch}(x'_{1,1}; r'_{1,1})$ and thus, the tree authenticates the leaves ℓ_2, ℓ_3 (using randomness $r'_{1,1}$). The authentication path of the leaves ℓ_2, ℓ_3 consists of $\mathsf{aPath} = (v_{0,1}, v_{2,1})$ and $R = (r'_{1,1}, r_\rho)$. Thus, the attacker only learns $x'_{1,1}, r'_{1,1}$ and not the dummy values $x_{1,1}, r_{1,1}$ that has been used to compute $v_{1,1}$.

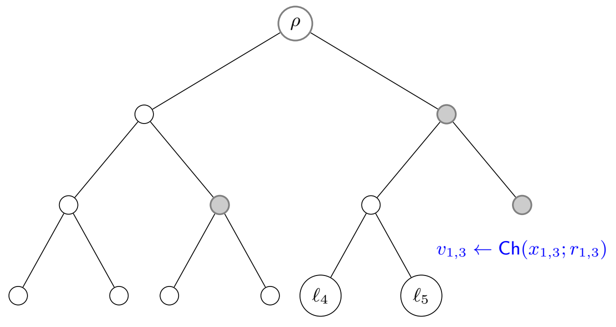


Figure 3: Appending the leaves ℓ_4 and ℓ_5 to the CAT, requires the computation of a collision in node $v_{2,1}$.

Now, assume that we would like to add two additional elements ℓ_4 and ℓ_5 to the CAT (c.f. Figure 3). Observe that we are in the situation where all leaves in the left part of tree are used and the right part of the tree consists only of the element $v_{2,1}$. The complete right subtree with root node $v_{2,1}$ does not exist at this point as it was unnecessary to authenticate any of the previous leaves. In order to authenticate the leaves ℓ_4 and ℓ_5, our algorithm generates the skeleton of the right subtree that is needed for the corresponding authentication path. That is, the algorithm computes $v_{1,2} \leftarrow H(\ell_4 \| \ell_5)$, picks two random values $x_{1,3}, r_{1,3}$, and sets $v_{1,3} \leftarrow \mathsf{Ch}(x_{1,3}; r_{1,3})$. The last step is to apply the trapdoor to the chameleon hash function used in node $v_{2,1} = \mathsf{Ch}(x_{2,1}; r_{2,1})$, i.e., it sets $x'_{2,1} \leftarrow (v_{1,2} \| v_{1,3})$ computes $r'_{2,1} \leftarrow \mathsf{Col}(csk, x_{2,1}, r_{2,1}, x'_{2,1})$. The authentication path of the leaves ℓ_4, ℓ_5 consists of $\mathsf{aPath} = (v_{1,3}, v_{2,0})$ and $R = (r_{2,1}, r_\rho)$.

We would like to draw the readers attention to the way we apply the trapdoor to the nodes. The idea is to apply the trapdoor to the first node on the path from the leaf ℓ_i to the root ρ that is computed by a chameleon hash. This way we guarantee that each trapdoor is applied to each node only once and therefore, the one who stores the tree never sees a collision.

5.3 Special Property of the Construction

One of the interesting properties of our construction is the amount of information that is needed to add an element to the tree. In fact, only the current authentication path, the pre-images of the chameleon hashes with the corresponding randomness, and the trapdoor are needed. More precisely, consider the tree shown in Figure 2. It is basically the authentication path for the leaves ℓ_0, ℓ_1 and the missing nodes that are required to compute the authentication path for the leaves ℓ_4 and ℓ_5 are computed on the fly. This means that if the tree has depth 2^D for some $D = \mathsf{poly}(\lambda)$, the clients stores only $\log(2^D) = \mathsf{poly}(\lambda)$ elements. As it turns out, this property will be very useful for our verifiable data streaming protocol, where the client essentially stores these elements and the sever the entire tree.

6. OUR CONSTRUCTION

Although the high level idea of CATs is quickly graspable, the formal description is rather complicated. To simplify the exposition, we denote by $[\mathbf{a}]$ a vector of elements, i.e., $[\mathbf{a}] = (a_0, \ldots, a_{D-1})$ and $[(\mathbf{x}, \mathbf{r})] = ((x_0, r_0), \ldots, (x_{D-1}, r_{D-1}))$, resp.

Construction 1. Let H be a hash function and let $\mathcal{CH} = (\mathsf{Gen}, \mathsf{Ch}, \mathsf{Col})$ be a chameleon hash function. We define the chameleon authentication tree $\mathsf{CAT} = (\mathsf{catGen}, \mathsf{addLeaf}, \mathsf{catVrfy})$ as follows:

$\mathsf{catGen}(1^\lambda, D)$**:** The key generation algorithm computes keys of the chameleon hash function $(cpk, csk) \leftarrow \mathsf{Gen}(1^\lambda)$ and $(cpk_1, csk_1) \leftarrow \mathsf{Gen}(1^\lambda)$. It picks two random values x_ρ, r_ρ, sets $\rho \leftarrow \mathsf{Ch}(x_\rho; r_\rho)$, sets the counter $c \leftarrow 0$, and the state to $\mathtt{st} \leftarrow (c, D, x_\rho, r_\rho)$. It returns the private key $\mathtt{sp}$ as $(csk, \mathtt{st})$ and the public key $\mathtt{vp}$ as (cpk, ρ).

