

Verifiable and Provably Secure Machine Unlearning

Thorsten Eisenhofer^{+‡}, Doreen Riepel[‡], Varun Chandrasekaran^{*†},
Esha Ghosh^{*}, Olga Ohrimenko[§], Nicolas Papernot^{¶||}

Ruhr University Bochum[‡], Microsoft Research^{*}, The University of Melbourne[§],
University of Illinois Urbana-Champaign[†], University of Toronto[¶], Vector Institute^{||}

Abstract—Machine unlearning aims to remove points from the training dataset of a machine learning model after training; for example when a user requests their data to be deleted. While many machine unlearning methods have been proposed, none of them enable users to audit the unlearning procedure and verify that their data was indeed unlearned. To address this, we define the first cryptographic framework to formally capture the security of verifiable machine unlearning. While our framework is generally applicable to different approaches, its advantages are perhaps best illustrated by our instantiation for the canonical approach to unlearning: retraining without the data to be unlearned. In our protocol, the server first computes a proof that the model was trained on a dataset D . Given a user data point d , the server then computes a proof of unlearning that shows that $d \notin D$. We realize our protocol using a SNARK and Merkle trees to obtain proofs of update and unlearning on the data. Based on cryptographic assumptions, we then present a game-based proof that our instantiation is secure. Finally, we validate the practicality of our constructions for unlearning in linear regression, logistic regression, and neural networks.

I. INTRODUCTION

The right to be forgotten entitles individuals to self-determine the possession of their private data and compel its deletion. In practice, this is now mandated by recent regulations like the GDPR [1], CCPA [2], or PIPEDA [3]. Consider, for example, the case where a company or service provider collects data from its users. These regulations allow users to request a deletion of their data, and legally compels the company to fulfil the request. However, this can be challenging when the data is used for downstream analyses, *e.g.*, training machine learning (ML) models, where the relationship between model parameters and the data used to obtain them is complex [41]. In particular, ML models are known to memorize information from their training set [23], [17], resulting in a myriad of attacks against the privacy of training data [19], [44].

Thus, techniques have been introduced for *unlearning*: a trained model is updated to remove the influence a training point had on the model’s parameters and predictions [18]. Yet, regardless of the particular approach, all existing techniques [68], [15], [37], [29], [31], [7], [53] suffer from one critical limitation: they are unable to provide the user with a proof that their data was indeed unlearned. Put another way, the user is asked to blindly trust that the server executed the unlearning algorithm to remove their data with no ability

to verify this. This is problematic because dishonest service providers may falsify unlearning to avoid paying the large computational costs or to maintain model utility [54], [28].

Additionally, verifying that a point is unlearned is non-trivial *from the user’s perspective*. A primary reason is that users (or third-party auditors) can not determine whether a data point is unlearned (or not) by comparing the model’s predictions or parameters before and after claimed unlearning. The complex relationship between training data, models’ parameters, and their predictions make it difficult to isolate the effects of any training point. In fact, prior work [60], [63] demonstrate that under certain constraints, a model’s parameters can be identical when trained with or without a data point.

To address these concerns, we propose an algorithmic approach to verify unlearning. Rather than trying to verify unlearning by examining changes in the model, we ask the service provider (*i.e.*, the server) to present a cryptographic proof that the process to unlearn the user’s data was executed correctly. This leads us to propose a general framework to provide a verifiable protocol for machine unlearning.

While our framework is applicable to different unlearning approaches, it is perhaps best illustrated with the canonical approach of naively retraining the model from scratch. Here, the server retrains a model without using a data point d that needs to be unlearned. In this example, we identify the following guarantees that need to be satisfied: (a) the user’s data point is no longer present in the server’s training set D , and (b) the model was trained from that dataset D (without the user’s data point). Thus, the framework has two major components. First, the server computes a *proof of update* whenever the model’s training data changes: this establishes that they trained the model on a particular dataset D . Second, at any point in time, when a user submits a request to unlearn data point d , the server can leverage its dataset D to provide the user with a *proof of unlearning*. This proves non-membership of their data point d in the training set D . This proof of $d \notin D$ together with the proof of update allows the user to verify the aforementioned guarantees.

This framework also allows a server to prove to a user that their data was deleted from the model’s training data, without revealing anything else about the dataset. To reason about the guarantees provided by our unlearning framework, we propose *the first formal security definition* of verifiable machine unlearning. Under this definition, we can instantiate protocols using exact unlearning techniques—as illustrated above with the example approach of naively retraining—and

⁺Work done while the author was interning at Vector Institute.

prove their security.

Our practical instantiation is based on SNARK-based verifiable computation for the proof of update and Merkle tree-based authenticated data structures for the proof of unlearning. We implement and demonstrate the versatility and scalability of our construction. We evaluate the unlearning on a variety of binary classification tasks from the PMLB benchmark suite [52] as considered in prior work [5], using linear regression, logistic regression, and neural network classifier models.

Contributions. We make the following contributions:

- *Formal framework.* We introduce a general framework to construct protocols for verifiable exact machine unlearning. Our framework is designed to be general enough to capture different exact unlearning algorithms and secure primitives. The protocols are aimed at generating a proof that unlearning of a point took place.
- *Security definition of verifiable machine unlearning.* We then propose a formal security definition of a verifiable machine unlearning scheme. This game-based definition allows one to prove security of their instantiation of the unlearning protocol. Our framework models verifiable unlearning as a 2-party scheme (executed between the server and the users). We guarantee that even a malicious server (and potentially malicious users colluding with the server) cannot falsely convince a honest user that their data point was unlearned.
- *Instantiation.* We present a concrete construction for unlearning through retraining and prove that it satisfies the completeness requirements and security definition. This construction is based on SNARKs for the proof of update and a Merkle tree to provide a proof of unlearning.
- *Practical implementation.* We implement the protocol’s main functionality, study its applicability to different classes of ML models and real-world datasets, and evaluate its performance. For the largest dataset (with 128 points with 7 features each), the generation of proving a model update takes 1h 25m for a simple fully connected feed-forward neural network. Verification time runs independent in constant time and takes less than 1s.

II. BACKGROUND

In this section, we discuss the preliminaries needed to understand the contributions of our work.

A. Machine Learning Preliminaries

Supervised Machine Learning. Supervised machine learning (ML) is the process of learning a parameterized function f_θ that is able to predict an output (from the space of outputs \mathcal{Y}) given an input (from the space of inputs \mathcal{X}) *i.e.*, $f_\theta : \mathcal{X} \rightarrow \mathcal{Y}$. Commonly learnt functions include linear regression, logistic regression, decision trees, simple perceptrons, and feed-forward neural networks. The parameters of the model are often optimized using methods such as gradient descent (GD), which is formalized by the equation

$$w_t = w_{t-1} - \eta \nabla \mathcal{L}$$

or stochastic gradient descent (SGD), which is formalized by the equation

$$w_t = w_{t-1} - \eta \nabla_{x_{t-1}} \mathcal{L}$$

where \mathcal{L} is a suitably chosen loss function (*e.g.*, cross-entropy loss). The difference is that in SGD, the gradient of the loss (at step $t - 1$) is calculated given a randomly chosen batch of input $x_{t-1} \in \mathcal{X}$, while the entire dataset is used to calculate the gradient in GD.

Machine Unlearning. Machine unlearning is an area of research where the goal is to design algorithms that enable an ML model (specifically, its parameters) to forget the contribution of a (subset of) data point(s). Unlearning can broadly be classified into two categories: *exact unlearning*, where there are guarantees that the data’s contribution is entirely removed [15], [18], [69], [46], and *approximate unlearning* where the guarantees tolerate some error [37], [20], [62], [7], [53], [29], [32].

Verifying Unlearning. Prior work [26], [61] aims at verifying unlearning by embedding backdoors [36] in models (using data whose unlearning is to be verified) and verifying its removal on unlearning. However, such approaches are probabilistic with no theoretical guarantees of when they work, unlike our rigorous cryptography-informed approach which produces verifiable proofs. The work of Guo *et al.* [37] provides end-users with a certificate that the new model is influenced by the specific data in a quantifiably low manner. While this certificate conceptually bounds the influence of a data point from an algorithmic perspective, it provides no guarantee that the entity executing the algorithm (*i.e.*, server) did so correctly. In our work, we demonstrate just that, and provide cryptographic guarantees of correctness of execution.

B. Cryptographic Preliminaries

Throughout the paper, let λ denote the security parameter. We call a function negligible in λ (denoted by $\text{negl}(\lambda)$) if it is smaller than the inverse of any polynomial for all large enough values of λ . $[m : n]$ denotes the set $\{m, m + 1, \dots, n\}$ for integers $m < n$. For $m = 1$, we simply write $[n]$. $y \leftarrow \mathbb{M}(x_1, x_2, \dots)$ denotes that on input x_1, x_2, \dots , the probabilistic algorithm \mathbb{M} returns y . An adversary \mathcal{A} is a probabilistic algorithm, and is *efficient* or Probabilistic Polynomial-Time (PPT) if its run-time is bounded by some polynomial in the length of its input. We will use code-based games, where $\Pr[G \Rightarrow 1]$ denotes the probability that the final output of game G is 1. For a boolean statement B , $\llbracket B \rrbracket$ refers to a bit that is 1 if the statement is true and 0 otherwise.

Collision-Resistant Hash (Hash). A function $\text{Hash} : \{0, 1\}^n \rightarrow \{0, 1\}^\kappa$ is collision-resistant if

- It is length-compressing *i.e.*, $\kappa < n$.
- It is hard to find collisions *i.e.*, for all PPT adversaries \mathcal{A} , and for all security parameters λ ,

$$\Pr \left[\begin{array}{l} (x_0, x_1) \leftarrow \mathcal{A}(1^\lambda, \text{Hash}) : \\ x_0 \neq x_1 \wedge \text{Hash}(x_0) = \text{Hash}(x_1) \end{array} \right] \leq \text{negl}(\lambda).$$

SNARKS (II). Succinct Non-Interactive Arguments of Knowledge (SNARK) are comprised of 3 algorithms (II.Setup, II.Prove, II.Vrfy). Formally,

- $(\text{crs}, \text{td}) \leftarrow \text{II.Setup}(1^\lambda, R)$: The setup algorithm computes a common reference string crs and a simulation trapdoor td for a polynomial-time decidable relation R .
- $\pi \leftarrow \text{II.Prove}(R, \text{crs}, \phi, \omega)$: The prover algorithm takes as input the crs and $(\phi, \omega) \in R$ and returns an argument π , where ϕ is termed the statement and ω the witness.
- $b \leftarrow \text{II.Vrfy}(R, \text{crs}, \phi, \pi)$: The verification algorithm takes the crs , a statement ϕ and an argument π and returns a bit b . $b = 1$ indicates success and $b = 0$ indicates failure.

