# VeriStream – A Framework for Verifiable Data Streaming

Dominique Schöder and Mark Simkin

Saarland University
Saarbrücken, Germany

**Abstract** In a Verifiable Data Streaming (VDS) protocol a computationally weak client outsources his storage to an untrusted storage provider. Later, the client can efficiently append and update data elements in the already outsourced and authenticated data set. Other users can stream arbitrary subsets of the authenticated data and verify their integrity on-the-fly, using the data owner's public verification key. In this work, we present VeriStream, a fully-fledged open source framework for verifiable data streaming with integration into Dropbox. At its core, our framework is based upon a novel construction of an authenticated data structure, which is the first one that allows verifiable data streams of unbounded length and at the same time outperforms the best known constructions in terms of bandwidth and computational overhead. We provide a detailed performance evaluation, showing that VeriStream only incurs a small bandwidth overhead, while providing various security guarantees, such as freshness, integrity, authenticity, and public verifiability, at the same time.

## 1 Introduction

Cloud storage providers like Dropbox, Amazon Cloud Drive, and Google Drive are on the rise and constantly gain popularity. Users are able to outsource their storage into the "cloud" of some dedicated provider and access or share their data with others later on. The advantages of cloud storage are manifold. Among many, users are no longer bound to specific devices or locations when accessing their data and users can share or collaborate on their data with others easily. Many of these providers allow their users to retrieve, i.e. stream, smaller subsets of the initially outsourced data set. In the case of multimedia content, prominent examples are YouTube and SoundCloud. They allow users to upload their audio and video files and share them with others. A different user can stream the whole uploaded file or just smaller parts of it. This streaming scenario is not solely limited to multimedia content. Another interesting example can be found in the stock market. Here, stock brokers base their purchasing decisions on the latest published stock quotes. These stock quotes are published by trusted stock managers and distributed through web services like Yahoo Finance or quote.com Brokers use these services to stream the latest published stock quotes and buy or sell stocks accordingly.

| | **Proof size** | **Client's state** | **Upload** time/space | **Update** time/space | **Streaming** time/space | **Unbounded** | **Security proof** |
|---|---|---|---|---|---|---|---|
| [22] | $\mathcal{O}(\log N)$ | $\mathcal{O}(\log N)$ | $\mathcal{O}(\log N)$ | $\mathcal{O}(\log N)$ | $\mathcal{O}(\log N)$ | ✗ | Standard |
| Dynamic | $\mathcal{O}(\log M)$ | $\mathcal{O}(1)$ | $\mathcal{O}(\log M)$ | $\mathcal{O}(\log M)$ | $\mathcal{O}(\log M)$ | ✓ | ROM |
| $\delta$-bounded | $\mathcal{O}(\log M)$ | $\mathcal{O}(1)$ | $\mathcal{O}(\log M)$ | $\mathcal{O}(\log M)$ | $\mathcal{O}(\log M)$ | ✗ | Standard |

**Table 1.** Comparison of existing and proposed CAT constructions. N is the upper bound of elements that can be authenticated, whereas M is the number of already authenticated elements. Security proof indicates whether the construction's proof of security is given in the standard model or whether it requires the random oracle model.

All these scenarios have in common that the users have to trust the storage provider that streams the content back to the requesting user. Currently, there are little or no mechanisms in place to protect and ensure the integrity of such dynamic streamed content. A first step towards solving this problem was done in [22], where the problem of Verifiable Data Streaming (VDS) was defined on a theoretical level. The authors provided a first solution based on generalized Merkle-Trees, so called Chameleon Authentication Trees (CATs), that allow a data owner, having a secret signing and a public verification key, to upload his content in a unidirectional fashion. That is, the owner can upload and append data to the existing data set by sending one message per chunk to the server, without needing to update his public verification key after each transmitted data chunk. In addition, the CAT allows the data owner to efficiently update arbitrary subsets of the authenticated outsourced data set, without the need to re-upload or re-authenticate any of the elements that are not updated. After an update, the verification key is updated to invalidate the stale data elements, however all other data elements remain authenticated under the new verification key.

## 1.1 Our Contribution

On the practical side, we present VeriStream, the first fully-fledged framework for providing streaming applications with security guarantees, such as stream authenticity, integrity, correct ordering of the streamed elements, public verifiability, and efficient updates simultaneously. The VeriStream standalone client can be used upload, update, and stream content from personal web servers. In addition, VeriStream allows its users to use their Dropbox account as the underlying storage layer. Apart from up- and downloading arbitrary files in an authenticated fashion, our framework also supports video and audio streaming with on-the-fly verification. In Section 5 we provide a detailed performance evaluation of VeriStream and compare its performance to the construction from [22].

On the theoretical side we improve upon the state-of-the-art for verifiable data streaming [22]. Their construction is upper bounded during the initialization by some parameter $N$, meaning that it can authenticate up to $N$ elements. Their construction incurs a computational and bandwidth overhead of $\mathcal{O}(\log N)$ for each outsourced, updated, or streamed element. The size of their client's state is $\mathcal{O}(\log N)$. In particular this means, either $N$ is chosen large, e.g. polynomial, to be able to authenticate a quasi unbounded amount of elements, which incurs a prohibitively large overhead, or $N$ is chosen small, which in turn means that the resulting construction can only authenticate a limited number of elements.

We propose two novel constructions. The first one, the fully-dynamic CAT, is the first scheme that can authenticate an *unbounded* number of elements and is secure in the random oracle model. The second one, the $\delta$-bounded CAT, has an upper bound on the number of elements it can authenticate, and we prove its security in the standard model. Both of our constructions only incur a computational and bandwidth overhead of $\mathcal{O}(\log M)$ for each outsourced, updated, or streamed element, where $M$ is the number of authenticated elements so far. Note that in general the number of outsourced elements $M$ is significantly smaller than the upper bound $N$. The size of our client's state is $\mathcal{O}(1)$. For a concise comparison of the existing and our proposed constructions see Table 1.

## 1.2 System Overview

In this section we outline the high-level workflow and usage of VeriStream based on the classic task of outsourcing and sharing data. An overview is given in Figure 1. The major entities are the data owner, other clients that may read data uploaded by the data owner, and the untrusted server storing the data. To allow an easy integration of our framework into already deployed systems, we designed VeriStream to coexist with the existing system. This means that our framework does not directly alter or modify any of the transmitted data, but only appends and strips its own additional data to and from the transmitted data chunks. All involved entities use VDS handlers to authenticate or verify transmitted data. When the data owner wants to upload some data to the server, he initializes his local VDS handler with his secret key. Rather than transmitting all data chunks directly to the server, they are passed through the VDS handler, which authenticates them on-the-fly by appending a proof of correctness to the data chunk. The retrieving server uses his VDS handler to strip the proof from each data chunk. The data itself is stored in a database and the proofs are stored in a CAT. It should be noted, that uploading data to the server does not require the owner to update his verification key after each chunk, which would put an unrealistic burden on the public directory or PKI that handles the keys.

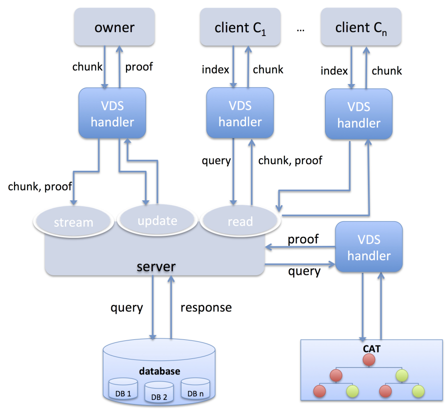


**Figure 1.** High-level overview of VeriStream.

A client can request data chunks by transmitting their indices. The server fetches the data chunks from the database and computes the corresponding proofs of correctness using his VDS handler, which has access to the CAT. The data and the appended proofs are sent to the client, who can verify the correctness and authenticity of each received data chunk separately on-the-fly, using the VDS handler and the data owner's public verification key.

When the data owner wants to update some data chunk in the database, he first receives the element the same way other clients do. After verifying the authenticity of the retrieved chunk, he uses his VDS handler in combination with the retrieved data chunk, its proof of correctness, and the new data chunk to compute a new proof of correctness for the new chunk. The new data chunk with the appended proof is then sent to the server. The owner updates his public verification key at the PKI or the public directory. This is necessary to invalidate the now stale chunk and protect other users from retrieving old data from the server. However, even though the owner only requested and modified the updated chunk, all other outsourced data chunks remain valid under the new verification key.

## 1.3 Related Work

Verifiable data streaming (VDS) protocols have been introduced by Schröder and Schröder [22]. The authors formalized the problem on a theoretical level and gave a first construction for a bounded number of elements. Their shortcomings in comparison to our constructions are already discussed in Section 1.1. A related line of research investigates verifiable databases (VDBs). VDBs have been extensively investigated in the context of accumulators [16,5,6] and authenticated data structures [15,14,19,26]. These approaches, however, often rely on non-constant assumptions, such as the $q$-Strong Diffie-Hellman assumption, as observed in [4]. More recent works, such as [4] or [8], only support a polynomial number of values instead of exponentially many, and the scheme of [4] is not publicly verifiable. Furthermore, the VDB schemes require the data owner to update his verification key after each newly uploaded element. In contrast, in a VDS protocol, data can be added non-interactively and without updating the verification key by sending a single message to the server. VDS can be seen as a generalization of VDBs.

Another line of research deals with (dynamic) proofs of retrievability (PoR). Here, the client uploads his data to some untrusted server. A PoR protocol allows the user to efficiently verify, whether all of his data is still stored on the server [24,10,23,7]. A weaker form of PoR are so called proofs of data possession [9]. They only ensure that the server stores most of the data. In these scenarios the protocols only ensure that all or most of the data is stored, but they do not provide any security guarantees w.r.t. the authenticity of streamed content.