$\mathsf{addLeaf}(\mathtt{sp}, \ell)$**:** The path generation algorithm parses the private key $\mathtt{sp}$ as $(csk, \mathtt{st})$ and recovers the counter c from $\mathtt{st}$. Then, it picks a random value r, sets $(\ell_c, \ell_{c+1}) \leftarrow \mathsf{Ch}_1(\ell; r)$ and distinguishes between two cases:

$c = 0$**:** $\mathsf{addLeaf}$ picks random values $x_{h,1} \leftarrow \{0,1\}^{2\mathsf{len}}$ $r_{h,1} \leftarrow \{0,1\}^\lambda$ (for $h = 1, \ldots, D-2$), and sets $v_{h,1} \leftarrow \mathsf{Ch}(x_{h,1}; r_{h,1})$. Subsequently, it computes the authentication path for ℓ as defined in the algorithm $\mathsf{catVrfy}$ up to the level $D-2$. Denote by x'_ρ the resulting value. Then, $\mathsf{addLeaf}$ applies the trapdoor csk to the root node ρ to obtain the matching randomness r'_ρ, i.e., $r'_\rho \leftarrow \mathsf{Col}(csk, x_\rho, r_\rho, x'_\rho)$ and sets $R = (r'_\rho, r)$. The algorithm computes the corresponding authentication path for the leaf ℓ as $\mathsf{aPath} = ((v_{h+1,1}, \ldots, v_{D-1,1}), R)$, it sets the counter $c \leftarrow 2$, and the state $\mathtt{st'} \leftarrow (c, D, x'_\rho, r'_\rho, [\mathbf{x}, \mathbf{r}], \ell_0, \ell_1)$. The algorithm returns $\mathtt{sp}' = (csk, \mathtt{st}')$, the index 0, and the authentication path aPath.

```
$c \leftarrow c + 2$
for $h = 1$ to $D - 2$ do
  if $\lfloor c/2^h \rfloor$ is even then
    if $(v_{h,\lfloor c/2^h \rfloor+1}) \notin \mathsf{st}$ then
      $x_{h,\lfloor c/2^h \rfloor+1} \leftarrow \{0,1\}^{2\mathsf{len}}$
      $r_{h,\lfloor c/2^h \rfloor+1} \leftarrow \{0,1\}^{\lambda}$
      $v_{h,\lfloor c/2^h \rfloor+1} = \mathsf{Ch}(x_{h,\lfloor c/2^h \rfloor+1}; r_{h,\lfloor c/2^h \rfloor+1})$
      $\mathsf{st.add}(x_{h,\lfloor c/2^h \rfloor+1}, r_{h,\lfloor c/2^h \rfloor+1})$
      $\mathsf{aPath.add}(v_{h,\lfloor c/2^h \rfloor+1})$
    else
      $\mathsf{aPath.add}(v_{h,\lfloor c/2^h \rfloor+1})$
    end if
  end if
  if $\lfloor c/2^h \rfloor$ is odd then
    if $(v_{h,\lfloor c/2^h \rfloor}) \in \mathsf{st}$ then
      $x'_{h,\lfloor c/2^h \rfloor} = (v_{h-1,\lfloor c/2^{h-1} \rfloor} || v_{h-1,\lfloor c/2^{h-1} \lfloor +1})$
      $r'_{h,\lfloor c/2^h \rfloor} \leftarrow \mathsf{Col}(csk, x_{h,\lfloor c/2^h \rfloor}, r_{h,\lfloor c/2^h \rfloor}, x'_{h,\lfloor c/2^h \rfloor})$
      $v_{h,\lfloor c/2^h \rfloor-1} = H(v_{h-1,\lfloor c/2^{h-1} \rfloor-2} || v_{h-1,\lfloor c/2^{h-1} \rfloor-1})$
      $R.\mathsf{add}(r'_{h,\lfloor c/2^h \rfloor})$
      $\mathsf{st.add}(r'_{h,\lfloor c/2^h \rfloor})$
      $\mathsf{aPath.add}(v_{h,\lfloor c/2^h \rfloor-1})$
      $\mathsf{st.del}(v_{h-1,\lfloor c/2^{h-1} \rfloor-2}, v_{h-1,\lfloor c/2^{h-1} \rfloor-1}, x_{h,\lfloor c/2^h \rfloor}, r_{h,\lfloor c/2^h \rfloor})$
    else
  end if
end for
$R.\mathsf{add}(r)$
output $(\mathsf{sp}', c, (\mathsf{aPath}, R))$
```

Figure 4: Algorithm to generate the authentication path.

$c > 0$**:** The algorithm $\mathsf{addLeaf}$ gets the counter c from the state st, creates a new list aPath and proceeds as defined in Figure 4.

$\mathsf{catVrfy}(\mathsf{vp}, i, \ell, \mathsf{aPath})$**:** The input of the path verification algorithm is a public key $\mathsf{vp} = (cpk, \rho)$, the index i of the leaf ℓ, and the authentication path $\mathsf{aPath} = ((v_{1,\lfloor i/2 \rfloor}, \ldots, v_{D-2,\lfloor i/2^{D-2} \rfloor}), R)$, where R is a non-empty set that contains all randomness that are necessary to compute the chameleon hash functions. The verification algorithm sets $(\ell_i, \ell_{i+1}) \leftarrow \mathsf{Ch}_1(\ell; r)$ and computes the node $v_{h,i}$ for $h = 2, \ldots, D-2$ as follows:

If $\lfloor i/2^h \rfloor \equiv 1 \mod 2$**:**

$$x \leftarrow v_{h-1,\lfloor i/2^{h-1} \rfloor} || v_{h-1,\lfloor i/2^{h-1} \rfloor+1},$$
$$v_{h,i} \leftarrow \mathsf{Ch}\left(x; r_{h,\lfloor i/2^h \rfloor}\right), \text{ with } r_{h,\lfloor i/2^h \rfloor} \in R.$$

If $\lfloor i/2^h \rfloor \equiv 0 \mod 2$**:**

$$x \leftarrow v_{h-1,\lfloor i/2^{h-1} \rfloor-2} || v_{h-1,\lfloor i/2^{h-1} \rfloor-1},$$
$$v_{h,i} \leftarrow H(x).$$

Finally, the verifier computes the root node $\hat{\rho}$ as $\hat{\rho} \leftarrow \mathsf{Ch}(v_{D-2,0} || v_{D-2,1}; r_\rho)$ (with $r_\rho \in R$). If $\hat{\rho} = \rho$, then the leaf is authenticated, and otherwise rejected.