Perfect Completeness. Given any true statement, an honest prover should be able to convince an honest verifier. More formally, let \mathcal{R} be a sequence of families of efficiently decidable relations R . For all $R \in \mathcal{R}$ and $(\phi, \omega) \in R$

$$\Pr \left[\text{II.Vrfy}(R, \text{crs}, \phi, \pi) \left| \begin{array}{l} (\text{crs}, \text{td}) \leftarrow \text{II.Setup}(1^\lambda, R); \\ \pi \leftarrow \text{II.Prove}(R, \text{crs}, \phi, \omega) \end{array} \right. \right] = 1.$$

Computational Soundness. We say that Π is sound if it is not possible to prove a false statement. Let L_R be the language consisting of statements for which there exists corresponding witnesses in R . For a relation $R \sim \mathcal{R}$, we require that for all non-uniform PPT adversaries \mathcal{A}

$$\Pr \left[\begin{array}{l} \phi \notin L_R \text{ and} \\ \text{II.Vrfy}(R, \text{crs}, \phi, \pi) \end{array} \left| \begin{array}{l} (\text{crs}, \text{td}) \leftarrow \text{II.Setup}(1^\lambda, R); \\ (\phi, \pi) \leftarrow \mathcal{A}(R, \text{crs}) \end{array} \right. \right] \leq \text{negl}(\lambda).$$

We further define the notion of witness extractability or knowledge soundness.

Computational Knowledge Soundness. A SNARK satisfies computational knowledge soundness if there is an extractor that can compute a witness whenever the adversary produces a valid argument. More formally, for a relation $R \sim \mathcal{R}$, we require that for all non-uniform PPT adversaries \mathcal{A} there exists a non-uniform PPT extractor \mathcal{E} such that

$$\Pr \left[\begin{array}{l} (\phi, \omega) \notin R \text{ and} \\ \text{II.Vrfy}(R, \text{crs}, \phi, \pi) \end{array} \left| \begin{array}{l} (\text{crs}, \text{td}) \leftarrow \text{II.Setup}(1^\lambda, R); \\ ((\phi, \pi); \omega) \leftarrow (\mathcal{A} \parallel \mathcal{E})(R, \text{crs}) \end{array} \right. \right] \leq \text{negl}(\lambda).$$

We say a SNARK Π is secure if it satisfies perfect completeness, computational soundness and knowledge soundness.

III. VERIFIABLE MACHINE UNLEARNING

We now propose our approach for designing a formal framework for verifiable unlearning. This framework allows to rigorously define and prove requirements for concrete instantiations of verifiable unlearning protocols. We consider the following ecosystem: there are many *users*, each of whom has access to a set of data points (or dataset) that can be uniquely identified *e.g.*, this can be achieved by combining an identifier for the user with the data point(s). These users share their data with a *server* which uses it to learn an ML model.

Our focus is on a user requesting its data to be deleted by the server.

Requirements. The server should (**R1**) comply and ensure that the data is deleted, and (**R2**) update the ML model to account for the deletion of the requested data points. This step may be performed with exact or approximate unlearning techniques, each introducing different trade-offs (more details are provided in Section IV-B). Then, the server must prove that the aforementioned operations have been executed correctly.

Motivation. There is a need for verifiable machine unlearning because the server may not perform **R1** and/or **R2**. This is because the server may not be willing to tolerate a degradation in the model’s performance after data deletion [54], [28], or pay the computational penalty associated model updating.

In the status quo, there is no rigorous way for the user to verify if the model being used is devoid of their data. A naive solution would be to provide the user with the trained parameters of the model, and request them to locally run influence techniques [41] to understand if their data contributes to these parameters. However, this does not suffice; recent work by Shumailov *et al.* [60] and Thudi *et al.* [63] describe how a user’s contribution (towards model parameters) can be approximated from other entries in a dataset rendering such approaches meaningless: the server can claim to have obtained the exact same model parameters from a number of different datasets. The data attribution problem is far more fundamental; correlated entries in a dataset may result in similar contributions to the model’s final parameters. Imagine that two users A and B have the same (or even similar) data: if A requests unlearning but not B, user A may be surprised to find that some influence techniques indicate that their data still has an effect on the model—which would be possible even if the server had unlearned their data because the data of user B is still in the model’s training set.

Furthermore, in some cases, the trained models are proprietary information, and providing users access to the parameters would be impossible. In such settings, how does the user assess their particular contribution (towards a trained model’s parameters), without access to (a) the trained model’s parameters, and (b) the training dataset?

Desiderata. From the discussion thus far, we unearth two main requirements to achieve our goals.

- D1.** Given an ML model that was trained on a dataset D , machine unlearning mechanisms update this model by *removing* data points from set D . We require a *proof of update* to establish that the *updated* model was obtained from the modified dataset.
- D2.** Further, to prove unlearning we require a proof that a data point d is not part of the (modified) training data D . More specifically, we require a *proof of unlearning* that proves non-membership in this set (*i.e.*, $d \notin D$).

Threat Model. We assume (a) the user requesting their data point to be unlearned is honest, and (b) the server is malicious,

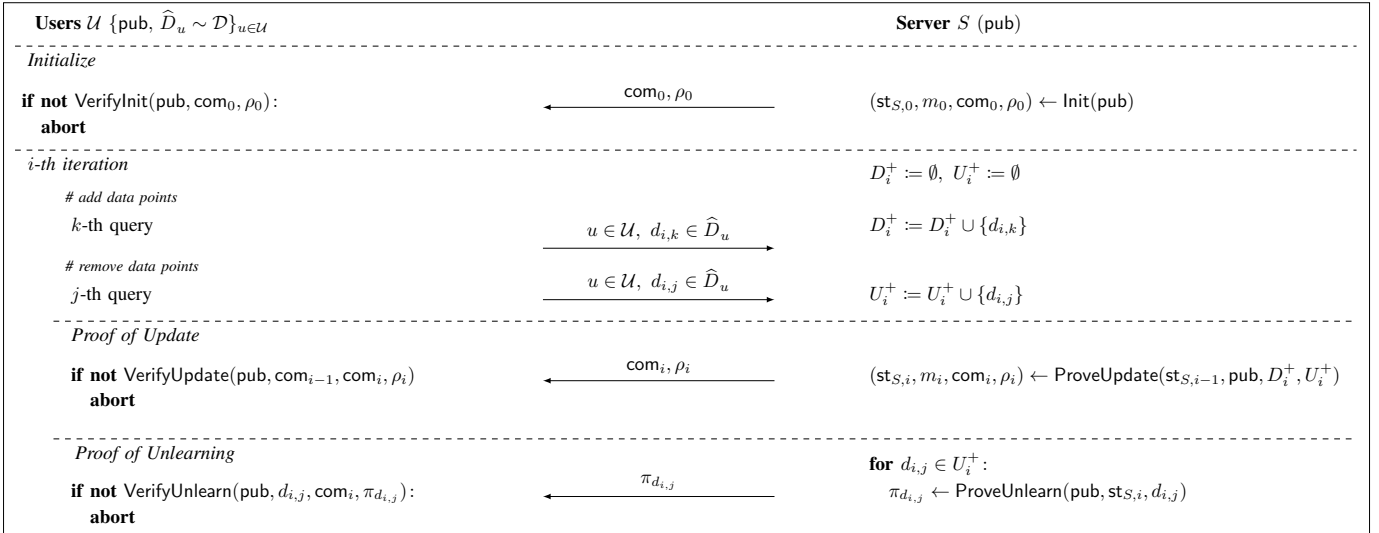


Fig. 1: **Unlearning Framework.** Our framework allows to instantiate protocols for verifiable machine unlearning. We assume a set of users \mathcal{U} and a server S . After initialization, execution proceeds in iterations. In the beginning of each iteration i , users can issue requests for data to be added or deleted. After this phase, the server updates the model with any added or deleted data points. It computes a commitment com_i on the updated model m_i and updated training dataset. Furthermore, the server creates a *proof of update* to prove to the users that m_i was obtained from this dataset. The users verify this proof and the commitment. Subsequently, the server creates a *proof of unlearning* for every unlearned data point conforming to a user that it has complied with a data deletion request. This proof can be verified by the user against com_i .

(c) all other users could potentially collude with the server but still follow the protocol faithfully. The goal of the server is to prove to a user that a data point was unlearned, but is in fact included in its training data.

We also expect the server to naturally exercise access control mechanisms. If user A requests her data to be deleted, the server has some out-of-the-band authentication mechanism to ensure that it is indeed A who is making the request. Since this can be done using standard cryptographic authentication mechanisms, we do not model this explicitly in our threat model. In Section VII-C, we discuss this further with other practical considerations.

IV. OUR FRAMEWORK

We now present our formal framework for verifiable machine unlearning. This framework defines a generic protocol for unlearning (an overview is depicted in Figure 1) that allows various instantiations (e.g., using different unlearning techniques or cryptographic primitives). We consider unlearning protocols that are executed interactively by the two roles introduced in the previous section: a set of users \mathcal{U} and a server S . In this section, we introduce our framework, and define requirements for completeness and security. In the section that follows, we will give an example of a full instantiation of an unlearning protocol using retraining-based exact unlearning, verified computation and authenticated data structures based on SNARKs and Merkle trees. We then prove its completeness and security in our framework.

Dataset and Data Points. Let \mathcal{D} be the distribution of data points. Each user u possesses a set of data points $\widehat{D}_u \sim \mathcal{D}$.