Recently, the notion of streaming authenticated data structures was introduced by Papamanthou, Shi, Tamassia and Yi [18], where a computationally weak client and a server observe a stream of data independently. Afterwards the client can perform range queries and verify the results from the server against a verification value it computed while observing the stream. However both notions differ in the following aspects: The verification token of their scheme changes after each streamed/uploaded element, while ours does not. In their scheme, no secret key is involved, which means that a client can only verify responses if he has either seen the seen stream, or if he obtained the verification token from a trusted party. Furthermore, since the key changes after each new element, all elements that are transmitted after receiving the verification token cannot be verified. Our proofs are logarithmic in the size of the uploaded dataset, while theirs are logarithmic in the size of the universe from which the elements are drawn. Finally, we provide comprehensive benchmark results, while their work only provides a asymptotical run time analysis.

Another successful line of research consider "pure" streaming protocols between a sender and possibly multiple clients, such as TESLA and their variants such e.g., [21,20]. In contrast to our setting, the TESLA protocols assume that the sender and the receiver are loosely synchronized and these protocols do not offer public verifiability. The signature based solution of [21] is also different, because the protocol does not support efficient updates, which is a necessary property for our applications, such as e.g., verifiable cloud storage.

Naïve Approaches: There are a few seemingly simple solutions to the described problem, which do not work. In this paragraph we would like to discuss the shortcomings of some of them. The first idea might be to use a simple Merkle Tree. In a Merkle tree the data is stored in the leaves and the value of each internal node is defined as the hash of the concatenation of its children's values. The verification key is the value of the root node and a proof of correctness for some data chunk consists of all nodes that are required to compute the root node's value starting from the leaf, where the data chunk is stored. This solution would recompute the tree after each uploaded element. This means, that whenever we upload some new data chunk, the verification key is updated, which puts an infeasible burden on the public directory or PKI that stores the public keys. A different approach would be to use signature chains, i.e. for all two adjacent data chunks we compute one signature. Here, the problem is that efficient updates are not possible. When updating a data chunk, we need to invalidate its old version, but here the verification key is just the signature's public key and does not depend on the uploaded data itself. The data owner would need to update the signature's public key and recompute all signature in the chain, which is clearly infeasible. The same argument also holds for forward-secure signature schemes [13], where the secret key is updated from time to time. Since the public key remains the same, freshness cannot be ensured, i.e. a user is not able to distinguish the fresh data chunk from a stale version thereof.

## 2 Chameleon Authentication Trees

Our formal definition of CATs differs slightly from [22], since we directly model updates as a property of the CATs. This allows us to build VDS protocols in a black-box way from CATs, while [22] needed to make specific *non*black-box assumptions about the proof that might not hold in general. The second difference is that we do not put an upper bound on the number of leaves. Thus, the only input of $\mathsf{catGen}$ is the security parameter. The formal definition of VDS itself is deferred to Appendix A.

**Definition 1.** *A* chameleon authentication tree *is a tuple of efficient algorithms* $\Pi_{\mathsf{CAT}} = (\mathsf{catGen}, \mathsf{catAdd}, \mathsf{catUpdate}, \mathsf{catVerify})$*, which are defined as follows:*

$\mathsf{catGen}(1^\lambda)$*: The key generation algorithm takes the security parameter* $\lambda$ *and outputs a key pair* $(\mathsf{vp}, \mathsf{sp})$*. For simplicity we always assume that* $\mathsf{vp}$ *is contained in* $\mathsf{sp}$*.*

$\mathsf{catAdd}(\mathsf{sp}, \ell)$*: The insertion algorithm takes a secret key* $\mathsf{sp}$*, and a datum* $\ell$ *from some data space* $\mathcal{L}$*. It outputs a new secret key* $\mathsf{sp}'$*, a position* $i$ *at which* $\ell$ *was inserted and a proof* $\pi_i$*, which is a publicly verifable proof showing that* $\ell$ *is indeed stored at position* $i$*.*

$\mathsf{catUpdate}(\mathsf{sp}, i, \ell)$*: The update algorithm takes the secret key* $\mathsf{sp}$*, a position* $i$ *at which we want to perform the update, and the new datum* $\ell \in \mathcal{L}$ *as input. It replaces the current datum at* $i$ *with* $\ell$ *and outputs a new key pair* $(\mathsf{vp}', \mathsf{sp}')$ *as well as a proof* $\pi_i$ *for the new datum.*

$\mathsf{catVerify}(\mathsf{vp}, i, \ell, \pi_i)$*: The verification algorithm takes the public key* $\mathsf{vp}$*, a position* $i$*, a datum* $\ell$ *and a proof* $\pi_i$ *as input and outputs 1 iff* $\ell$ *is stored at position* $i$*. It outputs 0 otherwise.*

Security of CATs: Our security definition deviates from the one given in [22], by taking update queries of the adversary into account. We present a single definition that covers both, structure-preservation and one-wayness. Intuitively, we say that a CAT is *secure* if no efficient adversary can modify the tree by changing the sequence of the data stored in it, substituting any datum, or by adding further data to it. In particular, the definition also prevents the adversary from returning stale to clients. The game is defined as follows:

Setup: The challenger generates a key-pair $(\mathsf{sp}, \mathsf{vp}) \leftarrow \mathsf{catGen}(1^\lambda)$ and hands $\mathsf{vp}$ over to the adversary $\mathcal{A}$.

Uploading: Proceeding adaptively, the attacker $\mathcal{A}$ uploads a datum $\ell \in \mathcal{L}$ to the challenger. The challenger adds $\ell$ to the database, computes $(\mathsf{sp}', i, \hat{\pi}) \leftarrow \mathsf{catAdd}(\mathsf{sp}, \ell)$, and returns $(i, \hat{\pi})$ to $\mathcal{A}$. Alternatively, the adversary may update any element in the outsourced database by sending an index $i$, a datum $\ell'$ to the challenger. The challenger then runs the update algorithm with $\mathcal{A}$ updating $\ell_i$ to $\ell'$. At the end of the update protocol the challenger returns the updated proof $\pi_i'$ and the updated public-key $\mathsf{vp}'$ to $\mathcal{A}$. Denote by $Q := \{(\ell_1, 1, \hat{\pi}_1), \ldots, (\ell_{q(\lambda)}, q(\lambda), \hat{\pi}_{q(\lambda)})\}$ the ordered sequence of the latest versions of all uploaded elements and let $\mathsf{vp}^*$ be the corresponding public key.

Output: Eventually, $\mathcal{A}$ outputs $(\ell^*, i^*, \hat{\pi}^*)$. The attacker $\mathcal{A}$ is said to win the game if one of the following two conditions is true:

a) If $1 \leq i^* \leq q(\lambda)$ and $(\ell^*, i^*, \hat{\pi}^*) \notin Q$ and $\mathsf{catVerify}(\mathsf{vp}^*, i^*, \ell^*, \hat{\pi}^*) = 1$.

b) If $i^* > q(\lambda)$ and $\mathsf{catVerify}(\mathsf{vp}^*, i^*, \ell^*, \hat{\pi}^*) = 1$.

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

**Definition 2.** *A chameleon authentication tree $\Pi_{\mathsf{CAT}} = (\mathsf{catGen}, \mathsf{catAdd}, \mathsf{catUpdate}, \mathsf{catVerify})$ is* secure *if for any $q \in \mathbb{N}$, and for any efficient algorithm $\mathcal{A}$, the probability $\mathbf{Adv}_{\mathcal{A}}^{\mathsf{sec}}$ evaluates to 1 is negligible (as a function of $\lambda$).*

## 3 Constructing Fully Dynamic CATs

We now present our fully dynamic CAT construction, which is the first construction that is able to authenticate an unbounded number of data elements and improves upon the state-of-the-art in terms of computational and bandwidth overhead. In the following we first recall the definition of chameleon hash functions and then present our construction.

### 3.1 Chameleon Hash Functions

A chameleon hash function is a randomized hash function that is collision-resistant but provides a trapdoor to efficiently compute collisions. It is defined through the tuple $\mathcal{CH} = (\mathsf{chGen}, \mathsf{ch}, \mathsf{col})$ [12], where the key generation algorithm $\mathsf{chGen}(1^\lambda)$ returns a key pair $(csk, cpk)$. We set $\mathsf{ch}(\cdot) := \mathsf{ch}(cpk, \cdot)$ for the remainder of this paper. The function $\mathsf{ch}(x; r)$ produces a hash value $h \in \{0,1\}^{\mathrm{out}}$ for a message $m \in \{0,1\}^{\mathrm{in}}$ and a randomness $r \in \{0,1\}^\lambda$. The function is collision-resistant meaning that given $cpk$ it is computationally difficult to compute a tuple $(m, r), (m', r')$ such that $(m, r) \neq (m', r')$ and $\mathsf{ch}(m, r) = \mathsf{ch}(m', r')$. However, using the trapdoor $csk$ and the collision-finding algorithm $\mathsf{col}(csk, x, r, x')$ we can break the collision-resistance property and find a value $r'$ such that both, $(x, r)$ and $(x', r')$ map to the same hash value.

We call $\mathcal{CH}$ invertible if it is surjective and there exists an efficient algorithm $\mathsf{scol}(csk, x, y)$ that outputs an $r$ for any input $x$ and $y$ such that $y = \mathsf{ch}(x; r)$. This property has previously been defined by Shamir and Tauman [25].

Chameleon hash functions can be instantiated from the discrete-logarithm assumption [12,2], the factoring assumption [25], the RSA assumption [2,11], or in a generic way from certain $\Sigma$-protocols [3].