6.1 Intuition of the Security Proof

To explain the proof idea, consider an efficient adversary $\mathcal{A}$ that inserts at most $q := q(\lambda)$ leaves. Since the adversary is efficient, the number of leaves are polynomially bounded. The idea is to store the q leaves $\ell_1, \ldots, \ell_q$ in the tree and then to choose dummy nodes such that the entire tree has polynomial depth $D = \mathsf{poly}(\lambda)$. Notice that the entire tree does not exist at any time (cf. the tree shown in Figure 5), i.e., the subtree consists of the leaves $\ell_1, \ldots, \ell_q$, but the gray nodes, and the dotted nodes in the tree are dummy nodes. Now, recall that the adversary wins if it outputs a tuple $(\ell^*, i^*, \mathsf{aPath}^*) \notin Q$. We distinguish between the case where $1 \le i^* \le q$ and where $q + 1 \le i^* \le 2^D$.

In the first part of the proof, where $1 \le i^* \le q$, we show how to find a collision in either (1) the hash function or (2) the chameleon hash function. In the second case where we assume that $q + 1 \le i^* \le 2^D$, we further distinguish between the cases where either (2.1) the adversary inverts the chameleon hash function or (2.2) it finds a collision in it.

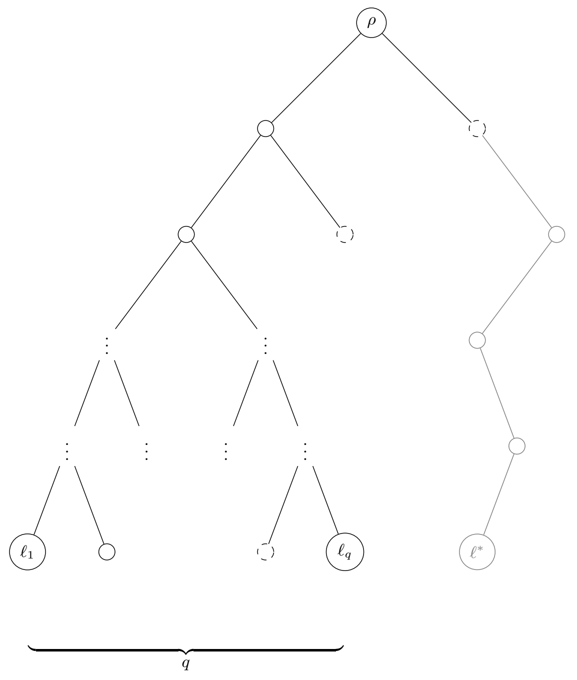


Figure 5: This figure shows how we set up the tree in the proof. The gray path corresponds to the case where the adversary outputs a pair $(\ell^*, i^*, \mathsf{aPath}^*)$ such that $i^* > q$.

The main observation in the second part is that the path from the leaf ℓ^* to the root ρ must contain a *right-handed* node on the authentication path of the node ℓ_q. In particular, this node must be one of the dummy nodes. Since we only create a polynomial number of dummy nodes, we can guess which of these nodes is contained in the path. If the reduction guesses this index correctly, then it can embed the challenge of the one-wayness game. Notice, that the adversary might compute a different pre-image. In this case, however, we break the collision-resistance of the chameleon hash function. Notice that embedding this challenge is only possible, because of the careful construction of the tree as discussed in Section 5.2. In particular, this technique would *not* work if we would use the trapdoor of the chameleon hash function in a different way. Observe, that the tree still authenticates an exponential number of leaves (even if $\mathcal{A}$ is only capable of adding polynomial number of leaves to the tree).

THEOREM 1. *If H is a collision-resistant hash function and $\mathcal{CH}$ a one-way collision-resistant chameleon hash function, then Construction 1 is a chameleon authentication tree with depth $D = \mathsf{poly}(\lambda)$ that is structure-preserving and one-way.*

The proof is giving in the full version [SS12].

7. CONSTRUCTION OF A VERIFIABLE DATA STREAMING PROTOCOL

Our VDS is not a completely black-box construction from a CAT, because updating leaves is not supported by a CAT in general. Instead, we use the algorithms of a CAT whenever it is possible and exploit the concrete structure of our scheme when we describe the update mechanism and also in the proof.

The main idea of our construction is to split the data in the CAT between the server $\mathcal{S}$ and the client $\mathcal{C}$. That is, the client basically stores the trapdoor and the authentication path of the current value. As discussed in Section 5.3, this information is sufficient to authenticate the next leaf. The server, however, stores the entire tree (as it has been specified so far) and the randomness of all chameleon hashes learned so far. As an example, consider the tree in Figure 1. The client stores the blue authentication path including the values $(x_{3,1}, r_{3,1})$ and $(x_{1,3}, r_{1,3})$ of the two "unused" inner nodes and the trapdoor csk, while the sever stores the entire tree.