At the server side, there is initially an empty dataset $D_0 = \emptyset$ (which is later populated for training an ML model). During the execution of the unlearning protocol, users can request to add or delete their data points. Different versions of the server’s dataset (used for training) is denoted by its corresponding index (i.e., D_0, D_1, \dots). Since each data point is distinctly identifiable (refer Section III) and because we assume that the user requesting deletion is honest, the server can distinguish between identical data contributed by different users while processing unlearning requests. Furthermore, a data point cannot be added after it has been deleted, and similarly, a user may request the deletion of a data point which has not yet been added (with the result that it then can never be added in future). Looking ahead, this is exactly what security of such a protocol aims to ensure (refer Section IV-C).

A. Framework Overview

The framework defines a generic interface for verifiable machine unlearning and is depicted in Figure 1. We can describe the execution of an unlearning protocol by three phases (executed in an iterative manner):

P1. Data Addition/Deletion: Any user can issue an addition/deletion request to the server at any iteration. The server can batch multiple addition/deletion requests within a single iteration. The server then adds/removes the data points to/from the training dataset.

P2. Proof of Update: At the end of each iteration i , the server modifies its training set using the data points added/deleted, and uses the resulting training dataset to

update the model. It then computes a proof of this update. This update is verified by the corresponding users.

P3. Proof of Unlearning: Finally, the server computes a proof of unlearning for each deleted data point which proves that the data point is not contained in the updated training set. The server sends each proof to the corresponding user who can use it to verify that their data point was unlearned.

We next describe the interfaces, including all algorithms, in more detail. The completeness properties of these algorithms are formally presented in Section IV-B. Note that a global setup procedure generates public parameters pub which is given to all actors.

1. Initialization. During initialization, the ML model and a state (which captures information needed for subsequent iterations) between the server and users are initialized using the Init and VerifyInit algorithms. Note that the party executing the algorithm is denoted in bold font to the left of the algorithm. Formally, the algorithms are defined as follows.

Server: $(\text{st}_{S,0}, m_0, \text{com}_0, \rho_0) \leftarrow \text{Init}(\text{pub})$

Init takes as input the public parameters pub and outputs (a) initial state $\text{st}_{S,0}$, (b) model m_0 , (c) commitment com_0 , and (d) proof ρ_0 . The algorithm runs as follows: the server first suitably initializes the model m_0 and the set of training points $D_0 := \emptyset$. It then commits to both of these with $\text{com}_0 := (\text{com}_0^m \parallel \text{com}_0^D)$ where each commitment corresponds, respectively, to m_0 and D_0 . Finally, proof ρ_0 attests the initialization of model m_0 .

User: $0/1 \leftarrow \text{VerifyInit}(\text{pub}, \text{com}_0, \rho_0)$

VerifyInit takes as input the public parameters pub , a commitment com_0 , and a proof ρ_0 . If the verification is successful, it outputs 1. On failure, it outputs 0. The user verifies (a) commitment com_0 with $D_0 = \emptyset$, and (b) model initialization m_0 against the proof ρ_0 .

2. Proof of Update. In each iteration i , the server executes ProveUpdate to update the model with any newly added or deleted data points and proves that this was done correctly. The users verify the proof with VerifyUpdate . Formally, we define the two algorithms as follows:

Server: $(\text{st}_{S,i}, m_i, \text{com}_i, \rho_i) \leftarrow \text{ProveUpdate}(\text{st}_{S,i-1}, \text{pub}, D_i^+, U_i^+)$

ProveUpdate takes as input the previous state $\text{st}_{S,i-1}$, public parameters pub , the set of new data points D_i^+ , and the set of points requested to be unlearned U_i^+ . It outputs (a) the updated state $\text{st}_{S,i}$, (b) model m_i , (c) commitment com_i and (d) proof ρ_i . The algorithm runs as follows: the server updates m_i with D_i^+ and U_i^+ . We define the resulting training set of m_i as the union of all added data points excluding data points that were unlearned, *i.e.*, $D_i := \bigcup_{k \leq i} D_k^+ \setminus \bigcup_{k \leq i} U_k^+$. The server commits to both the model and training data with $\text{com}_i := (\text{com}_i^m \parallel \text{com}_i^D)$. We assume that the commitment to the

dataset is computed deterministically from pub and D_i , which we denote by $\text{com}_i^D := \text{Commit}(\text{pub}, D_i)$. The server then computes the proof ρ_i that (1) model m_i was obtained from training data D_i , and (2) training data D_i does not contain any unlearned points, *i.e.*, $D_i \cap U_i = \emptyset$, where $U_i := \bigcup_{k \leq i} U_k^+$. The proof also attests that (3) the previous set of unlearned data points is a subset of the updated set $U_{i-1} \subseteq U_i$. This ensures that an unlearned data point is never re-added into the training data.

User: $0/1 \leftarrow \text{VerifyUpdate}(\text{pub}, \text{com}_{i-1}, \text{com}_i, \rho_i)$

VerifyUpdate takes as input the public parameters pub , two commitments com_{i-1} , com_i and a proof ρ_i . It outputs 1 if the verification is successful, and 0 otherwise. Users validate properties (1)-(3) described in the earlier paragraph by verifying the proof ρ_i against the previous commitment com_{i-1} and the new commitment com_i .

3. Proof of Unlearning. With the proof of unlearning, the server proves to a user that their requested data point is *absent* in the training data of the model. The proof is created with ProveUnlearn and is then verified by the user with VerifyUnlearn . Formally, we define the two algorithms as follows with $d_{i,j}$ being the j -th data point that was requested to be unlearned in iteration i :

Server: $\pi_{d_{i,j}} \leftarrow \text{ProveUnlearn}(\text{pub}, \text{st}_{S,i}, d_{i,j})$

ProveUnlearn takes as input the public parameters pub , state $\text{st}_{S,i}$ and an unlearned data point $d_{i,j}$. It outputs the proof of unlearning $\pi_{d_{i,j}}$. This proof is computed by the server and proves that $d_{i,j}$ is not part of the training set of model m_i , *i.e.*, $d_{i,j} \notin D_i$. The server needs to ensure that $d_{i,j}$ is indeed not in D_i prior to executing ProveUnlearn .

User: $0/1 \leftarrow \text{VerifyUnlearn}(\text{pub}, d_{i,j}, \text{com}_i, \pi_{d_{i,j}})$

VerifyUnlearn takes as input the public parameters pub , the unlearned data point $d_{i,j}$, the commitment of the iteration where $d_{i,j}$ was unlearned and the proof of unlearning $\pi_{d_{i,j}}$. It outputs the result of the verification. The user verifies with both $\pi_{d_{i,j}}$ and com_i that $d_{i,j}$ was not part of the training data of model m_i , *i.e.*, $d_{i,j} \notin D_i$.

Redundant computation. In the discussion above, note that the users are responsible for verification of all proofs generated at every iteration, including those iterations where they do not perform any data addition/deletion. This results in a lot of redundant computation performed across users. If the users trust a third party (*e.g.*, an auditor), then the VerifyUpdate algorithm can be executed by this entity; the output can be shared with all users. In Section VII-B, we will discuss different ways to instantiate such an auditor, and their pros and cons.

B. Completeness

For completeness of an unlearning protocol, we require that an honest execution of the protocol yields the expected outputs. Recall that an honest user does not add the same data point multiple times or requests to add a data point that was

deleted before. In such a situation, the notion of completeness states that users successfully verify the initialization of the model and the proofs for *all updates* performed by the server. Further, a proof of unlearning that was generated for an unlearned data point is also successfully verified by the user. In the following, we give a formal definition of computational completeness.

Definition 1 (Completeness). *Let λ be the security parameter. An unlearning protocol Φ is complete if for all $\text{pub} \leftarrow \text{Setup}(1^\lambda)$, the following properties are satisfied:*

1) *Let $(\text{st}_{S,0}, m_0, \text{com}_0, \rho_0) \leftarrow \text{Init}(\text{pub})$. Then*

$$\Pr[\text{VerifyNit}(\text{pub}, \text{com}_0, \rho_0) = 0] \leq \text{negl}(\lambda) .$$

2) *Let \mathcal{A} be a PPT adversary that outputs a valid sequence of datasets $(D_1^+, U_1^+), \dots, (D_\ell^+, U_\ell^+)$. For all $i \in [\ell]$ let $(\text{st}_{S,i}, m_i, \text{com}_i, \rho_i) \leftarrow \text{ProveUpdate}(\text{st}_{S,i-1}, \text{pub}, D_i^+, U_i^+)$. Then*

$$\Pr[\text{VerifyUpdate}(\text{pub}, \text{com}_{i-1}, \text{com}_i, \rho_i) = 0] \leq \text{negl}(\lambda) ,$$

where validity is defined with the following two conditions: $\forall i, j \in [\ell], i \neq j: D_i^+ \cap D_j^+ = \emptyset$ and $\forall i, j \in [\ell], j < i: D_i^+ \cap U_j^+ = \emptyset$.

3) *For all $i \in [\ell]$ and for all $d \in U_i^+$: Let $\pi_d \leftarrow \text{ProveUnlearn}(\text{pub}, \text{st}_{S,i}, d)$. Then*

$$\Pr[\text{VerifyUnlearn}(\text{pub}, d, \text{com}_i, \pi_{d_i}) = 0] \leq \text{negl}(\lambda) .$$

We require computational completeness here to allow for a wide range of instantiations. For example, an instantiation that works on hash values of data points cannot achieve perfect completeness because of hash collisions. By allowing for computational completeness, however, we only require that it should be hard for a PPT adversary to find such collisions (*i.e.*, with a negligible probability assuming collision-resistance).

Furthermore, we focus on exact unlearning; we require that one can *completely remove* a data point from the training set of an ML model. We made this choice because defining approximate unlearning remains an open problem [53], [62] and exact unlearning allows to give strong guarantees for verifiable unlearning. That said, our framework is generic and we expect that it can be instantiated with different exact unlearning techniques.

C. Security Definition

In this section, We present the security definition for unlearning in a game `GameUnlearn`. In this game, the adversary, described by a probabilistic algorithm \mathcal{A} , takes the role of the server. Intuitively, the definition captures that a malicious server cannot add (and train on) a data point that a user requested to delete in a previous iteration. However, we go a step further and let the server choose *which data points* it will add or delete. Thus, the goal of the server (*i.e.*, adversary) is to find a data point for which it can prove deletion, but which is re-added in some subsequent iteration.