### 3.2 Intuition

The main idea of our construction is to build a binary tree, which stores the data elements in its leaves and grows dynamically from bottom up. Whenever the tree of a certain depth $d$ is full, the data owner can increase the tree's depth by one using his secret trapdoor. The resulting tree is of depth $d+1$ and can therefore store another $2^d$ elements. The previous full tree becomes the left child under the new root and the right child serves as a place holder for an empty subtree, which can be used to store new data.

This new approach of dynamically increasing the depth of the tree means that we cannot simply store the root node's value in the public key, since it changes whenever the depth is extended. Instead, our idea is to define the new root through a deterministic function that is applied to a fixed value in the public key and depends on the current depth of the tree. More precisely, let $\rho$ be the value in the public key $pk$ and assume that the depth of the tree is $d$. Then, the root node is defined as $H^d(\rho)$, where $H$ is a collision-resistant hash function and by $H^d(\rho)$ we denote the $d$-fold application of $H$ to $\rho$.

Our binary tree is structured as follows: Each node value is computed as the output of a function of the concatenated values of its children. For nodes which are left children themselves we use a collision-resistant hash function. For right children we use a chameleon hash function. The only exception to this rule are all nodes, that have been a root at some point, i.e., all nodes at the very left of each level. We insert the data elements into the trees starting from the leftmost leaf moving to the right. Whenever we insert some data element into a leaf we have to ensure that, roughly speaking, the root node's value computed from that leaf remains the same as before the insertion. Therefore, we search for a node computed by a chameleon hash function on the path from the inserted leaf to the root and compute an appropriate collision using our secret trapdoor.

In the following, we exemplify basic idea of our construction with a small example, where we will refer to a node $v$ at height $h$ and index $i$ by $v_{h,i}$. Please note, even though we will be including two leaves at the same time, this should not be seen as a restriction or a problem. It is done for the sake of clarity and the construction can be easily extended to insert one leaf at a time, as it is done in our framework.

SETUP: We compute $(cpk, csk) \leftarrow \mathsf{chGen}(1^\lambda)$ using the key generation algorithm of the chameleon hash function. The setup algorithm stores the trapdoor $csk$ of the chameleon hash function in the private key $sk$; the corresponding public verification key $pk$ contains $cpk$ and a randomly chosen value $\rho$. At the beginning of the streaming protocol, the tree is empty.

APPENDING THE ELEMENTS $\ell_1, \ell_2$: In the first step, we append the elements $\ell_1$ and $\ell_2$ to the tree. Since the tree is empty, i.e. it has depth 0, it is necessary to increase its depth by one. In order to add $\ell_1$ and $\ell_2$ to the tree without changing the root, the data owner uses his secret trapdoor to compute $r_{1,0} \leftarrow \mathsf{scol}(csk, \ell_1 \| \ell_2, H^1(\rho))$. Hence $\mathsf{ch}(\ell_1 \| \ell_2; r_{1,0}) = H(\rho)$. Recall that $\mathsf{scol}$ outputs some randomness $r$ when given $(y, x)$ and the secret key $csk$ such that $y = \mathsf{ch}(x; r)$. At this stage the entire tree consists only of two leaves and one root node as depicted in Figure 2 (level 1). To verify that the leaves $(\ell_1, \ell_2)$ are in the tree, the verification algorithm checks whether $H^1(\rho) = \mathsf{ch}(\ell_1 \| \ell_2; r_{1,0})$ holds.

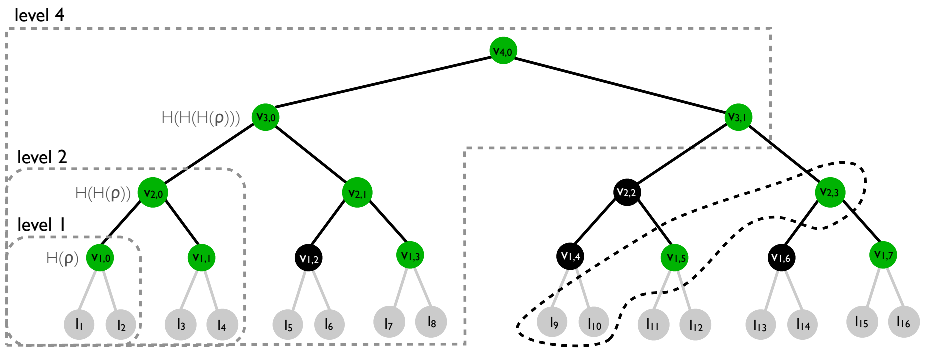


**Figure 2.** The fully dynamic CAT. Green nodes are computed using the chameleon and black nodes using the collision-resistant hash functions. The tree stores $2^i$ elements at level $i$.

<u>Appending the elements $\ell_3, \ell_4$:</u> Next, we add $\ell_3$ and $\ell_4$ to the tree. Since the current tree is full, we need to extend its height to obtain new free leaf positions. Therefore, we pick a random $x_{1,1}$ and $r_{1,1}$ and we compute the dummy node $v_{1,1} \leftarrow \mathsf{ch}(x_{1,1}; r_{1,1})$. The randomly chosen pre-images are stored by the client in his secret local state. To ensure the integrity of the tree, we need to find a randomness $r_{2,0}$ for the new root $v_{2,0}$ such that $\mathsf{ch}(H^1(\rho)\|v_{1,1}; r_{2,0}) = H^2(\rho)$. Again, this is achieved by exploiting the inversion property of the chameleon hash function to compute $r_{2,0} \leftarrow \mathsf{scol}(csk, H^1(\rho)\|v_{1,1}, H^2(\rho))$. We can now add our leaves $\ell_3$ and $\ell_4$ to the tree by appending them to the lowest free right child, which is $v_{1,1}$. Thus, we compute $r'_{1,1} \leftarrow \mathsf{col}(csk, x_{1,1}, r_{1,1}, \ell_3\|\ell_4)$. The resulting proof for $\ell_3, \ell_4$ would therefore contain $(v_{1,0}, r_{2,0}, r_{1,1})$. The corresponding tree is shown in Figure 2 (level 2).

<u>Appending the elements $\ell_5, \ell_6$ and $\ell_7, \ell_8$:</u> Since the tree is full again, we need to increase its depth the same way we did before. Afterwards, we search for the lowest right child, which does not have any children yet. In this case the node is $v_{2,1}$ that has been computed by $\mathsf{ch}(x_{2,1}; r_{2,1})$. The dummy values $(x_{2,1}, r_{2,1})$ can be retrieved from the local client state. In order to append $\ell_5$ and $\ell_6$, we generate an empty subtree below $v_{2,1}$. This subtree consists of a dummy node $v_{1,3}$ and the leaves $\ell_5$ and $\ell_6$. After appending $\ell_5$ and $\ell_6$ below $v_{1,2}$, we compute $r'_{2,1} \leftarrow \mathsf{col}(csk, x_{2,1}, r_{2,1}, v_{1,2}\|v_{1,3})$. The proof for these elements contains $(r'_{2,1}, v_{1,3}, v_{2,0})$. Next, $\ell_7$ and $\ell_8$ can authenticated by appending them to $v_{1,3}$ and computing a collision in the same fashion as in the previous steps.

<u>Verification:</u> The verification algorithm works analogously to the one of a Merkle tree. One might get the impression that the size of the proofs grows with the number of leaves *for all leaves*. This, however, is not the case. For instance, the node $v_{1,0}$ verifies the leaves $\ell_1$ and $\ell_2$ even if $2^{50}$ elements are stored in the tree. The verification algorithm still simply checks whether $H(\rho) = \mathsf{ch}(\ell_0\|\ell_1; r_{1,0})$.

Updating the Tree: Whenever we wish to update the $i$-th element in the database to some element $\ell'_i$, we simply replace the element, recompute the values on the path from $\ell'_i$ to the corresponding root node, pick a fresh value $\rho'$, and update all sub-roots w.r.t. $\rho'$. Updating the sub-roots means that the client has to compute logarithmically many collisions.

Constant State: In the intuitive description of our construction, the client's state is logarithmic in the depth of the tree, since all created dummy nodes $y \leftarrow \mathsf{ch}(x; r)$ are stored by the client. To reduce the client's state to $\mathcal{O}(1)$ we use a pseudorandom function PRF to compute the dummy elements on the fly. That is, for each dummy node $v_{h,i}$, the clients computes the pair $(x_{h,i}, r_{h,i}) \leftarrow \mathsf{PRF}(k, h \| i)$, rather than choosing it randomly. This allows us to recompute the dummy nodes we need on-the-fly without storing them.

The secret seed of the PRF is stored as part of the secret key. Therefore, the final secret key in our construction consists of the trapdoor $csk$ of the chameleon hash function, the seed $k$ of the PRF, and a counter $c$ that keeps track of the next free leaf index. In practice and in our framework, one can instantiate the PRF using a symmetric encryption scheme, such as AES.

## 3.3 Formal Construction

We now provide a detailed description of all algorithms, that have been sketched in the previous section. We avoid using the PRF in this description for the sake of clarity, but the modification is absolutely straightforward as described above.