To retrieve any element from the database, the client sends the index i to the server who returns the element $s[i]$ together with the corresponding authentication path $\pi_{s[i]} = \mathsf{aPath}_{s[i]}$. Verifying works straightforwardly by checking the authentication path.

Updating an element $s[i]$ to $s'[i]$ in DB work as follows: First, $\mathcal{C}$ runs the query algorithm to obtain the element $s[i]$ and the corresponding authentication path $\pi_{s[i]} = \mathsf{aPath}_{s[i]}$. If the verification algorithm $\mathsf{Verify}(PK, i, s, \pi_{s[i]})$ evaluates to 1, then $\mathcal{C}$ updates the leaf $\ell_i = s[i]$ to $s[i]'$. Subsequently, it updates the authentication path $\mathsf{aPath}_{\ell[i]}$ to $\mathsf{aPath}_{s'[i]}$ analogously to the algorithm $\mathsf{wcatVrfy}$ as defined in Construction 1. Denote by ρ' the resulting value. The client $\mathcal{C}$ sets $\rho \leftarrow \rho'$ in its public key PK and sends the new authentication path $\mathsf{aPath}_{s'[i]}$ to $\mathcal{S}$. The server updates the entry in DB, all leaves, and the root, which results in a new public key PK'.

Construction 2. Let $\mathsf{CAT} = (\mathsf{catGen}, \mathsf{addLeaf}, \mathsf{catVrfy})$ be the chameleon authentication tree as defined in Construction 1. We define the verifiable data streaming protocol $\mathcal{VDS} = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Update})$ as follows:

$\mathsf{Setup}(1^\lambda)$**:** The setup algorithm picks some $D = \mathsf{poly}(\lambda)$ and generates the CAT $(\mathsf{sp}, \mathsf{vp}) \leftarrow \mathsf{catGen}(1^\lambda, D)$ as defined in Construction 1. In particular, the private key is $SK = \mathsf{sp} = (csk, csk_1, \mathtt{st})$ and the public key is $PK = \mathsf{vp} = (cpk, cpk_1, \rho)$, where ρ is the root of the initially empty tree. The client $\mathcal{C}$ gets the private key SK and the server the public key PK. The server also sets up an initially empty database DB.

$\mathsf{Append}(\boldsymbol{SK}, s)$**:** To append an element s to DB, the client $\mathcal{C}$ runs the algorithm $\mathsf{addLeaf}(\mathsf{sp}, s)$ locally which returns a key sp', an index i, and an authentication path aPath_i. It sends i, s and aPath_i to the sever $\mathcal{S}$. The server appends s to DB, it adds the unknown nodes from $\mathsf{aPath}_i = ((v_{1,\lfloor i/2 \rfloor}, \ldots, v_{D-2,\lfloor i/2^{D-2} \rfloor}), R)$ to its tree, and stores the new randomness from R.

$\mathsf{Query}(\boldsymbol{PK}, \boldsymbol{DB}, i)$**:** The client sends the index i to the server who responses with $s[i]$ and the corresponding authentication path $\pi_{s[i]} = \mathsf{aPath}_i$, or with $\perp$ if the i^{th} entry in DB is empty.

$\mathsf{Verify}(\boldsymbol{PK}, i, s, \pi_{s[i]})$**:** The verification algorithm parses PK as vp and $\pi_{s[i]}$ as $\mathsf{aPath}_{s[i]}$, it returns s if the algorithm $\mathsf{catVrfy}(\mathsf{vp}, i, s, \pi_i)$ outputs 1. Otherwise, it outputs $\perp$.

$\mathsf{Update}(\boldsymbol{PK}, \boldsymbol{DB}, \boldsymbol{SK}, i, s')$**:** To update the i^{th} element in DB to s', the clients parses SK as the trapdoor sp of the CAT, its state $\mathtt{st}$, and the pairs $(x_{i,j}, r_{i,j})$ of the "unused" nodes computed via the chameleon hash function ("unused" means that the trapdoor has not been applied to these nodes). The first move in the protocol is by the client who sends the index i to $\mathcal{S}$. The server returns $s[i]$ and the corresponding proof $\pi_{s[i]}$ (which is the authentication path aPath_i). The client $\mathcal{C}$ runs $\mathsf{Verify}(PK, i, s[i], \pi_{s[i]})$ to check the validity of $s[i]$. If Verify returns $\perp$, then $\mathcal{C}$ aborts. Otherwise, it sets the leaf $\ell_i = s[i]$ to s' and re-computes the authentication path with the new leaf (as described in Construction 1). The output of this algorithm is a new root ρ'. Subsequently, the client updates all nodes that are stored in its state $\mathtt{st}$, but that have been updated by re-computing the authentication path with the new leaf (this includes at least the root ρ and thus, the verification key PK). Notice that the randomness used for the chameleon hash functions remain the same. Then, $\mathcal{C}$ sends the new authentication path aPath'_i, the updated leaf s', and the updated verification key PK' to the server. The server first verifies the authentication path. If it is valid, then $\mathcal{S}$ updates the stored value $s[i]$ to $s[i]'$, the corresponding nodes in the CAT, as well as its verification key PK'. Otherwise, it aborts.

Regarding security, we prove the following theorem:

THEOREM 2. *If H is a collision-resistant hash function and $\mathcal{CH} = (\mathsf{Gen}, \mathsf{Ch}, \mathsf{Col})$ a chameleon hash function that is one-way and collision-resistant, then Construction 2 is a secure verifiable data streaming protocol w.r.t. Definition 2.*

The proof is giving in the full version [SS12].

8. CONCRETE INSTANTIATION WITH FASTER VERIFICATION

Our construction can be instantiated with any chameleon hash function, but we choose the one of Krawczyk and Rabin (in the following called KR chameleon hash function) [KR00].