GameUnlearn $_{\mathcal{A}, \mathcal{E}, \Phi, \mathcal{D}}(1^\lambda)$	
00	$\text{pub} \leftarrow \text{Setup}(1^\lambda)$
01	$(k, d, \pi_d, \{\text{com}_i, \rho_i\}_{i \in [0, \ell]}; \{D_i\}_{i \in [0, \ell]}) \leftarrow (\mathcal{A} \parallel \mathcal{E})(\text{pub}, \text{aux})$
02	
03	# Pre-processing
04	$U_k^+ := D_{k-1} \setminus D_k$
05	Parse com_i as $(\text{com}_i^m \parallel \text{com}_i^D) \forall i \in [0, \ell]$
06	
07	# Evaluate winning condition
08	if $\text{Commit}(\text{pub}, D_i) \neq \text{com}_i^D \forall i \in [0, \ell]$ # Commitments valid
09	and $\text{VerifyNit}(\text{pub}, \text{com}_0, \rho_0)$ # Initialization
10	and $\text{VerifyUpdate}(\text{pub}, \text{com}_{i-1}, \text{com}_i, \rho_i) \forall i \in [\ell]$ # Proof of update
11	and $\text{VerifyUnlearn}(\text{pub}, d, \text{com}_k, \pi_d)$ # Proof of unlearning
12	and $k < \ell$ and $d \in U_k^+$ and $d \in D_\ell$: # Point unlearned & re-added
13	return 1
14	return 0

Fig. 2: Security Game. We define the security of an unlearning protocol Φ in terms of game `GameUnlearn`. The notation $(\mathcal{A} \parallel \mathcal{E})$ denotes that both algorithms are run on the same input and random coins and assigning their results to variables before resp. after the semicolon. Input `aux` refers to auxiliary input.

In order to capture such a strong setting, we propose an *extractability-based* security definition as used in the context of hash functions or SNARKs [14], [24], [33]. That is, we require the existence of an extractor. While the adversary in our game has to provide the outputs, *i.e.*, the commitments and proofs, the extractor is able to produce the corresponding inputs that the adversary used—the underlying datasets in our case. We give a formal description of game `GameUnlearn` in Figure 2, which is divided into the following two stages:

S1. Simulation. The game draws the public parameters `pub` using `Setup` and runs the adversary \mathcal{A} on input `pub`. The extractor \mathcal{E} is run on the same input and random coins. Additionally, we provide benign auxiliary input `aux`, which captures any extra information that the adversary may have (possibly obtained prior to the start of executing the current protocol). At some point, \mathcal{A} will terminate and output a sequence of tuples $(k, d, \pi_d, \{\text{com}_i, \rho_i\}_{i \in [0, \ell]})$ for some $\ell \in \mathbb{N}$, where d is a point that was proved to be deleted in the k -th iteration. At the same time, the extractor outputs a sequence of datasets (D_0, \dots, D_ℓ) .

S2. Finalize. Before evaluating the winning condition, the game computes the set of data points unlearned in the k -th iteration based on the datasets D_k and D_{k-1} . Recall that the commitment in the framework consists of two parts com_i^m and com_i^D , where we need the second part for verification. Then the game checks for the following conditions: (1) com_i^D was obtained from D_i , (2) the initial proof ρ_0 verifies for the initial commitment com_0 , (3) each proof of update ρ_i verifies for commitments com_{i-1} and com_i , (4) the proof of unlearning π_d verifies for d and com_k , (5) $k < \ell$ and d was unlearned in iteration k and re-added in iteration ℓ . If all these properties are satisfied, then the game outputs 1 and \mathcal{A} wins.

We summarize this in the following definition.

Definition 2 (Unlearning). Let λ be the security parameter and consider game GameUnlearn in Figure 2. We say that protocol Φ for data distribution \mathcal{D} is unlearning-secure if for all PPT adversaries \mathcal{A} there exists an extractor \mathcal{E} such that for all benign auxiliary inputs aux :

$$\Pr[\text{GameUnlearn}_{\mathcal{A}, \mathcal{E}, \Phi, \mathcal{D}}(1^\lambda) \Rightarrow 1] \leq \text{negl}(\lambda).$$

V. INSTANTIATION

To instantiate a protocol in our framework, we first need to define the unlearning technique used. This can be any technique, but we require that it can entirely remove a data point after training (*i.e.*, satisfies exact unlearning). This is important to define a strong definition of security as discussed in Section IV-B. Furthermore, we need to define the procedures from Section IV-A and show that the resulting protocol satisfies both the completeness (Section IV-B) and security definition (Section IV-C).

In the following, we describe our instantiation. A formal description of the protocol is given in Figure 7 in Appendix A. For unlearning, we choose retraining-based unlearning as the canonical approach for unlearning and a key component of many other approaches. Here, the server retrains a model without using a data point d that needs to be unlearned. The protocol is further based on two main primitives, a SNARK Π and a hash function Hash (cf. Section II-B).

Data Representation. We internally split the data into training data D and unlearned data U . The server stores the training data D as a set of raw data points. As a practical privacy consideration, unlearned data points are only stored in hashed form. Although the server may always keep unlearned data points in clear, this is not and should not be necessary to perform the proof of unlearning.

Providing proof that the data was indeed deleted (as considered in *e.g.*, [51]) is an interesting problem itself, but is out of scope for this framework. In our protocol, the server stores an ordered set \mathcal{H}_U of hashed unlearned data points. The ordering of points is needed to maintain the Merkle tree structure described below. To improve efficiency, the server stores the training data points together with an ordered set \mathcal{H}_D of their hashed values. To account for the partition of training and unlearned data, we instantiate the commitment com as a triplet: (a) hash of the model h_m , (b) hash of the training data h_D , and (c) hash of the unlearned data h_U .

Models are hashed with algorithm HashModel as a hash of their individual weights. To hash the training data, we use the root of a Merkle tree computed with algorithm HashData on D . For unlearned data points, we use an append-only Merkle tree computed with algorithm HashUnlearn and use its root as the hash of unlearned data U . Once a data point is unlearned, the append-only property of the Merkle tree ensures that it cannot be part of the training data again. Further, Merkle trees allow for fast membership verification and efficient caching of intermediate hashes (*i.e.*, the roots of its subtrees). We present an example of both data structures in Figure 5 and a formal description of the hashing algorithms in Appendix A.

```

ProveModelCircuit(public  $h_m, h_D$ , private  $D$ )
00 # Step 1: commit to the dataset
01  $h'_D := \text{HashData}(\{\text{HashDataPoint}(d)\}_{d \in D})$ 
02 if  $h'_D \neq h_D$ :
03   return false
04 # Step 2: verify model
05  $m := \text{TrainModel}(D)$ 
06 if  $h_m \neq \text{HashModel}(m)$ :
07   return false
08 return true

```

Fig. 3: **Circuit C_m for Proof of Update.** First, the commitment on the dataset is verified. Subsequently, the model is retrained and verified against the provided one.

1. Initialization. During the protocol's initialization, training data and sets of hash values are initialized as empty sets. As retraining-based machine unlearning does not depend on any prior models, model m_0 and proof ρ_0 are not used. Therefore, we set $m_0 = \rho_0 = \text{empty}$.

2. Proof of Update. The proof of update consists of two procedures: ProveUpdate and VerifyUpdate . The server starts by executing ProveUpdate . In the i -th iteration, this procedure updates the training data D_i and the sets of hashed data points \mathcal{H}_{D_i} and \mathcal{H}_{U_i} with any data points requested to be added or deleted (respectively) in this iteration. Using the updated training data, the server then trains a new model m_i and commits to both this model and the updated dataset.

Proof ρ_i is used to attest that the (a) model, and (b) dataset were updated correctly. We instantiate ρ_i with two individual proofs using a cryptographic proof system. Recall that a proof system describes an interactive protocol between a *prover* (*i.e.*, server) and a *verifier* (*i.e.*, users), where the prover wants to convince the verifier that some statement for a given polynomial time decidable relation R is true. In general, the prover holds a witness ω for the statement. In our instantiation we use a SNARK Π (cf. Section II-B), which allows a prover to *non-interactively* prove a statement with a short (or succinct) cryptographic proof.

The two relations for which we want to prove statements can be algorithmically described by circuits ProveModelCircuit (for model updates) and ProveDataCircuit (for dataset updates) given in Figures 3 and 4. Those circuits have public and private inputs (statement and witness) and return **true** if the input is in the relation. We explain their internal routines below. In order to use these circuits in our instantiation however, we need to encode them as statements over a verification procedure. In particular, a statement is valid if the verification procedure is successful. We define relations R_m and R_D , encoding ProveModelCircuit and ProveDataCircuit , such that given the inputs to the corresponding circuit, one can efficiently compute a statement ϕ and a witness ω for the verification procedure captured by R_m or R_D respectively. We give a detailed overview in Section VI.

We will now describe our instantiation for algorithm ProveUpdate and which statement the server needs to prove.

```

ProveDataCircuit(public  $h_D, h_{U_{prev}}, h_U$ , private  $\mathcal{H}_D, \mathcal{H}_{U_{prev}}, \mathcal{H}_{U^+}$ )
00 # Step 1: verify commitments
01 if  $h_D \neq \text{HashData}(\mathcal{H}_D)$  or
     $h_{U_{prev}} \neq \text{HashUnlearn}(\mathcal{H}_{U_{prev}})$ :
02   return false
03 # Step 2: verify unlearned data
04  $\mathcal{H}_U := \mathcal{H}_{U_{prev}} \cup \mathcal{H}_{U^+}$ 
05 if  $h_U \neq \text{HashUnlearn}(\mathcal{H}_U)$ :
06   return false
07 # Step 3: verify intersection
08 if  $\mathcal{H}_D \cap \mathcal{H}_U \neq \emptyset$ :
09   return false
10 return true

```

Fig. 4: **Circuit C_D for Proof of Update.** First, the commitment on the dataset is verified. Second, the update of the unlearned data is verified. Finally, it is verified that the training data does not contain any unlearned data points.