**Construction 1** *Let $H : \{0,1\}^* \mapsto \{0,1\}^{len}$ be a hash function and $\mathcal{CH} = (\mathsf{chGen}, \mathsf{ch}, \mathsf{col}, \mathsf{scol})$ an invertible chameleon hash function that maps strings of length $\{0,1\}^*$ to $\{0,1\}^{len}$. The fully-dynamic chameleon authentication tree $\Pi_{\mathsf{CAT}} = (\mathsf{catGen}, \mathsf{catAdd}, \mathsf{catUpdate}, \mathsf{catVerify})$ consists of the following efficient algorithms:*

$\mathsf{catGen}(1^\lambda)$: *The setup algorithm generates a key-pair of the chameleon hash $(cpk, csk) \leftarrow \mathsf{chGen}(1^\lambda)$, it picks a uniformly random value $\rho \leftarrow \{0,1\}^\lambda$, and denote by $\mathsf{st}$ the private state. This state stores the next free leaf index $c$, a set of pre-images of unused dummy nodes and the last computed proof. Initially we set $c \leftarrow 0$, while the set of pre-images and the last computed proof are both empty. It returns the public verification key $\mathsf{vp} = (cpk, \rho)$ and the private key $\mathsf{sp} = (csk, \mathsf{st}, \mathsf{vp})$.*

$\mathsf{catAdd}(\mathsf{sp}, \ell)$: *Parse $\mathsf{sp}$ as $(csk, \mathsf{st}, \mathsf{vp})$ and check whether the current tree is full, i.e., whether $c$ is a power of two:*

The counter $c$ is a power of two: *In this case the current tree is full, we need to increase its current depth by one to obtain a tree of depth $d$, which has free leaves again. To do so, we store the old root node $H^{d-1}(\rho)$ as the left child of the new root node $H^d(\rho)$ and we create a dummy node $v_{d-1,1}$ for the* *right child as follows: First, we pick the values $x_{d-1,1}$ and $r_{d-1,1}$ uniformly at random and we compute the dummy node $v_{d-1,1} \leftarrow \mathsf{ch}(x_{d-1,1}; r_{d-1,1})$. Second, we exploit the inversion property of the chameleon hash function in order to compute $r_{d,0} \leftarrow \mathsf{scol}(csk, H^{d-1}(\rho)\|v_{d-1,1}, H^{d}(\rho))$. Next, we add $(x_{d-1,1}, r_{d-1,1})$ to the set of pre-images and $v_{d-1,1}$ to the proof in* st *and proceed as in the case where $c$ is not a power of two.*

<u>The counter $c$ is not a power of two:</u> *Since the tree is not full, we search for the lowest right child $v_{i,j}$, which has no children, in the proof that was stored in* st *during the last run of* catAdd*. Then, we generate a skeleton subtree below $v_{i,j}$ the following way. First, we descend from $v_{i,j}$ along the left edge until we reach height 1. Then we create one adjacent dummy node at each height, i.e., we create dummy nodes at $v_{i-k,2^k\cdot j+1}$ for $k = 1 \dots i-1$. The pre-image of each created dummy node is added to* st*. Now, we append the given leaf $\ell$ as the left most child to the newly generated subtree at height $0$. Given the leaf and the dummy nodes, the value of $v_{i,j}$ can now be determined recursively by computing $v_{i,j} \leftarrow v_{i-1,2\cdot j}\|v_{i-1,2\cdot j+1}$. We re-establish the tree's integrity by computing a randomness $r'_{i,j} \leftarrow \mathsf{col}(csk, x_{i,j}, r_{i,j}, v_{i,j})$. We create a proof $\pi$ for $\ell$, which contains all newly created dummy nodes, $r'_{i,j}$, the node adjacent to $v_{i,j}$ and all nodes from the old proof, which were above $v_{i,j}$. Finally, we increase the next free leaf index $c$ in the client state by one, replace the proof in* st *with the newly generated one, and return it.*

<u>$\mathsf{catVerify}(\mathsf{vp}, i, \ell, \pi)$:</u> *Parse* vp *as $(cpk, \rho)$. In order to verify, whether $\pi$ authenticates $\ell$ we compute, starting from the bottom, each node as the hash or the chameleon hash of the concatenation of its two children until we compute a node with index 0. All nodes and randomnesses that are needed are taken from the given $\pi$. In case the node we want to compute has a odd index, we use the chameleon hash function. Otherwise we use the hash function. Let $v_{d,0}$ be the node at which we terminated. We return 1 iff $v_{d,0} = H^{d}(\rho)$, and $0$ otherwise.*

<u>$\mathsf{catUpdate}(\mathsf{sp}, i, \ell')$:</u> *Parse* sp *as $(csk, \mathsf{st}, \mathsf{vp})$ and* vp *as $(cpk, \rho)$. Request $\ell_i$, with its proof of correctness $\pi_i$. Request $\ell_0$ with its proof of correctness $\pi_0$. Compute $\mathsf{catVerify}(\mathsf{vp}, i, \ell_i, \pi_i)$ and $\mathsf{catVerify}(\mathsf{vp}, 0, \ell_0, \pi_0)$ and abort if one of them outputs 0. Let $\pi = \pi_i \cup \pi_0$ denote the total set of nodes and randomnesses obtained by the client at this point. Replace $\ell_i$ with $\ell'$ and recompute all values that are on the path from $\ell'$ to the root recessively. Pick a new $\rho' \leftarrow \{0,1\}^{\lambda}$ and replace $\rho$ with $\rho'$ in* vp*. This means that at each height $h$ we now have to ensure again that $H^{h}(\rho') = \mathsf{ch}(v_{h-1,0}\|v_{h-1,1})$. Therefore, we compute $r'_{h,0} \leftarrow \mathsf{scol}(csk, v_{h-1,0}\|v_{h-1,1}, H^{h}(\rho'))$ at each height $h$ and add all newly computed $r'_{h,0}$ to $\pi$ and return $\pi$.*

**Theorem 1.** *If $\mathcal{CH}$ is an invertible one-way collision-resistant chameleon hash function and $H$ is a collision-resistant hash function modeled as a random oracle, then Construction 1 is a secure unbounded verifiable data streaming protocol.*

Due to space constraints the security proof is deferred to Section B.

## 4 Implementation

VeriStream is written in Java and it contains all protocols described in this paper as well as a separate library for chameleon hash functions, which contains implementations of the Krawczyk-Rabin [12], the Ateniese and de Medeiros chameleon hash [2], and its elliptic curve equivalent. For the elliptic curve operations we used the Bouncy Castle Cryptographic API (Release 1.49) [1]. In addition we provide a generic interface for transforming $\Sigma$-protocols, that fulfil certain properties into chameleon hash functions [3]. Using this interface we instantiated a chameleon hash function from the Fiat-Shamir protocol [3]. Many chameleon hash functions only take input from certain message spaces, e.g., Krawczyk-Rabin expects messages from $\mathbb{Z}_q^*$. We provide a simple wrapper that transforms them into functions that take arbitrary large inputs, by first hashing the input with a common collision-resistant hash function, like SHA-256, before passing it to the chameleon hash function. The remaining algorithms of the chameleon hash functions are adapted accordingly by the wrapper.

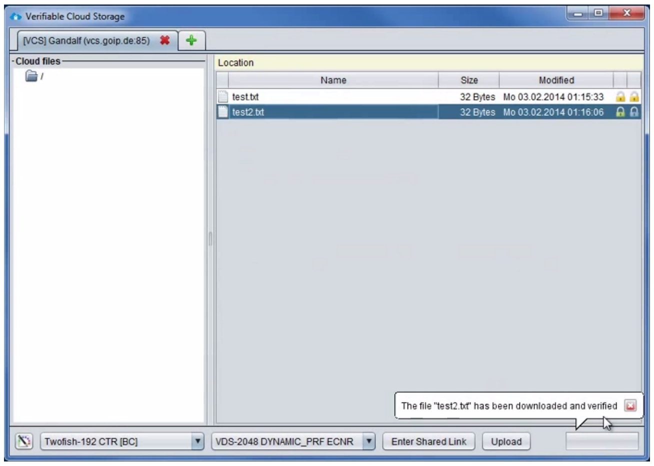
We developed a platform independent standalone client that uses VeriStream (see Figure 3). It allows its users to manage, upload, download, or share files of an arbitrary format in an authenticated fashion. Users can choose whether they want to upload their data to a private web storage or whether they want to use their Dropbox account as the underlying storage layer. The client is able to stream audio and video content with on-the-fly-verification, even if Dropbox is the underlying storage layer.

For developers, VeriStream offers a simple to use interface by the means of so called VDS handlers. This handler is parameterized by the VDS protocol type and chameleon hash function that shall be deployed. It offers methods for creating, verifying, updating, and obtaining proofs from CATs. Since network bandwidth is an important issue, we transform the proofs into compact byte sequences representations before sending them over the network, rather than relying on bloated formats like JSON. We implemented a server, based on a common thread-pool architecture, that receives from and streams data to clients, where both parties deploy a VDS handler to secure the transmitted content.

**Efficiency Optimizations.** For performance and bandwidth reasons, our framework differs from the theoretical description in several points that we discuss in the following.

SHORT PROOFS: When uploading data to the server it is not necessary to always send the whole proof. Instead, it is sufficient to only send all nodes that are below the chameleon hash node that was extended. One can think of this optimization as transmitting only the delta between the previous proof(s) and the current one.

PARALLELIZING THE CAT: Recall that the insertion algorithm always generates a certain amount of chameleon dummy nodes by picking random pre-images and



| Name | Size | Modified |
|---|---|---|
| test.txt | 32 Bytes | Mo 03.02.2014 01:15:33 |
| test2.txt | 32 Bytes | Mo 03.02.2014 01:16:06 |

The file "test2.txt" has been downloaded and verified

Twofish-192 CTR [BC]

VDS-2048 DYNAMIC_PRF ECNR

**Figure 3.** Screenshot of the VeriStream standalone client.

storing them in the client state. Later on, when we insert elements below one such dummy node, we use the pre-images from the state to compute a collision in that dummy accordingly. Now, instead of picking these pre-images completely at random, we provide the possibility to pick them using a pseudorandom function, which takes the dummy nodes position as input. This way, we reduce the client state to constant size, since we do not need to store the pre-images anymore. Furthermore, being able to compute the values of dummy nodes independent of the actual existing tree allows us to obtain concurrent versions of all protocols. The position of an element in the data stream while uploading uniquely defines its position in the CAT. Having the element, and using the pseudorandom function to obtain the dummy value to which that element will be appended, we can compute the short proof independent of the remaining tree. We believe that it is not straightforward to see that the parallelization of the CAT indeed works, because almost half of the nodes are computed using a collision-resistant hash functions and these nodes cannot be pre-computed without knowing the pre-images. However, a closer look at our construction shows that all these values belong to the left part of the tree and these elements have all been pre-computed before.