The scheme is secure under the discrete logarithm assumption (in the standard model)[1], which is a very appealing and mild assumption. Furthermore, the algebraic properties allow us to construct a very efficient path verification algorithm by applying batch verification techniques.

Assumption 1. Let q be a prime and let $p = 2q + 1$ be a strong prime. Let $\mathbb{G}$ be the unique cyclic subgroup of $\mathbb{Z}_p^*$ of order q and let g be a generator of $\mathbb{G}$. Then, the discrete logarithm problem holds if for all efficient algorithms $\mathcal{A}$ the probability

$$\text{Prob}\left[\begin{array}{l} y \leftarrow \mathbb{G}\,; \alpha \leftarrow \mathcal{A}(p, q, g, y) : \\ 0 \le \alpha \le q-1 \wedge g^{\alpha} \equiv y \mod p \end{array}\right] = \nu(\lambda)$$

is negligible (as a function of λ).

8.1 Building Block

The KR chameleon hash function $\mathcal{CH} = (\mathsf{Gen}, \mathsf{Ch}, \mathsf{Col})$ is defined as follows:

Gen(1^λ): The key generation algorithm picks a prime q such that $p = 2q+1$ is also prime. It also chooses a random generator g and a value $\alpha \in \mathbb{Z}_q^*$. It returns $(csk, cpk) \leftarrow ((\alpha, g), (X, g, p, q))$ with $X = g^{\alpha} \mod p$.

Ch$(\boldsymbol{cpk}, x)$: The input of the hash algorithm is a key $cpk = (X, g, p, q)$ and a message $x \in \mathbb{Z}_q^*$. It picks a random value $r \in \mathbb{Z}_q^*$ and outputs $g^x X^r \mod p$.

Col$(\boldsymbol{csk}, x, r, x')$: The collision finding algorithm returns $r' \leftarrow \alpha^{-1}(x - x') + r \mod q$.

8.2 Batch Verification of Chameleon Hash Functions

The most expensive operation in a CAT of depth D is the verification of an authentication path. This computation involves (in the worst case) D computations of the chameleon hash function (which is the authentication of the last leaf). For our concrete instantiation this means that the verification algorithm verifies D times equations of the form $h_i = g^{x_i} X^{r_i}$. This step is rather expensive as it involves many modular exponentiations. Instead of verifying all equations straightforwardly, we apply *batch verification* techniques as introduced by Bellare, Garay, and Rabin [BGR98]. The basic idea is to verify sequences of modular exponentiations significantly faster than the naïve re-computation method. In what follows, let λ_b be the security parameter such that the probability of accepting a batch that contains an invalid hash is at most $2^{-\lambda_b}$. Note, that it is necessary to test that all elements belong to the group $\mathbb{G}$ as discussed comprehensively by Boyd and Pavlovski in [BP00]. The size of λ_b is a trade off between efficiency and security. Therefore, it depends heavily on the application. Camenisch, Hohenberger, and Pedersen suggest $\lambda_b = 20$ bits for a rough check and $\lambda_b = 64$ bit for higher security [CHP07].

Definition 8. A batch verifier for a relation R is a probabilistic algorithm V that takes as input (possibly a description of R) a batch instance $X = (inst_1, \ldots, inst_D)$ for R, and a security parameter λ. It satisfies:

(1) If X is correct, then V outputs 1.

(2) If X is incorrect, then the probability that V outputs 1 is at most $2^{-\lambda_b}$.

The naïve batch verifier re-computes all instances, i.e., it consists of computing $R(inst_i)$ for each $i = 1, \ldots, D$, and checking that each of these D values is 1.

8.2.1 Small Exponent Test for Chameleon Hash Functions

Bellare, Garay, and Rabin suggest three different methods of computing batch verification for modular exponentiations [BGR98]. Here, we focus only on the small exponent test because it is the most efficient one for the verification of *up to* 200 elements. In our scenario, 200 elements means that we can authenticate 2^{200} elements. The authors consider equations of the form $y_i = g^{x_i}$. The naïve approach would be to check if $\prod_D y_i = g^{\sum_D x_i}$. This, however, is not sufficient as it is easy to produce two pairs (x_1, y_1) and (x_2, y_2) that pass the verification but each individual does not. One example of such a pair is $(x_1 - \beta, y_1)$ and $(x_2 + \beta, y_2)$ for any β.

According to [FGHP09], the small exponent test works as follows: Pick D exponents δ_i of a small number of $\{0,1\}^{\lambda_b}$ at random and compute $x \leftarrow \sum_D x_i \delta_i \mod q$ and $y \leftarrow \prod_D y_i^{\delta_i}$. Output 1 iff $g^x = y$. The probability of accepting a bad pair is $2^{-\lambda_b}$ and the value λ_b is a trade off between efficiency and security.

8.3 Batch Verification of KR Chameleon Hash Function

The algorithm Batch that performs the batch verification of D chameleon hash values $h_1, \ldots, h_D$ on messages $x_1, \ldots, x_D$ using the randomness $r_1, \ldots, r_D$ works as follows: It first checks that all elements are in the group. If not, then it rejects the query. Otherwise, it picks D random elements $\delta_1, \ldots, \delta_D$ where $\delta_i \in \{0,1\}^{\lambda_b}$ and checks that

$$g^{\sum_D x_i \delta_i} X^{\sum_D r_i \delta_i} = \prod_D h_i^{\delta_i}.$$

It outputs 1 if the equation holds and otherwise 0.

Theorem 3. *The algorithm* Batch *is a batch verifier for the KR chameleon hash function.*

The following proof follows the proofs of [CHP07, BGR98].