- **Model:** For the model update, the server proves that model m_i was trained on D_i . This is captured in procedure `ProveModelCircuit` which is depicted in Figure 3. The server computes and outputs a proof π_{m_i} .
- **Dataset:** To prove and verify the updates on training data D_i and unlearned data U_i , we define `ProveDataCircuit` in Figure 4. The server computes the proof π_{D_i} using its knowledge about \mathcal{H}_{D_i} , $\mathcal{H}_{U_{i-1}}$, and $\mathcal{H}_{U_i^+}$.

Both proofs are computed as SNARK proofs using the encoding described above. Subsequently, the users execute `VerifyUpdate` and verify both SNARK proofs using the hash values in the commitment and the previous commitment. In particular, h_{m_i} and h_{D_i} are used to verify the correctness of π_{m_i} with regard to R_m (i.e., if m_i was trained on data D_i). Then h_{D_i} , h_{U_i} and $h_{U_{i-1}}$ from the previous commitment together with π_{D_i} are used to check that the training data D_i does not contain any unlearned points, i.e., $D_i \cap U_i = \emptyset$. Note that by construction, it further holds that the previous set of unlearned data points is a subset of the updated set $U_{i-1} \subseteq U_i$, which is also proved inside π_{D_i} .

3. Proof of Unlearning. We prove unlearning of data point d by proving its membership in U_i . Since $U_i \cap D_i = \emptyset$ it follows that $d \notin D_i$. Specifically, we use the second Merkle tree for U_i . This is computed with algorithm `HashUnlearn`. For a given data point $d \in U_i$, we compute a membership proof as a path in this tree. This path can be verified against the tree root (i.e., the hash h_{U_i}). Figures 5 shows an example of this.

In the protocol, the server performs `ProveUnlearn` and proves membership in the set \mathcal{H}_{U_i} . For this, the server computes the tree path with `ComputeTreePath` to the data point of which we want to prove membership and outputs this as the proof which is then sent to the user. The user first uses the tree root h_{U_i} —fetched from the commitment—to verify membership with `VerifyTreePath`. If the path leads to the root, the user will accept. Otherwise, it will abort.

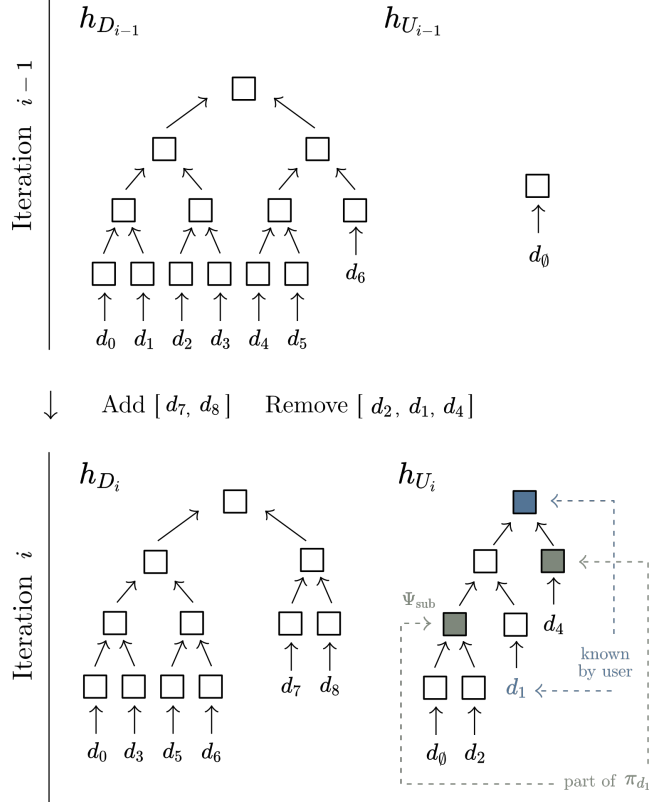


Fig. 5: **Merkle Trees.** We use two Merkle trees in our instantiation: (a) a tree for training data D (left) that is used for the proof of update and (b) a tree for the unlearned data U (right) used for the proof of unlearning. The top shows the Merkle trees in iteration $i-1$. In iteration i data points d_7 and d_8 are added and d_2, d_1, d_4 sequentially unlearned. Suppose, we want to create a proof of unlearning for d_1 , i.e., $d_1 \notin D_i$. We do so by proving membership in U_i . This proof consists of the tree path $\pi_{d_1} = [\Psi_{\text{sub}}, h_{d_4}]$, where Ψ_{sub} denotes the root of the subtree below d_1 . To verify this proof, the user computes the path to the root and checks $h_{U_i} \stackrel{!}{=} \text{Hash}(\text{Hash}(\Psi_{\text{sub}}, \text{Hash}(d_1), h_{d_4}))$.

A. Completeness and Security

We first show that our instantiation is complete according to Definition 1.

Theorem 1. *Let Π be a complete SNARK and `Hash` a collision-resistant hash function. Then the instantiated protocol in Figure 7 satisfies completeness.*

We give a proof sketch below and refer to Appendix B for the full proof.

Proof (Sketch). Completeness of the initialization is easy to observe since all values are initialized as empty and verified in the same way. The second property follows from the completeness of the SNARK and collision-resistance of the hash function, which is why we require computational completeness. If a collision occurs, it is not possible to provide the

proof of update. Given that the proof of update is successful, completeness of the proof of unlearning follows from the construction and correctness of the Merkle tree. \square

Now we want to prove that our instantiation is a *secure* unlearning protocol according to Definition 2.

Theorem 2. *Let Hash be a collision-resistant hash function and Π be a secure SNARK. Then the instantiated protocol in Figure 7 satisfies unlearning security.*

We give a proof sketch below and refer to Appendix C for the full proof.

Proof (Sketch). Let \mathcal{A} be an adversary in the unlearning security game (cf. Figure 2). By knowledge soundness of the SNARK, there exists an extractor which outputs the witness and thus the datasets D_i corresponding to the outputs of the adversary. We then use the fact that the first SNARK proof π_{m_i} verifies successfully. By soundness, this means that \mathcal{A} computed the proof using the witness, *i.e.*, the model m_i and the dataset D_i , which correspond to the hash values in the commitment. By collision-resistance of the hash function, the adversary cannot find another dataset or model for the same commitment and thus, training a model on the dataset provided by \mathcal{A} coincides with the witness used to create π_{m_i} .

Since all proofs π_{D_i} as well as the proof of unlearning of data point d must verify successfully, the hash of d must be contained in the set \mathcal{H}_{U_k} which was used to create π_{D_k} . Here, k is the iteration where d was unlearned; the observation holds by assuming soundness of the SNARK and collision-resistance of the hash function. We can further infer that d must also be part of future sets \mathcal{H}_{U_i} , $k < i \leq \ell$ and by collision-resistance d must also be part of the underlying datasets U_i . Finally, we use the fact that the proof π_{D_i} attests that the intersection of \mathcal{H}_{U_i} and \mathcal{H}_{D_i} is empty. This yields a contradiction and shows that d cannot be present in the last dataset D_ℓ . \square

VI. EVALUATION

In this section, we evaluate the instantiated protocol. First, we implement the protocol’s main building blocks for the proof of update and the proof of unlearning. To understand the time complexity of each of these building blocks, we micro-benchmark the underlying algorithms. Second, we focus on the proof of update as it is the dominant factor of the protocols’ run-time. We study its applicability to different classes of ML models and real-world datasets.

All experiments are performed on a server running Ubuntu 22.04 with 256 GB RAM and two Intel Xeon Gold 5320 CPUs. Our code is available at <https://github.com/cleverhans-lab/verifiable-unlearning>.

A. Experimental Setup

We describe our implementation, and highlight salient features of the components (datasets and models) used.

Proof System. To implement the instantiated protocol, we build upon the SNARK construction from Groth16 [34]. This construction requires pairing-friendly elliptic curves and we

use the Barreto-Naehrig curve `ALTBN_128` in our implementation. Groth16—as with most SNARKs in practice—requires a trusted setup [49], which we assume is executed during the global setup of the protocol. We instantiate the hash function Hash with the *Pedersen Hash* [4, p.76].

For the proof system used in the proof of update, we define the verification of the model and dataset updates in terms of a polynomial decidable binary relation R_m and R_D (respectively) over circuits `ProveModelCircuit` and `ProveDataCircuit` (respectively) as outlined in Figures 3 and 4 (respectively). The functionality of `ProveModelCircuit` is two-fold: it is to ensure that (a) a given hash h_D was computed correctly from a dataset D , and (b) a given hash h_m was computed on the model m that was obtained from the dataset corresponding to hash value h_D . Access to the private training data D is only needed for proof creation, and verification is possible with only the commitments h_m and h_D . Similarly, the functionality of `ProveDataCircuit` ensures that the Merkle tree roots of the hashed data points and unlearned data points are obtained and consistent to the previous Merkle tree root of unlearned data points. Here public verification can be performed with the Merkle tree roots only, whereas the prover uses the corresponding sets of hashed data points and unlearned data points.

Following prior work [5], [49], [68], we convert the computation on these circuits into *Rank-1 Constraint Systems* (R1CS) instances; *i.e.*, statements in R_m or R_D are represented as a constraint system over a *finite field*. We point the readers to the work of Angel *et al.* [5] which contains an excellent overview of the advantages of using R1CS. We use the ZoKrates compiler [22] as a frontend compiler to facilitate the conversion to R1CS. ZoKrates supports multiple proving schemes and provides an extensive (and well-documented) standard library to build circuits. The compiler ensures that the computation graph does not have loops, and a “flat” computation is performed. However, enabling the desired functionality requires overcoming several other challenges:

- C1. Real Numbers:** Often data and other parameters are represented as floating point numbers. However, real numbers cannot directly be represented in a finite field and, thus, we need to define some form of encoding. To this end, we convert inputs into *fixed precision* real numbers by scaling each number by the constant $\gamma = 10^5$. Addition and multiplication then follow by the following rules $\gamma(a + b) := \gamma a + \gamma b$ and $\gamma(a \cdot b) := \gamma^{-1}(\gamma a \cdot \gamma b)$.
- C2. Non-Linear Activation Functions:** Training ML models with non-linear activation functions are demonstrably more performant. However, encoding such functions as R1CS constraints is highly inefficient [5]. Consequently, they need to be replaced with linear approximations. In our implementation, the sigmoid activation function is replaced with a third-order polynomial. In this approximation, we use ordinary least squares regression with polynomial scaled input features [39], [40].