Continuous Requests: Depending on the concrete scenario, a single or multiple elements in succession can be requested. In the case, where multiple adjacent elements are requested, we exploit the following observation: Given the proof $\pi_i$ for some leaf $i$ and the proof $\pi_{i+1}$ for its successor $i+1$, all nodes in $\pi_{i+1}$, that are above the node which was extended when inserting the leaf $i+1$ are also contained in $\pi_i$. Hence when a set of adjacent elements is requested, we send the full proof for the first and short proofs for the remaining elements.

## 5 Evaluation

In this section, we provide a comprehensive efficiency analysis of VeriStream. In this analysis, the chunk size is an important variable, because we compute one proof for each chunk and we therefore test our implementation with differ-

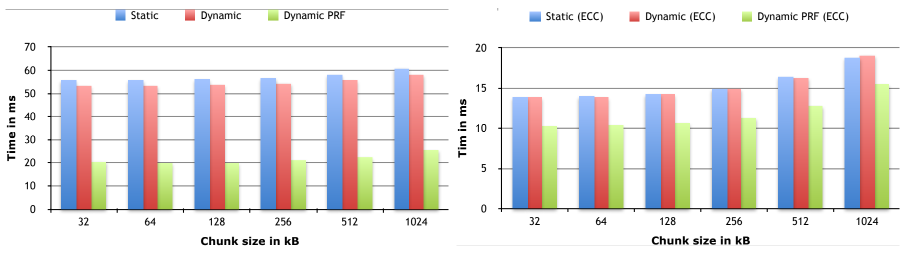


**Figure 4.** Average time for authenticating one data chunk. On the left all computations were performed on cyclic groups. On the right on elliptic curves.

ent chunk sized to obtain detailed insights into the protocols performance. In addition, we conduct several different benchmarks highlighting all possible operations for all discussed protocols. Our experimental analysis was conducted on a Intel Core i3-2120 CPU with 8 GB of RAM running Ubuntu 12.04 with Java 1.6.0.

Evaluation of our Framework: We analyzed the performances of all protocols by uploading and streaming 2GB of data with different chunk sizes, such as 32kB, 64kB, . . ., 1024kB. Smaller chunk sizes result in more chunks and therefore bigger CATs. In addition, we were interested in the performance impact of utilizing a pseudorandom function for computing the dummy nodes and therefore we conducted experiments with the fully dynamic CAT that used a pseudorandom function. As the underlying chameleon hash function we used the scheme due to Ateniese and de Medeiros. For a performance comparison of chameleon hash functions see Section D. As the underlying group we used both elliptic curves and regular cyclic groups with a security parameter of 112 bits. The CAT from [22] was initialized with a depth of 30, which results in a tree that can authenticate $2^{30}$ data chunks. In the following we will refer to their construction in the figures as *static*. To obtain meaningful and detailed performance results, we computed the averages of the following measurements:

– Time for hashing a chunk and authenticating it.
– Time for obtaining and verifying a proof from the CAT.
– Bandwidth overhead produced by a proof retrieved from the CAT.

Evaluation Results: When authenticating and uploading data chunks to the server, the data owner only sends short proofs to the server as described in Section 4. A comparison of the computational overhead incurred by this authentication step in the different constructions is depicted in Figure 4. One can see that our constructions outperform the construction from [22]. In particular, this is interesting, since our fully dynamic CAT also provides a better functionality, i.e. it allows uploading and streaming an unbounded amount of data. One somewhat surprising result is, that combining our construction with a pseudorandom function not only reduces the size of the client's state, but also significantly increases its performance. In practice, the computation of the client's state after each uploaded chunk is far more expensive than the evaluation of the pseudorandom function. All protocols perform better by more than a factor of two, when using the elliptic curve chameleon hash function.

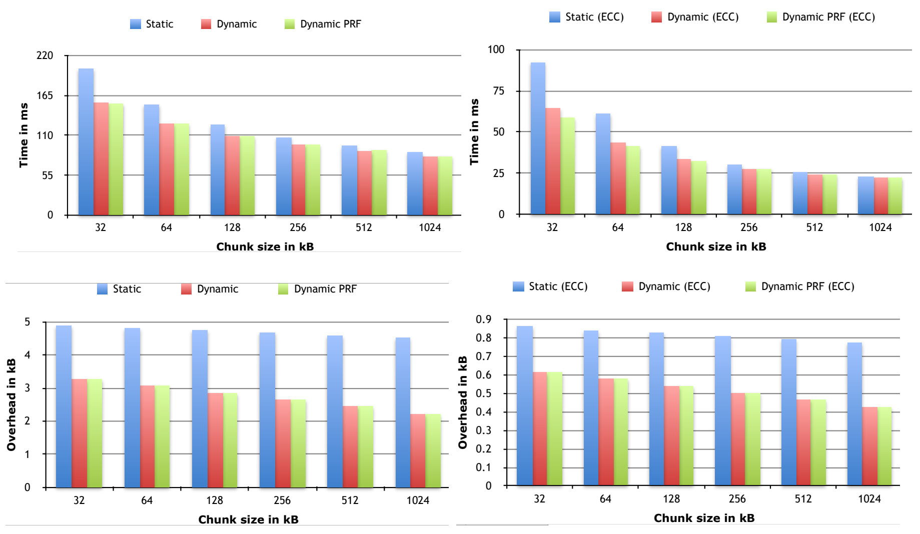


**Figure 5.** The two plots at the top depict the average full proof computation and verification time per data chunk. The two plots at the bottom show the average bandwidth overhead for one data chunk.

In the next step we analyzed the computational and bandwidth overhead of the retrieval operation. We measured the time it took to compute the proof from the CAT upon a client request for a certain element and verify the returned proof. More precisely we created a CAT that contained proofs for a 2GB large data set and requested all chunks from it, such that each proof in the CAT had to be computed once. For each received proof the verification algorithm was executed once. We stress that we did not use the efficent method for retrieving sequential parts of the uploaded data, but purposely requested each chunk on its own with its full proof. The results of this experiment can be seen in Figure 5. At the top one can see the average time it took to obtain a proof from the CAT and verify it. At the bottom one can see the average size of such a retrieved proof. The protocol from [22] performs worst w.r.t. to computational and bandwidth overhead, what confirms our expectation, since, in contrast to the dynamically growing trees, all elements in their construction verify against the very top level root value. This requires, on average, much more bandwidth and more computational power. Our two constructions perform roughly equally well as expected. Using the elliptic curve variant of the Ateniese and de Medeiros chameleon hash results in a improvement of roughly factor 5 with regards to the size of the proofs and a speed up of about factor 3.

## Acknowledgements

Dominique Schröder and Mark Simkin were supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy, and Accountability (CISPA; see `www.cispa-security.org`). Dominique Schröder is also supported by an Intel Early Career Faculty Honor Program Award.

## References

1. Bouncy Castle Crypto APIs. 4
2. 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. 3.1, 4, C
3. Mihir Bellare and Todor Ristov. Hash functions from sigma protocols and improvements to VSH. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *Lecture Notes in Computer Science*, pages 125–142, Melbourne, Australia, December 7–11, 2008. Springer, Berlin, Germany. 3.1, 4, C
4. 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. 1.3
5. 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. 1.3
6. 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. 1.3
7. David Cash, Alptekin Küpçü, and Daniel Wichs. 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, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 279–295. Springer, 2013. 1.3
8. Dario Catalano and Dario Fiore. Vector commitments and their applications. In Kaoru Kurosawa and Goichiro Hanaoka, editors, *PKC 2013: 16th International Workshop on Theory and Practice in Public Key Cryptography*, volume 7778 of *Lecture Notes in Computer Science*, pages 55–72, Nara, Japan, February 26 – March 1, 2013. Springer, Berlin, Germany. 1.3
9. C. Christopher Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In Ehab Al-Shaer, Somesh Jha, and Angelos D. Keromytis, editors, *ACM CCS 09: 16th Conference on Computer and Communications Security*, pages 213–222, Chicago, Illinois, USA, November 9–13, 2009. ACM Press. 1.3
10. 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/`. 1.3
11. Susan Hohenberger and Brent Waters. Realizing hash-and-sign signatures under standard assumptions. In Antoine Joux, editor, *Advances in Cryptology – EUROCRYPT 2009*, volume 5479 of *Lecture Notes in Computer Science*, pages 333–350, Cologne, Germany, April 26–30, 2009. Springer, Berlin, Germany. 3.1, B, B

12. 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. 3.1, 4
13. Tal Malkin, Daniele Micciancio, and Sara K. Miner. Efficient generic forward-secure signatures with an unbounded number of time periods. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 400–417, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Berlin, Germany. 1.3
14. 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. 1.3
15. Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on Selected Areas in Communications*, 18(4):561–570, 2000. 1.3
16. 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. 1.3
17. National Institute of Standards and Technology. Recommendation for key management. Special Publication 800-57 Part 1 Rev. 3, NIST, 2012. `http://www.keylength.com/`. D
18. Charalampos Papamanthou, Elaine Shi, Roberto Tamassia, and Ke Yi. Streaming authenticated data structures. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 2013. 1.3
19. 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. 1.3
20. Adrian Perrig, Ran Canetti, Dawn Xiaodong Song, and J. Doug Tygar. Efficient and secure source authentication for multicast. In *ISOC Network and Distributed System Security Symposium – NDSS 2001*, pages 35–46, San Diego, California, USA, February 7–9, 2001. The Internet Society. 1.3
21. Adrian Perrig, Ran Canetti, J. Doug Tygar, and Dawn Xiaodong Song. Efficient authentication and signing of multicast streams over lossy channels. In *2000 IEEE Symposium on Security and Privacy*, pages 56–73, Oakland, California, USA, May 2000. IEEE Computer Society Press. 1.3
22. Dominique Schröder and Heike Schröder. Verifiable data streaming. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 12: 19th Conference on Computer and Communications Security*, pages 953–964, Raleigh, NC, USA, October 16–18, 2012. ACM Press. 1.1, 1, 1.1, 1.3, 2, 2, 5, 5, A, A, B, C
23. 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. 1.3
24. 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. 1.3
25. Adi Shamir and Yael Tauman. Improved online/offline signature schemes. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture*