Proof. We first show that if all hash values have the desired form, then our batch verification algorithm accepts with probability 1. Keeping in mind that the validation of the hash function checks that $h_i = g^{x_i} X^{r_i}$, then we can easily show that

$$\begin{aligned} g^{\sum_D x_i \delta_i} X^{\sum_D r_i \delta_i} &= \prod_D h_i^{\delta_i} = \prod_D (g^{x_i} X^{r_i})^{\delta_i} \\ &= \prod_D g^{x_i \delta_i} X^{r_i \delta_i} = g^{\sum_D x_i \delta_i} X^{\sum_D r_i \delta_i}. \end{aligned}$$

The next step is to show that the other direction is also true. To do so, we apply the technique for proving small exponents test as in [BGR98]. Since our batch verification algorithm accepts, it follows that $h_i \in \mathbb{G}$. This allows us to write $h_i = g^{\rho_i}$ for some $\rho_i \in \mathbb{Z}_q$. Moreover, we know that $X = g^x$ for some $x \in \mathbb{Z}_q$. We then can re-write the above

[1]To the best of our knowledge, this instantiation is currently the most efficient one based on the discrete logarithm assumption in the standard model.

equation as

$$\prod_D h_i^{\delta_i} = g^{\rho_i \delta_i} = g^{\sum_D \delta_i(m_i+\alpha r_i)}$$

$$\Rightarrow \sum_D \rho_i\delta_i = \sum_D \delta_i(x_i+\alpha r_i)$$

$$\Rightarrow \sum_D \rho_i\delta_i - \sum_D \delta_i(x_i+\alpha r_i) \equiv 0 \mod q.$$

Setting $\beta_i = \rho_i - (x_i + \alpha r_i)$ this is equivalent to:

$$\prod_D \delta_i\beta_i \equiv 0 \mod q. \quad (1)$$

Now, assume that $\mathsf{Batch}((h_1, x_1, r_1), \ldots, (h_D, x_D, r_D))$ outputs 1, but there exists an index $i = 1$ (this holds w.l.o.g.) such that $g^{x_1\delta_1} X^{r_1\delta_1} \neq h_1^{\delta_1}$. In particular, this means that $\beta_1 \neq 0$. Since q is prime, then β_1 is the inverse of γ_1 such that $\beta_1\gamma_1 \equiv 1 \mod q$. Taking this and Equation (1), we obtain

$$\delta_1 = -\gamma_1 \sum_{i=2}^{D} \delta_i\beta_i \mod q. \quad (2)$$

Now, given the elements $((h_1, x_1, r_1), \ldots, (h_D, x_D, r_D))$ such that $\mathsf{Batch}((h_1, x_1, r_1), \ldots, (h_D, x_D, r_D)) = 1$ and let bad denote the event that we break the batch verification, i.e., $g^{x_1\delta_1} X^{r_1\delta_1} \neq h_1^{\delta_1}$. Observe that we do not make any assumptions about the remaining values. Let $\Delta' = \delta_2, \ldots, \delta_D$ and let $|\Delta'|$ be the number of possible values for this vector. It follows from Equation 2 and from the fact that Δ' is fixed that there exists exactly one value δ_1 that will make bad happen. This means, however, that the probability that bad occurs is $\text{Prob}[\,\mathsf{bad} \mid \Delta'\,] = 2^{-\lambda_b}$. Choosing the value δ_1 at random and summing over all possible choices of Δ', we get $\text{Prob}[\,\mathsf{bad}\,] \leq \sum_{i=1}^{\Delta'}(\text{Prob}[\,\mathsf{bad} \mid \Delta'\,] \cdot \text{Prob}[\,\Delta'\,])$. Thus, we can calculate the overall probability as $\text{Prob}[\,\mathsf{bad}\,] \leq \sum_{i=1}^{2^{\lambda_b(D-1)}}(2^{\lambda_b} \cdot 2^{-\lambda_b(D-1)}) = 2^{-\lambda_b}$. $\square$

8.4 Efficiency

We analyze the efficiency of our batch verifier for the KR chameleon hash using the following notation. By $\mathsf{exp}(k_1)$ we denote the time to compute g^b in the group $\mathbb{G}$ where $|b| = k_1$. The efficiency is measured in number of multiplications. First, we have to compute $\prod_a h_i^{\delta_i}$. Instead of computing this product straightforwardly, we apply the algorithm $\mathsf{FastMult}((h_1, \delta_1), \ldots, (h_D, \delta_D))$ obtaining a total number of $\lambda_b + D\lambda_b/2$ multiplications on the average [BGR98]. In addition we have to compute $2D$ multiplications and finally $2\mathsf{exp}(k_1)$ exponentiations. Thus, the total number of multiplications is $\lambda_b + D(2 + \lambda_b/2) + 2\mathsf{exp}(k_1)$.

8.5 Benchmarking Results

We estimate the performance of our scheme by analyzing the most expensive component of our construction. That is, we have implemented the KR chameleon hash function and we use the implementation of SHA1 provided by the Java security package. These are the two main components of our construction. The additional overhead determining the nodes should add only a negligible overhead to the overall computational costs (recall that computing a chameleon hash involves modular exponentiations, which we believe is the most expensive step). We have implement the KR chameleon hash in Java 1.6 on a Intel Core i5 using 4GB 1333MHz DDR3 RAM. We have conducted two different experiments where we executed each algorithm 500 times with a CAT of depth 80 (thus it authenticates 2^{80} elements). The bit length of the primes in the first experiment is 1024 bits and in the second 2048 bits. The following values are the average computational costs. Adding a leaf to the tree in the worst case (this happens when the tree is empty) takes on average 283ms for 1024 bits and 1400ms for 2048 bits. Each of these executions involves 40 evaluations of the chameleon hash (including the generation of randomness), 40 SHA1 computations, and the computation of a collision. The timings to verify a path and to update it are slightly faster, because both operations do neither include the generation of random values, nor the computation of a collision. In the full version of this paper, we will include running times of the full implementation.