TABLE I: **Datasets.** Binary classification datasets from the PMLB benchmark suite [52] that are used for the experiments. We report the number of data points, number of features together with their type.

Dataset	Data Points	Features	Feature Type		
			Cat.	Bool	Num.
creditscore	80	7	-	3	4
postoperative patient	70	9	6	2	1
cyyoung 9302	73	11	-	3	8
corral	128	7	-	7	-

Cat.: Categorical, Bool: Boolean, Num.: Real number

C3. Stochasticity: Training ML models often requires stochasticity at various stages (such as random data batch access, or weight initialization etc.). We hard-code any randomness needed for the training directly into the circuit itself (*e.g.*, we hard-code the random initialized values for the model parameters in the source of the circuit) as done in prior work [5].

Models Considered. We evaluate our proposal on binary classification tasks, trained using 4 models: (a) linear regression, (b) logistic regression, (c) feed-forward neural network (NN) with 2 hidden neurons, and (d) feed-forward NN (with 4 hidden neurons). The NNs have one hidden layer and a single output neuron. We use the sigmoid function (which we modify as described above) as the activation.

Datasets. We choose several real-world datasets from the PMLB benchmark suite [52] as considered in prior work [5]. These datasets encompass diversity in (a) type of features, (b) number of features, and (c) number of data points (the latter two of which influence the total number of features being processed in C_m). Salient features of the datasets we considered are in Table I.

B. Protocol Instantiation

We first implement the high-level functions of the instantiated protocol (from Section V). We consider the sub-tasks of proof of update with model and dataset updates (respectively), and the proof of unlearning.

In the first experiment we want to understand the overheads of each of the sub-tasks. To this end, we consider a simple linear regression model and train this model for a single epoch with SGD as a general purpose approach. We set the batch size to 1. For this experiment, we use a synthetic dataset D , and set $|D| \in \{1, 10, 100, 1000\}$; each data point has a single feature. Furthermore, we assume that 10% of data points were unlearned, *i.e.*, $|U| \in \{1, 10, 100\}$ (as appropriate). We consider a single iteration and assume that no data points are requested to be added and a single point d is requested to be deleted (*i.e.*, $D^+ = \emptyset$ and $U^+ = \{d\}$). To simplify the experimental setup, we assume that d is added directly to U (*i.e.*, $d \notin D$).

The results from these experiments are presented in Table II. Proof creation between sub-tasks ranges between 29s for

TABLE II: **Run-Time of Protocol Functions.** We compare the running time between the protocols subtasks. We report the numbers for a linear regression model trained with SGD with varying numbers of data points. We set $|D| \in \{10, 100, 1000\}$ and $|U| \in \{1, 10, 100\}$ (as appropriate). We assume that a single data point is requested for deletion. The running time of Π .Setup refers to the cumulative running time of creating relation R_m (respectively R_D) and setting up Π .

	Data Points $ D / U $		
	10 / 1	100 / 10	1000 / 100
<i>Proof of Update (Model Update)</i>			
Π .Setup w/ R_m	47s	7m 17s	1h 38m 23s
Π .Prove w/ R_m	14s	1m 54s	34m 28s
Π .Vrfy w/ R_m	1s	1s	1s
<i>Proof of Update (Dataset Update)</i>			
Π .Setup w/ R_D	12s	1m 52s	50m 56s
Π .Prove w/ R_D	4s	34s	15m 18s
Π .Vrfy w/ R_D	1s	1s	1s
<i>Proof of Unlearning</i>			
ComputeTreePath	1s	4s	29s
VerifyTreePath	1s	1s	1s

the proof of unlearning, 16m for proving the dataset update, and 35m for proving model update; each for 1000 training and 100 unlearned data points. Notably, all verification algorithms run in constant time independent of the number of data points. Overall, proving a model update is computationally the most expensive sub-task. The largest share is due to the cost of setting up the proof system. This is expected: Groth16 [34] amortizes the setup costs with highly efficient proof creation and constant-time verification. Note that this is a one-time effort, and the cost can be amortized over multiple iterations of unlearning requests. In case of the unlearning protocol, it is therefore often beneficial to trade-off an expensive setup with fast proving and verification.

C. Proof of Update

The dominant component of the protocols' run-time is the proof of model updates. Its run-time itself depends on the complexity of circuit C_m . We can measure this complexity as a function of the number of RICS constraints. In case of a model update, this number is determined by two factors:

- **Model Complexity.** A model is naturally represented by its parameters, which needs to be encoded in the circuit. Thus, larger the model, more the constraints required for this encoding. Furthermore, representing the training itself requires more constraints as we need to encode the update during training for every parameter.
- **Dataset.** Similar to the model, we also need to encode the data points as constraints in the circuit. Again, with increasing dataset size, we need more constraints. This applies both in terms of number of points *and* number of features per point.

TABLE III: **Proving Time vs. Model Capacity.** We compare the proving time for different classes of models with increasing capacity. All are trained with SGD on the same dataset of 100 points. The running time of Π .Setup refers to the cumulative running time of creating relation R_m and setting up Π .

	RICS	Π .Setup	Π .Prove	Π .Vrfy
Linear Regression	3,850,613	7m 16s	1m 55s	1s
Logistic Regression	7,043,435	11m 27s	3m 17s	1s
Neural Network ($N = 2$)	26,501,432	36m 51s	13m 7s	1s
Neural Network ($N = 4$)	48,018,155	1h 7m 9s	25m 47s	1s

RICS: Number of RICS constraints

In the following, we first consider model complexity and study different classes of models. Next, we focus on the complexity of the dataset.

1) *Model Complexity*: To understand the effects of the choice of ML model, we consider linear regression, logistic regression and neural networks for classification. For the neural networks, we focus on models with one hidden layer and varying numbers of (hidden) neurons $N \in \{2, 4\}$. Again, we train each model with SGD for a single epoch on a synthetic dataset with 100 (single feature) training points.

The results are summarized in Table III. We observe that the circuit size increases together with the complexity of the model. For instance, the number of RICS constraints increases by $1.8\times$ to 7,043,437 constraints when going from logistic to linear regression. This is intuitive: in logistic regression, we additionally need to evaluate the sigmoid activation which induces this overhead. In a similar vein, moving from logistic regression to neural networks increases the circuit further to 26,501,439 ($N = 2$) and 48,018,168 ($N = 4$) constraints respectively.

Setup costs per RICS constraint are between $83\mu\text{s}$ – $113\mu\text{s}$. Proof creation is more efficient and costs per constraint range between $28\mu\text{s}$ – $31\mu\text{s}$. Verification (independent of the circuit size) runs in constant time.

2) *Real-World Datasets*: To understand the impact of the dataset and the practical applicability of the protocol, we now turn to real-world datasets as listed in Table I. To make results comparable, we train all models for 10 epochs with a learning rate of 0.1. We split the data into 80:20 train test split. Models achieve a test accuracy between 59% and 95%. Results are presented in Figure 6.

For all models, we observe a linear dependence between run-time and dataset size. As before, the setup costs are much higher than computing the proof. This cost, for a dataset with 896 total features (*i.e.*, total points \times features), ranges between 13m for linear regression and 1h 25m for the neural network with 4 hidden neurons. From this, 3m are spent on Step 1 in Figure 3 (*i.e.*, verification of the data commitment); *i.e.*, most time is spent on proving the computation itself.

VII. DISCUSSION

In the previous sections, we presented a formal framework for verifying machine unlearning. We discuss the limitations of our constructions, along with potential improvements.

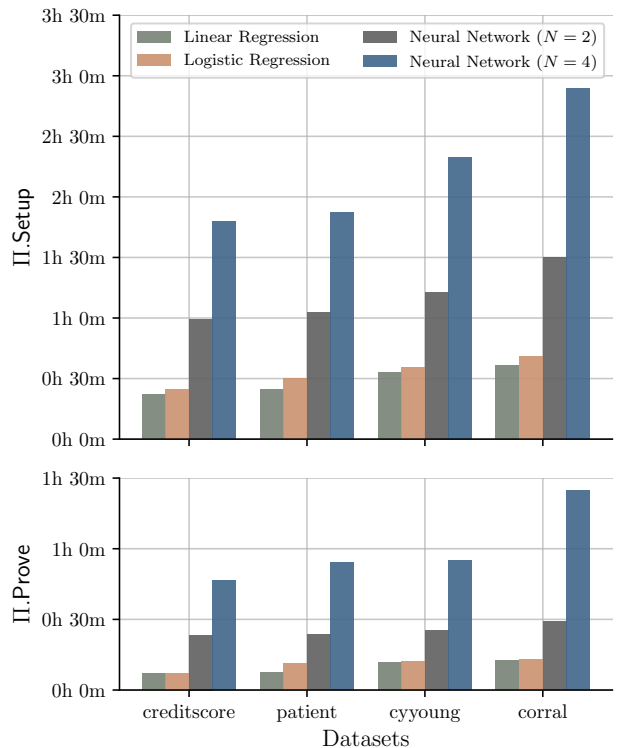


Fig. 6: **Scalability to Real-World Datasets.** We simulate the proof of update for different datasets and four different models: linear regression, logistic regression, neural networks with $N = 2$ and $N = 4$ hidden neurons (from left to right). All models are trained for 10 epochs with a learning rate of 0.1 achieve a test accuracy between 59% and 95%. The running time of Π .Setup refers to the cumulative running time of creating relation R_m and setting up Π .

A. Scalability

Our experiments with the instantiated system show that the run-time of the protocol is dominated by the proof of update. We base our construction on *Verified Computation* (VC) and, consequently, inherit its limitations. The limited scalability we observe in our experiments is comparable to similar problems which have been approached with VC [59], [38], [39], [40]. Nevertheless, we also discuss how we can improve performance with the primitives available today. In general, the run-time depends on the size of the verification circuit. In the following, we discuss two directions on how this can be reduced.