*Notes in Computer Science*, pages 355–367, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Berlin, Germany. 3.1
26. Roberto Tamassia and Nikos Triandopoulos. Certification and authentication of data structures. In *AMW*, 2010. 1.3

## A Verifiable Data Streaming

In the following we shortly recall the formal definitions of VDS protocols and CATs. Our definition of VDS is taken almost verbatim from [22]:

**Definition 3.** *A* verifiable data streaming *protocol* $\mathcal{VDS} = (\mathsf{Setup}, \mathsf{Append}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Update})$ *is a protocol between two efficient algorithms: a client* $\mathcal{C}$ *and a server* $\mathcal{S}$*. The server can store an exponential number* $N$ *of elements in its database* $DB$ *and the client keeps some small state, which shall not be larger than* $\mathcal{O}(\log N)$*. The scheme consists of the following efficient algorithms:*

$\mathsf{Setup}(1^\lambda)$*: The setup algorithm outputs a verification key* $pk$ *and a secret key* $sk$*, where* $pk$ *is given to the server* $\mathcal{S}$ *and* $sk$ *to the client* $\mathcal{C}$*. W.l.o.g.,* $sk$ *always contains* $pk$*.*

$\mathsf{Append}(sk, s)$*: This algorithm appends the value* $s$ *to the database* $DB$ *held 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* $pk$*.*

$\mathsf{Query}(pk, DB, i)$*: The interactive query protocol is defined as* $\langle \mathcal{S}(pk, DB), \mathcal{C}(i)\rangle$ *and is executed between the server* $\mathcal{S}(pk, DB)$ *and client* $\mathcal{C}(i)$*. 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* $\perp$*.*

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

$\mathsf{Update}(pk, DB, sk, i, s')$*: The interactive update protocol, denoted by* $\langle \mathcal{S}(pk, DB), \mathcal{C}(sk, i, s')\rangle$*, 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* $pk$ *is updated to* $pk'$*.*

<u>Security of VDS:</u> Loosely speaking, an adversary $\mathcal{A}$ against a VDS protocol tries to tamper with the uploaded data set by either adding an element to it, changing the order of two elements, or deleting an element from the authenticated data set. Since our modified security definition of CATs given in Section 2 directly implies the security definition for VDS from [22], we omit a formal security definition of VDS here.

## B Security Proof for the Fully Dynamic CAT

Before proving the security of our construction, we describe the main proof idea. Recall that our goal is to rule out any adversary that can creates a *valid* proof for an element that was *not* at a certain position in the stream. Intuitively, it seems obvious that there must be either a collision in the hash function or in the chameleon hash function. The problem, however, occurs in the reduction to the chameleon hash function. The reasons are twofold: First, in the reduction against the collision-resistance of the chameleon hash function does not have access to the trapdoor, thus, computing a colliding randomness while the adversary adds elements to the tree is not possible. The second issue stems from the fact, that once the attacker receives the public key, it learns the initial seed $\rho$. Thus, programming the random oracle to map to the output of the chameleon hash does also not work because the attacker can simply compute the $n$-fold application of $H$ to $\rho$ before sending a single element to the server.

To handle these issue, we split the proof in three parts. First, we first prove a weaker result in which the adversary has to commit to all values in the stream before seeing the public key of the CAT. Moreover, the attacker cannot perform any update query. The attacker then receives the public key, the corresponding proofs of correctness, and its task is to output a false statement.The main observation is that programming the random oracle is now indeed possible, because the adversary obtains the public key after outputting all elements.

Afterwards, we allow the adversary to choose the elements in the stream adaptively, but we still do not allow to perform any update queries. The main idea here is to achieve adaptivity by storing chameleon hash values in the leaves. Thus, whenever the attacker asks to add an element, we use the trapdoor of the chameleon hash to find a colluding randomness. This technique is well known and has previously been used in, e.g., [11,22].

In the last step of the proof we allow the attacker to perform updates. The main issue in simulating update queries is again that we have to find logarithmically many collisions, without having access to the trapdoor of the chameleon hash. The key idea to overcome this technical challenge is to exploit the programmability of the random oracle again. In contrast to the first step of the proof, this is indeed possible because we pick a fresh value $\rho'$ that will be stored in the public key. Thus, for a tree of depth $D$, we check that all values $H(\rho'), \ldots, H^d(\rho')$ have not been queried by the adversary. If this is the case then, we can program these values accordingly, by setting $H(\rho') := \mathsf{ch}(\ell_1, \|\ell_2; r'), \ldots$ for a freshly chosen randomness $r'$. Observe that $\mathsf{ch}(\ell_1, \|\ell_2; r')$ is uniformly distributed, because $r'$ is chosen at random. Thus, the distribution generated during the simulation is identical to the one in our construction. Furthermore, with all but negligible probability $\mathsf{ch}(\ell_1, \|\ell_2; r')$ maps to a fresh value. Otherwise, we could easily build a reduction against the collision-resistance of $\mathsf{ch}$.

**Proposition 1.** *If $\mathcal{CH}$ is an invertible one-way collision-resistant chameleon hash function and $H$ a collision-resistant hash function modeled as a random oracle, then Construction 1 is a secure fully dynamic chameleon authentication tree (according to Definition 2).*

In the first step of the proof we only consider *non*-adaptive adversaries that output all elements that should be added to the CAT before seeing the public-key. In addition, this class of adversaries cannot update any element. We distinguish between two different cases. Either, the attacker manages to change some element in the stream, or the adversary appended an element to it. We prove both cases with the following two claims.

*Claim.* If $\mathcal{A}$ is a non-adaptive adversary that outputs $(\ell^*, i^*, \pi^*)$ such that $1 \leq i^* \leq q(\lambda)$, then $\text{Prob}[\,\mathsf{catVerify}(pk, i^*, \ell^*, \pi^*) = 1] \approx 0$.

*Proof.* Let $\epsilon(\lambda) := \text{Prob}[\,\mathsf{catVerify}(pk, i^*, \ell^*, \pi^*) = 1]$ and assume towards contradiction that $\epsilon \not\approx 0$. Then, there exists an efficient non-adaptive adversary $\mathcal{A}$ that outputs $q$ elements $\ell_1, \ldots, \ell_q$ before obtaining the public key $pk$ together with the corresponding proofs $\pi_i$ that contain the authentication paths $\hat{\pi}_i$. Afterwards, it outputs a tuple $(\ell^*, i^*, \pi^*)$, where $\pi^*$ is the authentication path in the tree such that $1 \leq i^* \leq q(\lambda)$ and $\mathsf{catVerify}(pk, i^*, \ell^*, \pi^*) = 1$ holds with probability $\epsilon(\lambda)$. In what follows, we show how to either build an attacker $\mathcal{B}_h$ that finds a collision in the hash function, or an algorithm $\mathcal{B}_{\mathsf{ch}}$ against the collision-resistance of the chameleon hash function. Within the proof we use the following notions.

<ins>Setup of the Tree:</ins> The setup procedure for both algorithms $\mathcal{B}_h$ and $\mathcal{B}_{\mathsf{ch}}$ is the same: The algorithm $\mathcal{B}_S$ (where $\mathcal{B}_S \in \{\mathcal{B}_h, \mathcal{B}_{\mathsf{ch}}\}$ depending on the case) runs a black-box simulation of $\mathcal{A}$, in order to obtain the $q$ leaves $\ell_1, \ldots, \ell_q$. Denote by $t$ the depth of the resulting tree, i.e., $2^t \geq q$ and by $R$ the list of random oracle query/answer pairs that is initially empty. The algorithm $\mathcal{B}_S$ computes the tree from the bottom up as follows. First, it sets $u_{1,1} \leftarrow \ell_0 \| \ell_1$, it picks a fresh randomness $r_{1,1}$ and a random value $\rho \leftarrow \{0,1\}^{2len}$, and computes $\rho_1 \leftarrow \mathsf{ch}(x_{1,1}; r_{1,1})$. Afterwards, $\mathcal{B}_\ell$ programs the mapping of the random oracle as $H(\rho) := \rho_1$, it stores $(\rho, \rho_1)$ in $R$, and sets $pk := \rho$.

In order to add the leaves $\ell_3, \ell_4$ to the tree, $\mathcal{B}_S$ sets $u_{1,1} \leftarrow H(\ell_1 \| \ell_2)$ and $u_{1,2} \leftarrow \mathsf{ch}(\ell_3 \| \ell_4; r_{1,2})$ for a random value $r_{1,2}$. $\mathcal{B}_S$ computes the root by picking $r_{2,1}$ uniformly at random and by programming the random oracle as $H(\rho_1) = H(H(\rho)) := \mathsf{ch}(u_{1,1} \| u_{1,2}; r_{1,2})$. $\mathcal{B}_S$ adds the pair $(\rho_1, \rho_2)$ in $R$, where $\rho_2 = \mathsf{ch}(u_{1,1} \| u_{1,2}; r_{1,2})$.

$\mathcal{B}_S$ repeats these steps until all elements are stored in the tree. Following these steps, it is easy to see that all elements verify as valid leaves in the tree. $\mathcal{B}_S$ then returns $(pk, \{(\ell_i, i, \pi_i)\}_{i=1}^{q})$ to $\mathcal{A}$.