9. FROM ONE-TIME TO MANY-TIME SIGNATURE SCHEMES USING CATS

The second application of CATs is a new transformation that turns any one-time signature (OTS) scheme into a many-time signature (MTS) scheme. A one-time signature scheme allows a user to sign a *single* message. This primitive is well-known and has been introduced by Lamport [Lam79] and Rabin [Rab79]. One-time signature schemes are interesting from both, a theoretical and a practical point of view. Theoreticians study the primitive because it is used as a building block in realizing secure signature schemes based on one-way functions (in a black-box way). On the other hand, several extremely efficient instantiations have been suggested in practice. However, the "one-timeness" of such schemes regrettably causes a complex key-scheduling process since the signer has to generate a new key pair whenever it wishes to sign a new message. Motivated by this drawback – and also by the hope of obtaining an efficient many-time signature scheme – several publications investigate efficient transformations from a OTS to a MTS scheme.

9.1 Construction

The basic idea of our construction is to store the keys of the one-time signature scheme in the leaves of the CAT. Since the keys are generated uniformly at random (and in particular, independent of the message to be signed) a weaker security notion than the one giving in Definition 4.2 is sufficient. We refer the reader to the full version [SS12].

Construction 3. Let $\mathsf{Sig} = (\mathsf{Gen}, \mathsf{Sign}, \mathsf{Vrfy})$ be a signature scheme defined over the message space $\mathcal{M} = \{0,1\}^\lambda$ and let $\mathsf{wCAT} = (\mathsf{wcatGen}, \mathsf{waddLeaf}, \mathsf{wcatVrfy})$ be a chameleon authentication tree. We define the signature scheme $\mathsf{cSig} = (\mathsf{cGen}, \mathsf{cSign}, \mathsf{cVrfy})$ as follows:

$\mathsf{cGen}(1^\lambda)$**:** The key generation algorithm runs $(\mathsf{wsp}, \mathsf{wvp}) \leftarrow \mathsf{wcatGen}(1^\lambda)$. It returns the private key $SK \leftarrow \mathsf{wsp}$, and the corresponding public key $PK \leftarrow \mathsf{wvp}$.

$\mathsf{cSign}(\boldsymbol{SK}, m)$**:** To sign a message $m \in \{0,1\}^\lambda$, the signing algorithm generates a key pair $(SK', PK') \leftarrow \mathsf{Gen}(1^\lambda)$, signs the message $\sigma_0 \leftarrow \mathsf{Sign}(SK', m)$, and adds the public key to the CAT by computing $(\mathsf{wsp}', i, \mathsf{waPath}) \leftarrow \mathsf{waddLeaf}(\mathsf{wsp}, PK')$. It sets $\sigma_1 \leftarrow (i, \mathsf{waPath})$ and returns the signature $\sigma \leftarrow (\sigma_0, \sigma_1, PK')$.

$\mathsf{cVrfy}(\boldsymbol{PK}, m, \sigma)$**:** The verification algorithm parses $PK = \mathsf{wvp}$ and $\sigma = (\sigma_0, \sigma_1, PK')$ and $\sigma_1 = (i, \mathsf{waPath})$. It outputs 1 if both condition hold: $\mathsf{catVrfy}(\mathsf{wvp}, i, PK',$ $\mathsf{aPath}) = 1$ and $\mathsf{Vrfy}(PK', m, \sigma_0) = 1$.

As already discussed in Section 6.1, the security of our construction holds for a tree that authenticates an exponential number of 2^D signatures. Again, this follows from our construction (as we never store all elements at the same time). Let $q := q(\lambda)$ be an upper bound on the number of signing queries from the adversary. Then, we distinguish the case where the adversary outputs a forgery for a leaf $1 \leq i^* \leq q$ and the case where $q + 1 \leq i^* \leq 2^D$. In the first case, where $1 \leq i^* \leq q$ is, we build an adversary that either breaks the structure-preserving property of the CAT or that forges the one-time signature scheme. In the second case, where $q + 1 \leq i^* \leq 2^D$ is, we show how to beak the one-wayness the CAT.

THEOREM 4. *If* Sig *is a secure one-time signature scheme and* CAT *a weakly structure-preserving and weakly one-way chameleon authentication tree, then Construction 3 is unforgeable under adaptive chosen message attacks.*

The proof is giving in the full version of this paper [SS12].

Acknowledgements

We are very thankful to Nick Hopper for his comprehensive suggestions. We thank Marc Fischlin for suggesting batch verification techniques to improve the running time of our verification algorithm. We also thank the anonymous reviewers for valuable and comprehensive comments. This work was supported by the German Ministry for Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA — `www.cispa-security.de`) and by CASED — `www.cased.de`. This work was partially supported by the US Army Research Laboratory and the UK Ministry of Defense under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Army Research Laboratory, the US Government, the UK Ministry of Defense, or the UK Government. The US and UK Governments are authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

10. REFERENCES

[Ad04] Giuseppe Ateniese and Breno de Medeiros. On the key exposure problem in chameleon hashes. In Carlo Blundo and Stelvio Cimato, editors, *SCN 04: 4th International Conference on Security in Communication Networks*, volume 3352 of *Lecture Notes in Computer Science*, pages 165–179, Amalfi, Italy, September 8–10, 2004. Springer, Berlin, Germany.