Sharded, Isolated, Sliced, and Aggregated (SISA) training [15]. It is a technique to improve the performance of exact machine unlearning. The basic idea is to split the training data into non-overlapping shards, train a separate model on each shard and aggregate results. To further reduce computational costs, each shard can be divided into slices and training is done iteratively with increasing number of slices. For unlearning, this allows to begin training from an intermediate checkpoint (*i.e.*, before the slice with the unlearned

data points was introduced).

In our setting, SISA training can also be leveraged to reduce the computational cost of proof generation. As shards only contain a subset of data, the verification circuit can be much smaller; the setup needs only be executed once and parameters can be shared across all shards. As shards are independent of each other, and verification can be parallelized.

Chaining. Sharding reduces the size of the circuit by reducing the number of training points. Similarly, we can also reduce the number of epochs in the circuit. In particular, we can define the circuit to verify only a single epoch and chain together multiple executions of the same circuit. Moreover, each epoch can be verified independently — allowing further verification parallelization.

B. Alternative Instantiations

Our instantiation in Section V avoids having a trusted third party and instead relies only on cryptographic protocols to guarantee security. For efficiency purposes and to remove the burden from the user, one can introduce a trusted auditor who verifies on behalf of a user (as we discuss towards the end of Section IV-A). The trust in auditing can be achieved either by having a dedicated trusted third party (*e.g.*, one that does not have a motivation to collude with the server such as another cloud provider), or a set of distributed auditors where trust is established from multiple independent verifications, or a trusted execution environment (TEE).

If TEEs (*e.g.*, Intel SGX [45]) are available, then one can run the whole procedure within it and return a digest signed by a TEE provider to the user. When using a TEE, one needs to consider common concerns such as trusting a hardware vendor, availability of said vendor for signing the digest, limited memory [35], their applicability to ML-related tasks that involve GPU computation [65], and side-channels [67]. Though there is a significant amount of ongoing work on addressing these concerns [21], [47], [64], addressing them when using TEEs for verifying unlearning is future work.

C. Practical Implementation Concerns

Our construction is based on assumptions which might not always hold in practice. Next, we discuss how these assumptions could be relaxed.

User Authentication. The current framework assumes that users execute the protocol faithfully. For example, a user could impersonate another user and request the deletion of their data. To avoid this, data addition/deletion requests can be signed by the user and verified by the server.

Data Authentication. In a similar vein, the server can re-add deleted data points by impersonating a user and duplicate points. To account for this, we need a stronger form of authentication and, more importantly, require that users *own* their data. This can be achieved by signing data points. These signatures need to be verified within the proof of update.

Reusing of Old Models. Broadly speaking, there are two reasons for a server to not faithfully unlearn: (a) drop in utility

when removing data points, and (b) to save the computational cost of updating or re-training a model. In its current form, the framework only prevents (b) as it guarantees that the server needs to update the model when data is deleted. To address (a), the inference could be tied to a particular model. This is an open problem, which is interesting to tackle in itself outside the context of unlearning.

Confidentiality. Our instantiation does not require the users to know the datasets or model. In fact, they only see hash commitments and the SNARK proofs. In order to give formal confidentiality guarantees, one would need a pseudo-random hash function. By modelling the hash function as a random oracle, hash values do not leak any information about the underlying data points. For the SNARK proofs, we additionally need the zero-knowledge property [30] (which Groth16 satisfies) so that the proof does not leak information about the witness (which is computed using the dataset).

VIII. RELATED WORK

Our approach for provably secure machine unlearning touches different areas of security and ML research. In the following, we examine related concepts and methods.

Verifiable Computation. Proof systems are core components of schemes that are used to enable VC. In this paradigm, delegated computation (such as code executing in a remote cloud server) is verified for correctness. A simple mechanism to achieve this is by re-executing the computation. However, this is prohibitively expensive in many settings. For example, approaches based on probabilistically checkable proofs [6] were deemed expensive. A series of research by Setty *et al.* (and collaborators) [57], [58], [16], [56], [66], [55], [43], Ben-Sasson *et al.* (and collaborators) [10], [11], [13], [8], [12], [9], and others [27], [50], [25] have made remarkable progress in making these schemes (and those related to verification of data used for computation) practical.

SNARK Creation. CirC [48] is a compiler infrastructure build on an intermediate representation for systems of arithmetic constraints (such as proof systems). xjSnark [42] aims at minimizing the circuit size to reduce redundant computation across multiple executions, making the compiling process user friendly. Oti [5] is a compiler that is aimed at designing efficient arithmetic circuits for problems that involve optimization (such as those commonly found in ML). DIZK [68] is a distributed system capable of distributing the compute required for proof creation.

Proof of Training. The work of Zhao *et al.* [70] is most similar to ours. However, their primary objective is to design a scheme for verifying computation to ensure that the payments made to servers are correct. In our work, however, we aim to design a scheme to verify the correctness of data deletion when training ML models.

IX. CONCLUSION

Unlike prior work that focused on defining unlearning from the server’s perspective, we took a user-centric approach in this paper. Our framework leverages cryptographic primitives to have the server prove to the user that their data has been unlearned. Specifically, we instantiated our proposed framework through an approach that combines verified computing for training with a proof of unlearning. Future work may further improve this initial solution to leverage scalability advances in verified computing but also propose solutions that authenticate the user and data involved in unlearning protocols.

ACKNOWLEDGEMENTS

Thorsten Eisenhofer and Doreen Riepel were funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany’s Excellence Strategy - EXC 2092 CASA - 390781972.

REFERENCES

- [1] General Data Protection Regulation (GDPR). , 2016. Accessed: October 18, 2022.
- [2] California Consumer Privacy Act (CCPA). , 2018. Accessed: October 18, 2022.
- [3] Personal Information Protection and Electronic Documents Act. , 2019. Accessed: October 18, 2022.
- [4] Zcash Protocol Specification. Version 2022.3.4. , 2022.
- [5] Sebastian Angel, Andrew J. Blumberg, Eleftherios Ioannidis, and Jess Woods. Efficient representation of numerical optimization problems for snarks. In *USENIX Security Symposium (USENIX Security)*, 2022.
- [6] Sanjeev Arora and Shmuel Safra. Probabilistic checking of proofs: A new characterization of np. *Journal of the ACM (JACM)*, 45(1):70–122, 1998.
- [7] Thomas Baumhauer, Pascal Schöttle, and Matthias Zeppelzauer. Machine unlearning: Linear filtration for logit-based classifiers. *arXiv preprint arXiv:2002.02730*, 2020.
- [8] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Annual international cryptology conference*, pages 701–732. Springer, 2019.
- [9] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete efficiency of probabilistically-checkable proofs. In *Proceedings of the forty-fifth annual ACM symposium on Theory of Computing*, pages 585–594, 2013.
- [10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference*, pages 90–108. Springer, 2013.
- [11] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct {Non-Interactive} zero knowledge for a von neumann architecture. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 781–796, 2014.
- [12] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Short pcps verifiable in polylogarithmic time. In *20th Annual IEEE Conference on Computational Complexity (CCC’05)*, pages 120–134. IEEE, 2005.
- [13] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Robust pcps of proximity, shorter pcps, and applications to coding. *SIAM Journal on Computing*, 36(4):889–974, 2006.
- [14] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference, ITCS ’12*, page 326–349, New York, NY, USA, 2012. Association for Computing Machinery.
- [15] Lucas Bourtole, Varun Chandrasekaran, Christopher A Choquette-Choo, Hengrui Jia, Adelin Travers, Baiwu Zhang, David Lie, and Nicolas Papernot. Machine unlearning. *arXiv preprint arXiv:1912.03817*, 2019.
- [16] Benjamin Braun, Ariel J. Feldman, Zuo Cheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. Verifying computations with state. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2013.
- [17] Gavin Brown, Mark Bun, Vitaly Feldman, Adam Smith, and Kunal Talwar. When is memorization of irrelevant training data necessary for high-accuracy learning? In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 123–132, 2021.
- [18] Yinzhi Cao and Junfeng Yang. Towards making systems forget with machine unlearning. In *2015 IEEE Symposium on Security and Privacy*, pages 463–480. IEEE, 2015.
- [19] Nicholas Carlini, Chang Liu, Úlfar Erlingsson, Jernej Kos, and Dawn Song. The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 267–284, 2019.
- [20] Min Chen, Zhikun Zhang, Tianhao Wang, Michael Backes, Mathias Humbert, and Yang Zhang. Graph unlearning. *arXiv preprint arXiv:2103.14991*, 2021.
- [21] Victor Costan and Srinivas Devadas. Intel SGX explained. *Cryptology ePrint Archive*, 2016.
- [22] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018.
- [23] Vitaly Feldman. Does learning require memorization? a short tale about a long tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 954–959, 2020.
- [24] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, page 1304–1316, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *ACM SIGSAC Conference on Computer and Communications Security*, 2016.
- [26] Xiangshan Gao, Xingjun Ma, Jingyi Wang, Youcheng Sun, Bo Li, Shouling Ji, Peng Cheng, and Jiming Chen. Verifi: Towards verifiable federated unlearning. *arXiv preprint arXiv:2205.12709*, 2022.
- [27] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*, pages 465–482. Springer, 2010.
- [28] Amirata Ghorbani and James Zou. Data shapley: Equitable valuation of data for machine learning. In *International Conference on Machine Learning*, pages 2242–2251. PMLR, 2019.
- [29] Aditya Golatkar, Alessandro Achille, and Stefano Soatto. Eternal sunshine of the spotless net: Selective forgetting in deep networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9304–9312, 2020.
- [30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989.
- [31] Laura Graves, Vineel Nagisetty, and Vijay Ganesh. Amnesiac machine learning. *arXiv preprint arXiv:2010.10981*, 2020.
- [32] Laura Graves, Vineel Nagisetty, and Vijay Ganesh. Amnesiac machine learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, number 13, pages 11516–11524, 2021.
- [33] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.
- [34] Jens Groth. On the size of pairing-based non-interactive arguments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2016.
- [35] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure dnn inference with enclaves. *arXiv preprint arXiv:1810.00602*, 2018.
- [36] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *arXiv preprint arXiv:1708.06733*, 2017.