<ins>Answering Random Oracle Queries:</ins> $\mathcal{B}_S$ answers all random oracle queries $x$ from $\mathcal{A}$ as follows. If there exists an element $(x, \cdot)$ in $R$, then $\mathcal{B}_S$ returns the first one. (If there is more than one element $(x, \cdot)$ in $R$, then $\mathcal{B}_S$ aborts. Otherwise, if such a pairs does not exist, then $\mathcal{B}_S$ samples a value $y \in \{0,1\}^{len}$ uniformly at random. If there exists an element $(\cdot, y) \in R$, then $\mathcal{B}_S$ repeats keeps sampling until it finds a fresh value $y$. $\mathcal{B}_S$ stores $(x, y)$ in $R$ and returns $y$ to $\mathcal{A}$.

Computing a Collision: We show how to build an attacker $\mathcal{B}_H$ against the collision-resistant of $H$ using an attacker $\mathcal{A}_H$. The reduction against the collision-resistance of $\mathsf{ch}$ works analogously and is omitted.

Eventually, $\mathcal{A}_H$ stops outputting a pair $(\ell^*, i^*, \pi^*) \notin Q$ with $\pi^* = \hat{\pi}^* = ((v^*_{1,\lfloor i/2 \rfloor}, \ldots, v^*_{D-2,\lfloor i/2^{D-2} \rfloor}), R^*)$ and $1 \leq i^* \leq q$. Let $\mathsf{mPath}^* = (v'^*_0, \ldots, v'^*_{D-1})$ denote the path from the leaf $\ell^*$ to the root. If $\mathcal{A}_H$ succeeds, then the nodes $\hat{\pi}^*$ authenticates the leaf $\ell^*$, but $\mathcal{A}_H$ has not received the tuple $(\ell^*, i^*, \hat{\pi}^*)$, i.e., $(\ell^*, i^*, \hat{\pi}^*) \notin Q$ where $Q = ((\ell_1, 1, \hat{\pi}_1), \ldots, (\ell_{q(\lambda)}, q, \hat{\pi}_q))$. Now consider the leaf $\ell_{i^*}$ together with the corresponding authentication path $\hat{\pi} = ((v_{1,\lfloor i/2 \rfloor}, \ldots, v_{D-2,\lfloor i/2^{D-2} \rfloor}), R)$ and with its path $\mathsf{mPath} = (v'_0, \ldots, v'_{D-1}))$ to the root. Let $\hat{\pi}_{i^*} := (v'_0, \ldots, v'_{i^*})$ be the sub-path and let $\mathsf{mPath}_{i^*} = (i, s, (v'_0, \ldots, v'_{i^*}))$, respectively. Then, we distinguish two cases:

Case 1: Suppose that $\mathsf{mPath} \neq \mathsf{mPathMsgFake}$. Since both paths have the same root $\rho$, there must exist an index $0 \leq i < D-1$ with $\mathsf{mPath}_{i+1} = \mathsf{mPathMsgFake}_{i+1}$ and $\mathsf{mPath}_i \neq \mathsf{mPathMsgFake}_i$. Now, a collision is found since $\mathsf{mPath}_{i+1} = H(\hat{\pi}_i || \mathsf{mPath}_i)$

Case 2: Suppose that $\mathsf{mPath} = \mathsf{mPathMsgFake}$. If $\ell_i \neq \ell^*$, then a collision is found. On the other hand, if $\ell_i = \ell^*$, then $\hat{\pi}_i$ and $\hat{\pi}^*$ are distinct. Suppose that $\hat{\pi}_i \neq \mathsf{myPathFake}_i$ for an index $i < D-1$. Since $\mathsf{mPath}_{i+1} = H(\hat{\pi}_i || \mathsf{mPath}_i)$ and because $\mathsf{mPathMsgFake}_{i+1} = H(\mathsf{myPathFake}_i || \mathsf{mPathMsgFake}_i)$ a collision is found.

For the analysis, it is easy to see that $\mathcal{B}$ terminates (in particular when answering the random oracle queries). Furthermore, the probability that $\mathcal{B}$ aborts when answering the random oracle queries is negligible. Thus, we assume in the following that $\mathcal{B}$ does not abort. Then, it follows easily from the reduction that $\mathcal{B}_H$ performs a perfect simulation from $\mathcal{A}_H$'s point of view, and that both algorithms are efficient. Thus, $\mathcal{B}_H$ finds a collision whenever $\mathcal{A}_H$ succeeds. Denote by $\epsilon_H$ the corresponding probability. Assuming that $\epsilon_H$ is non-negligible, however, contradicts the assumption that the hash function is collision-resistant.

The algorithm $\mathcal{B}_{\mathsf{ch}}$ that finds a collision in the chameleon hash function $\mathsf{ch}$ works analogously to $\mathcal{B}_H$. We denote by $\mathcal{A}_{\mathsf{ch}}$ the underlying adversary and by $\epsilon_{\mathsf{ch}}$ its success probability. The algorithm $\mathcal{B}$ then simply guesses if it has access to $\mathcal{A}_H$ or $\mathcal{A}_{\mathsf{ch}}$. Thus, we calculate the overall success probability $\epsilon'(\lambda)$ of $\mathcal{B}$ as

$$\epsilon'(\lambda) := \mathrm{Prob}[\,\mathcal{B}\ \mathsf{succ}] = \frac{1}{2}(\epsilon_H(\lambda) + \epsilon_{\mathsf{ch}}(\lambda)),$$

where $X$ $\mathsf{succ}$ denotes the event that the algorithm $X$ wins its security game.

*Claim.* If $\mathcal{A}$ is a non-adaptive adversary that outputs $(\ell^*, i^*, \pi^*)$ such that $i^* > q(\lambda)$, then $\mathrm{Prob}[\,\mathsf{catVerify}(pk, i^*, \ell^*, \pi^*) = 1] \approx 0$.

The main observation to prove this theorem is that any authentication path for a leaf $\ell^*$ with index $i^*$ for $q+1 \leq i^* \leq 2^D$ must contain a right-handed node on a subtree where no leaf has been added yet. Recall that every right-handed node is computed by chameleon hash function using a randomly chosen value. Thus, if the attacker manages to compute a valid authentication path, then it must must invert the chameleon hash on this value. It might be the case, however, that we find a different pre-image. But then we can build a reduction against the collision resistance.

*Proof.* We prove this theorem by contradiction assuming that $\mathcal{A}$ is an efficent adversary that returns a tuple $(\ell^*, i^*, \hat{\pi}^*)$ such that $q+1 \leq i^* \leq 2^D$. Then, we construct an attacker $\mathcal{B}$ against the one-wayness of $\mathsf{ch}$ (resp. against the collision-resistance).

<u>Setup:</u> The input of $\mathcal{B}$ is an image $y^* \leftarrow \mathsf{ch}(x; r)$ and the public key $cpk$ of the chameleon hash function. It runs $\mathcal{A}$ in a black-box way obtaining $q$ leaves $\ell_1, \ldots, \ell_q$. Let $D$ be the depth of the resulting tree such that $2^D \geq q$ and denote by $t$ the unused dummy nodes (i.e., these are right nodes that do not have any children and that would be computed by a dummy value). $\mathcal{B}$ sets up the tree by picking $t-1$ dummy nodes $y_i \leftarrow \{0,1\}^{len}$ and guessing a random index $j = 1, \ldots, t$. Then, $\mathcal{B}$ computes the tree from the bottom to the top using where the first $q$ leaves are $\ell_1, \ldots, \ell_q$. To do so, it programs the random oracle as described in the proof of Proposition 1 and it uses the nodes $y_1, \ldots, y^*, \ldots, y_{t-1}$ as dummy nodes. Furthermore, $\mathcal{B}$ computes the authentication paths $\hat{\pi}_i$ for the leaves $\ell_i$ (for $i = 1, \ldots, q$). The attacker $\mathcal{B}$ then sets $pk \leftarrow (cpk, \rho)$ and runs a black-box simulation of $\mathcal{A}$ on input $(pk, \{(\ell_i, i, \hat{\pi}_i)\}_{i=1}^q)$.

<u>Answering the Random Oracle Queries:</u> Our reduction answers the random oracle queries by lazy sampling as described in the proof of Proposition 1.

<u>Inverting $\mathcal{CH}$:</u> Eventually, the attacker $\mathcal{A}$ terminates, outputting a pair $(\ell^*, i^*, \hat{\pi}^*)$ with $\hat{\pi}^* = ((v^*_{1,\lfloor i/2 \rfloor}, \ldots, v^*_{D-2,\lfloor i/2^{D-2} \rfloor}), R^*)$ and $q+1 \leq i^* \leq 2^D$. Let $\mathsf{mPath}^* = (v'^*_0, \ldots, v'^*_{D-1})$ denote the path from the leaf $\ell^*$ to the root. If there exists an index $j$ such that $v'_j = y^*$, then $\mathcal{B}$ computes the authentication path $\hat{\pi}^*$ up to $y^*$. Then, the algorithm $\mathcal{B}$ outputs the resulting pre-image $x^* = \hat{\pi}^*$ together with the corresponding randomness $r^*$. Otherwise, it aborts.