[BDK$^+$07] Johannes Buchmann, Erik Dahmen, Elena Klintsevich, Katsuyuki Okeya, and Camille Vuillaume. Merkle signatures with virtually unlimited signature capacity. In Jonathan Katz and Moti Yung, editors, *ACNS 07: 5th International Conference on Applied Cryptography and Network Security*, volume 4521 of *Lecture Notes in Computer Science*, pages 31–45, Zhuhai, China, June 5–8, 2007. Springer, Berlin, Germany.

[BGR98] Mihir Bellare, Juan A. Garay, and Tal Rabin. Fast batch verification for modular exponentiation and digital signatures. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 236–250, Espoo, Finland, May 31 – June 4, 1998. Springer, Berlin, Germany.

[BGV11] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 111–131, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Berlin, Germany.

[BP00] Colin Boyd and Chris Pavlovski. Attacking and repairing batch verification schemes. In Tatsuaki Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 58–71, Kyoto, Japan, December 3–7, 2000. Springer, Berlin, Germany.

[CF11] Dario Catalano and Dario Fiore. Vector commitments and their applications. Cryptology ePrint Archive, Report 2011/495, 2011. `http://eprint.iacr.org/`.

[CHP07] Jan Camenisch, Susan Hohenberger, and Michael Østergaard Pedersen. Batch verification of short signatures. In Moni Naor, editor, *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *Lecture Notes in Computer Science*, pages 246–263, Barcelona, Spain, May 20–24, 2007. Springer, Berlin, Germany.

[CKLR11] Kai-Min Chung, Yael Tauman Kalai, Feng-Hao Liu, and Ran Raz. Memory delegation. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 151–168, Santa Barbara, CA, USA, August 14–18, 2011. Springer, Berlin, Germany.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 481–500, Irvine, CA, USA, March 18–20, 2009. Springer, Berlin, Germany.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 61–76, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Berlin, Germany.

[CMT12] Graham Cormode, Michael Mitzenmacher, and

Justin Thaler. Practical verified computation with streaming interactive proofs. In *Innovations in Theoretical Computer Science (ITCS)*, 2012.

[FB06] Décio Luiz Gazzoni Filho and Paulo Sérgio Licciardi Messeder Barreto. Demonstrating data possession and uncheatable data transfer. Cryptology ePrint Archive, Report 2006/150, 2006. `http://eprint.iacr.org/`.

[FGHP09] Anna Lisa Ferrara, Matthew Green, Susan Hohenberger, and Michael Østergaard Pedersen. Practical short signature batch verification. In Marc Fischlin, editor, *Topics in Cryptology – CT-RSA 2009*, volume 5473 of *Lecture Notes in Computer Science*, pages 309–324, San Francisco, CA, USA, April 20–24, 2009. Springer, Berlin, Germany.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[KR00] Hugo Krawczyk and Tal Rabin. Chameleon signatures. In *ISOC Network and Distributed System Security Symposium – NDSS 2000*, San Diego, California, USA, February 2–4, 2000. The Internet Society.

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, October 1979.

[Mer88] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology – CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Santa Barbara, CA, USA, August 16–20, 1988. Springer, Berlin, Germany.

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *Advances in Cryptology – CRYPTO'89*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Berlin, Germany.

[MND$^+$01] Chip Martel, Glen Nuckolls, Prem Devanbu, Michael Gertz, April Kwong, and Stuart G. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39:2004, 2001.

[Nao91] Moni Naor. Bit commitment using pseudorandomness. *Journal of Cryptology*, 4(2):151–158, 1991.

[Ngu05] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292, San Francisco, CA, USA, February 14–18, 2005. Springer, Berlin, Germany.

[NN00] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *21st Annual ACM Symposium on Theory of Computing*, pages 33–43, Seattle, Washington, USA, May 15–17, 1989. ACM Press.

[oCMoP12] University of California Museum of Paleontology. The effects of mutations. understanding evolution., 2012. Last access 05/03/12 - `http://evolution.berkeley.edu/evolibrary/article/0_0_0/mutations_05`.

[PT07] Charalampos Papamanthou and Roberto Tamassia. Time and space efficient algorithms for two-party authenticated data structures. In *Proceedings of the 9th international conference on Information and communications security*, ICICS'07, pages 1–15, Berlin, Heidelberg, 2007. Springer-Verlag.

[Rab79] Michael O. Rabin. Digital signatures and public key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, Massachusetts Institute of Technology, January 1979.

[RLB$^+$08] Andy Rupp, Gregor Leander, Endre Bangerter, Alexander W. Dent, and Ahmad-Reza Sadeghi. Sufficient conditions for intractability over black-box groups: Generic lower bounds for generalized DL and DH problems. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 489–505, Melbourne, Australia, December 7–11, 2008. Springer, Berlin, Germany.

[SM06] Thomas Schwarz and Ethan L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. Proceedings of the IEEE Int'l Conference on Distributed Computing Systems (ICDCS '06), July 2006.

[SS12] Dominique Schröder and Heike Schröder. Verifiable data streaming. Cryptology ePrint Archive, Report 2012, 2012. Full version, available at `http://eprint.iacr.org/`.

[SW08] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 90–107, Melbourne, Australia, December 7–11, 2008. Springer, Berlin, Germany.

[Szy04] Michael Szydlo. Merkle tree traversal in log space and time. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 541–554, Interlaken, Switzerland, May 2–6, 2004. Springer, Berlin, Germany.

[TT10] Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. In *AMW*, 2010.