For the analysis first note that $\mathcal{B}$ performs a perfect simulation from $\mathcal{A}$'s point of view (applying the same arguments as in the previous proof). Now, assume that $\mathcal{A}$ succeeds with non-negligible probability. Then, it returns a valid pair $(\ell^*, i^*, \hat{\pi}^*)$ with $\hat{\pi}^* = ((v^*_{1,\lfloor i/2 \rfloor}, \ldots, v^*_{D-2,\lfloor i/2^{D-2} \rfloor}), R^*)$ such that $q+1 \leq i^* \leq 2^D$. Since the authentication path verifies, it follows from our construction that one node in $\hat{\pi}^*$ is a right-handed node on the authentication path of $\ell_q$. Since it is a right node, it follows that this node is the output of a chameleon hash function. Now, assume that $\mathcal{B}$ has guessed this node correctly. Then, it follows from our construction that $\mathcal{B}$ computes a pre-image $(x^*, r^*)$ of $y^*$. We have to show that $(x^*, r^*) = (x, r)$. This, however, follows from the collision-resistance of $\mathsf{ch}$. If we assume towards contradiction that $(x^*, r^*) \neq (x, r)$, then we can easily build an adversary that finds a collision in $\mathsf{ch}$.

Assume further that $\mathcal{A}$ succeeds with non-negligible probability $\epsilon_O(\lambda)$. Then, it is easy to see that $\mathcal{B}$ wins with probability $\delta_O(\lambda) := \epsilon_O(\lambda)/t$. This, however, either contradicts the one-wayness or the collision-resistance of $\mathsf{ch}$.

The next step of the proof to achieve full security, is to apply the general transformation that turns any weakly secure CAT into a fully secure one. The basic idea is to store chameleon hash values (using a different trapdoor) in the leaves, such that the reduction can adjust these values for each query. Let $\Pi'_{\mathsf{CAT}}$ be the CAT that is identical to the scheme defined in Construction 1, with the difference being that the leaves store the values of a chameleon hash with an independently chosen key.

**Proposition 2.** *If $\Pi_{\mathsf{CAT}}$ is a weakly secure chameleon authentication tree and $\mathcal{CH}$ is a collision-resistant chameleon hash function, then $\Pi'_{\mathsf{CAT}}$ as defined above is a secure chameleon authentication tree that does not support updates.*

The proof is almost the same as the one for signature schemes (see [11]) and is omitted.

**Proposition 3.** *The chameleon authentication tree as defined in Construction 1 is secure against adaptive updates.*

The main idea of the proof of this proposition is as follows. Whenever, $\mathcal{A}$ asks to update an element in the tree, then we pick a fresh value $\rho'$ such that the mapping $H(\rho'), \ldots, H^t(\rho')$ are unknown. If this is the case, then we program the mapping of the hash function accordingly. Otherwise, if one of the mappings are known, then the reduction aborts.

## C $\delta$-bounded CATs

The fully dynamic CAT is more efficient than previous constructions [22] and allows the data owner to upload an unbounded amount of data. However, the proof is only given in the random oracle model and the construction requires the additional inversion property of chameleon hash functions. Although two of the three chameleon hash functions we consider, namely the Fiat-Shamir [3] and the Ateniese and de Medeiros [2] construction have this property, it is still desirable to find a solution based on weaker assumptions, which can be proven in the standard model. Therefore we propose the $\delta$-bounded CAT, which is upper bounded by the depth $\delta$, but is provably secure in the standard model and has roughly the same computational and bandwidth overhead as the fully dynamic construction.

## C.1 Intuition

Let us reconsider our first construction. There, we exploited the inversion property of our chameleon hash function to find randomnesses that mapped to certain root values. To provide a proof in the standard model we have to refrain from using this property. Instead, we pre-compute $\delta$ dummy root values $\rho_{h,0}$ where $h = 1 \dots \delta$, publish them in the public key $pk$, and keep their pre-images secret in our state $\mathsf{st}$. An authentication path with depth $i$ is then verified against $\rho_{i,0}$. Since we keep all pre-images in our state, we can use $\mathsf{col}$ rather than $\mathsf{scol}$ to find collisions.
Note, again we can make use of a pseudorandom function to make the client's state constant.

## C.2 Construction

We now provide the formal description of all algorithms of the $\delta$-bounded CAT construction.

**Construction 2** *Let $H : \{0,1\}^* \mapsto \{0,1\}^{len}$ be a hash function and $\mathcal{CH} = (\mathsf{chGen}, \mathsf{ch}, \mathsf{col})$ a chameleon hash function. The $\delta$-bounded chameleon authentication tree $\Pi_{\mathsf{CAT}} = (\mathsf{catGen}, \mathsf{catAdd}, \mathsf{catUpdate}, \mathsf{catVerify})$ is defined as follows:*

$\underline{\mathsf{catGen}(1^\lambda, \delta)}$: *The algorithm computes $(cpk, csk) \leftarrow \mathsf{chGen}(1^\lambda)$ and sets $c \leftarrow 0$. For $i = 1, \dots, \delta$ it generates dummy nodes $\rho_{i,0}$ and stores their pre-images in $\mathsf{st}$. It returns the private key $\mathsf{sp} = (csk, \mathsf{st}, \mathsf{vp})$ and the public key $\mathsf{vp} = (cpk, (\rho_{1,0}, \dots, \rho_{\delta,0}))$*

$\underline{\mathsf{catAdd}(\mathsf{sp}, s)}$: *Parse $\mathsf{sp}$ as $(csk, \mathsf{st}, \mathsf{vp})$ and check whether the current tree is full, i.e., whether $c$ is a power of two:*

<u>The counter $c$ is a power of two:</u> *In this case the tree is full again. We need to extend its height by one to create a tree which has free leaves. Let d be its depth before increasing it by one. If $d = \delta$ we abort, since the tree has reached its maximum capacity. Otherwise, we compute a dummy node $v_{d-1,1}$, and store its pre-images in $\mathsf{st}$. Next, we need to compute $r'_{d,0}$ such that $\mathsf{ch}(\rho_{d-1,0} \| v_{d-1,1}, r'_{d,0}) = \rho_d$. We use the stored pre-image $(x_{d,0}, r_{d,0})$ of $\rho_{d,0}$ from $\mathsf{st}$ and compute $r_{d,0} \leftarrow \mathsf{col}(csk, x_{d,0}, r_{d,0}, \rho_{d-1,0} \| v_{d-1,1})$. Now we add $v_{d-1,1}$, $r'_{d,0}$ to $\pi$ in $\mathsf{st}$ and proceed as in the case where $c$ is not a power of two.*

<u>The counter $c$ is not a power of two:</u> *In this case the tree is not full. The algorithms behaviour here is identical to the one in the fully dynamic version as defined in Constrution 1.*

$\underline{\mathsf{catVerify}(\mathsf{vp}, i, \ell, \pi)}$: *Parse $\mathsf{vp}$ as $(cpk, (\rho_{1,0}, \dots, \rho_{\delta,0}))$. In order to verify, whether $\pi$ authenticates $\ell$ we compute, starting from the bottom, each node as the hash or the chameleon hash of its two children until we compute a node with index 0. If the a nodes index is odd, we compute it using the chameleon hash function, and we use the hash function otherwise. All required nodes and randomnesses* *are taken from $\pi$. Let $v_{d,0}$ be the node at which we terminated. Return 1 iff $v_{d,0} = \rho_{d,0}$, and 0 otherwise.*

$\underline{\mathsf{catUpdate}(\mathsf{sp}, i, \ell')}$: *Parse* $\mathsf{sp}$ *as* $(csk, \mathsf{st}, \mathsf{vp})$*. Request $\ell_i$, with its proof of correctness $\pi_i$, and compute* $\mathsf{catVerify}(\mathsf{vp}, i, \ell_i, \pi_i)$*; abort if it outputs 0. Otherwise, replace $\ell$ with $\ell'$ and recompute the new value of the corresponding root $v_{d,0}$. Update the root node's value at that height in the public key* $\mathsf{vp}'$ *accordingly. Recompute all root node values above $v_{d,0}$ update the* $\mathsf{vp}'$ *accordingly.*

Regarding security, we obtain the following theorem.

**Theorem 2.** *Suppose that $\mathcal{CH}$ is a one-way collision-resistant chameleon hash function and $H$ is a collision-resistant hash function, then Construction 2 is a secure $\delta$-bounded chameleon authentication tree.*

The proofs is similar to the previous one, with the difference that we do not need to program the random oracle anymore, and can easily be deduced.

## D Evaluation of Chameleon Hash Functions

We discuss the performance of chameleon hash functions on their own, since they represent the most expensive building block in our protocols. In particular, we examine the hashing and collision finding performances of the Fiat-Shamir, the Ateniese and de Medeiros, its elliptic curve equivalent, and the Krawczyk-Rabin chameleon hash.

To evaluate their performances, we used each of them to compute 2000 hashes for randomly generated 160 bit long messages and then computed the average time it took. We used a security parameter of 2048 and chose all sizes in the underlying primitives according to the NIST Recommendations 2012 [17]. For the elliptic curve variation of the Ateniese and de Medeiros hash we used the P-224 curve.

The collision finding performances were measured by running the experiment above with the difference that we additionally computed a collision for another randomly generated message after each hash operation. The average times for computing one hash, or one hash and one collision respectively are depicted in Table 2.

One can see that when only performing the hash operation, the Fiat-Shamir construction is the fastest one. Unfortunately its performance for computing collisions is very poor, which renders it infeasible for applications that require high throughput. Quiet interestingly Ateniese and de Medeiros is slower than its elliptic curve pendant. Further tests with a smaller security parameter like 1024 showed that the elliptic curve variant is slower at first, but scales much better, when the security parameter increases. As expected from the mathematical description of the Krawczyk-Rabin chameleon hash, it performs very well and its

|  | Fiat-Shamir | Krawczyk-Rabin | Ateniese and de Medeiros | Ateniese and de Medeiros (EC) |
|---|---|---|---|---|
| Hash | 6.501 | 10.213 | 26.617 | 7.637 |
| Hash and Coll. | 2143.046 | 10.2305 | 54.1225 | 13.134 |

**Table 2.** Chameleon hash function benchmarks in milliseconds.

collision finding algorithm is extremely efficient. However, it is not invertible and therefore it cannot be used in the dynamic constructions.