- [37] Chuan Guo, Tom Goldstein, Awni Hannun, and Laurens Van Der Maaten. Certified data removal from machine learning models. *arXiv preprint arXiv:1911.03030*, 2019.
- [38] Inbar Helbitz and Shai Avidan. Reducing relu count for privacy-preserving cnn speedup. *arXiv preprint arXiv:2101.11835*, 2021.
- [39] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2018.
- [40] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, Xiaoqian Jiang, et al. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e8805, 2018.
- [41] Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *International Conference on Machine Learning*, pages 1885–1894. PMLR, 2017.
- [42] Ahmed E. Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *IEEE Symposium on Security and Privacy SP*, 2018.
- [43] Jonathan Lee, Kirill Nikitin, and Srinath Setty. Replicated state machines without replicated execution. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 119–134. IEEE, 2020.
- [44] Klas Leino and Matt Fredrikson. Stolen memories: Leveraging model memorization for calibrated {White-Box} membership inference. In *29th USENIX security symposium (USENIX Security 20)*, pages 1605–1622, 2020.
- [45] Frank McKeen, Ilya Alexandrovich, Alex Berenson, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. Association for Computing Machinery.
- [46] Seth Neel, Aaron Roth, and Saeed Sharifi-Malvajardi. Descent-to-delete: Gradient-based methods for machine unlearning. In *Algorithmic Learning Theory*, pages 931–962. PMLR, 2021.
- [47] Olga Ohrimenko, Felix Schuster, Cedric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 619–636, Austin, TX, August 2016. USENIX Association.
- [48] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *43rd IEEE Symposium on Security and Privacy, SP*, 2022.
- [49] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In Srđjan Capkun and Franziska Roesner, editors, *USENIX Security Symposium (USENIX)*, 2020.
- [50] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE Symposium on Security and Privacy (SP)*, 2013.
- [51] Daniele Perito and Gene Tsudik. Secure code update for embedded devices via proofs of secure erasure. In Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou, editors, *Computer Security – ESORICS 2010*, pages 643–662, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [52] Joseph D Romano, Trang T Le, William La Cava, John T Gregg, Daniel J Goldberg, Praneel Chakraborty, Natasha L Ray, Daniel Himmelstein, Weixuan Fu, and Jason H Moore. Pmlb v1.0: an open source dataset collection for benchmarking machine learning methods. *arXiv preprint arXiv:2012.00058v2*, 2021.
- [53] Ayush Sekhari, Jayadev Acharya, Gautam Kamath, and Ananda Theertha Suresh. Remember what you want to forget: Algorithms for machine unlearning. *arXiv preprint arXiv:2103.03279*, 2021.
- [54] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. *arXiv preprint arXiv:1708.00489*, 2017.
- [55] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 339–356, 2018.
- [56] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. Taking {Proof-Based} verified computation a few steps closer to practicality. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 253–268, 2012.
- [57] Srinath T. V. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In Daniele Micciancio and Thomas Ristenpart, editors, *Annual International Cryptology Conference CRYPTO*, 2020.
- [58] Srinath TV Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, number 9, page 17, 2012.
- [59] Avital Shafraan, Gil Segev, Shmuel Peleg, and Yedid Hoshen. Crypto-oriented neural architecture design. In *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2680–2684. IEEE, 2021.
- [60] Iliia Shumailov, Zakhar Shumaylov, Dmitry Kazhdan, Yiren Zhao, Nicolas Papernot, Murat A. Erdogdu, and Ross J. Anderson. Manipulating SGD with data ordering attacks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.
- [61] David Marco Sommer, Liwei Song, Sameer Wagh, and Prateek Mittal. Towards probabilistic verification of machine unlearning. *arXiv preprint arXiv:2003.04247*, 2020.
- [62] Anvith Thudi, Gabriel Deza, Varun Chandrasekaran, and Nicolas Papernot. Unrolling sgd: Understanding factors influencing machine unlearning. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 303–319. IEEE, 2022.
- [63] Anvith Thudi, Hengrui Jia, Iliia Shumailov, and Nicolas Papernot. On the necessity of auditable algorithmic definitions for machine unlearning. In *USENIX Security Symposium (USENIX Security)*, 2022.
- [64] Florian Tramèr and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.
- [65] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on {GPUs}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, 2018.
- [66] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 223–237. IEEE, 2013.
- [67] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2421–2434, 2017.
- [68] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. DIZK: A distributed zero knowledge proof system. In William Enck and Adrienne Porter Felt, editors, *USENIX Security Symposium, (USENIX)*, 2018.
- [69] Yinjun Wu, Edgar Dobriban, and Susan Davidson. Deltagrad: Rapid retraining of machine learning models. In *International Conference on Machine Learning*, pages 10355–10366. PMLR, 2020.
- [70] Lingchen Zhao, Qian Wang, Cong Wang, Qi Li, Chao Shen, and Bo Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Trans. Parallel Distributed Syst.*, 2021.

A. Our Instantiated Protocol

A schematic overview of our complete instantiated protocol is given in Figure 7. Additional algorithms are described below.

```

HashData( $\mathcal{H}_D$ )
00 # hash leafs
01 leafs := [ ]
02 for  $h_d \in \mathcal{H}_D$ :
03   leafs.append( $h_d$ )
04 # build tree
05 current_level := leafs
06 repeat:
07   prev_level := current_level
08   current_level := [ ]
09   # hash nodes from previous level
10   for lhs, rhs in zip(prev_level[:2],
11                       prev_level[1:2]):
12     current_level.append(Hash(lhs, rhs))
13   # add remaining node (if necessary)
14   if len(prev_level) mod 2 = 1:
15     current_level.append(prev_level[-1])
16 until len(current_level) = 1
17 # return tree root
18  $\Psi :=$  current_level[0]
19 return  $\Psi$ 

HashUnlearn( $\mathcal{H}_U$ )
19  $\Psi :=$  Hash( $d_0$ )
20 for  $h_d \in \mathcal{H}_U$ :
21    $\Psi :=$  Hash( $\Psi, h_d$ )
22 return  $\Psi$ 

HashDataPoint( $d = (uid, x, y)$ )
23  $h_d :=$  Hash( $uid$ )
24 for  $x_j \in x$ :
25    $h_d :=$  Hash( $h_d, Hash(x_j)$ )
26  $h_d :=$  Hash( $h_d, Hash(y)$ )
27 return  $h_d$ 

ComputeTreePath( $d, \mathcal{H}_U$ )
28  $h_d :=$  HashDataPoint( $d$ )
29  $idx_d := \mathcal{H}_U.index(h_d)$ 
30 if  $idx_d = \perp$ :
31   return  $\perp$ 
32 # get intermediate root from tree below  $d$ 
33  $\Psi :=$  HashUnlearn( $\mathcal{H}_U[idx_d]$ )
34  $\pi_d := [\Psi]$ 
35 # add path from  $d$  to root
36 for  $h_d \in \mathcal{H}_U[idx_d+1:]$ :
37    $\pi_d.append(h_d)$ 
38 return  $\pi_d$ 

VerifyTreePath( $d, h_U, \pi_d$ )
39 # recompute root  $\Psi$  from path  $\pi_d$ 
40  $\Psi :=$  Hash( $\pi_d[0], HashDataPoint(d)$ )
41 for node in  $\pi_d[1:]$ :
42    $\Psi :=$  Hash( $\Psi, node$ )
43 # verify tree roots
44 return [ $\Psi = h_U$ ]

HashModel( $m = [w_0, \dots, w_n]$ )
45  $h_m :=$  Hash( $m[0]$ )
46 for  $w_i \in m[1:]$ :
47    $h_m :=$  Hash( $h_m, Hash(w_i)$ )
48 return  $h_m$ 

```

B. Completeness Proof

Proof (of Theorem 1). We need to prove the three properties in Definition 1 capturing the initialization, the proof of update and the proof of unlearning.

Initialization. First, running `Init` yields the initialized (empty) model m_0 , the commitment com_0 consisting of the hash of the model and the Merkle tree roots of the data points and unlearned data points (both empty at initialization) as well as the initial proof ρ_0 . It is clear to see that `VerifyInit` computes the same hashes and verification will pass such that $\Pr[\text{VerifyInit}(\text{pub}, \text{com}_0, \rho_0) = 0] = 0$.

Proof of Update. For the second property, recall the inputs and outputs of `ProveUpdate`. The state $\text{st}_{S, i-1}$ contains the dataset D_{i-1} of the previous iteration, the two sets $\mathcal{H}_{D_{i-1}}$ and $\mathcal{H}_{U_{i-1}}$ of hashed data points and the tree root $h_{U_{i-1}}$ of $\mathcal{H}_{U_{i-1}}$. All sets are updated using the intermediate datasets D_i^+ and U_i^+ . The commitment consists of the hash of the new model m_i —trained on the new dataset D_i —and the Merkle tree roots computed on the new sets \mathcal{H}_{D_i} and \mathcal{H}_{U_i} . The proof of update ρ_i consists of two statements and SNARK proofs. The first one proves that m_i was indeed computed on D_i and that the hash values in the commitment are consistent with m_i and D_i . The second SNARK proves that \mathcal{H}_{D_i} , $\mathcal{H}_{U_{i-1}}$ and \mathcal{H}_{U_i} map to tree roots h_{D_i} , $h_{U_{i-1}}$ and h_{U_i} and that \mathcal{H}_{U_i} is obtained from $\mathcal{H}_{U_{i-1}}$ and the hash values of U_i^+ . It further proves that the intersection of \mathcal{H}_{D_i} and \mathcal{H}_{U_i} is empty. By perfect completeness of the SNARK, both proofs will verify correctly. Note that there exists a special case where the server is unable to create a proof although the datasets are valid. This is the case whenever there exist two distinct data points $d \in D_i$, $d' \in U_i$, where $U_i = \bigcup_{j \in [i]} U_j^+$ is the dataset implicitly contained in \mathcal{H}_{U_i} , such that $\text{HashDataPoint}(d) = \text{HashDataPoint}(d')$. However, we only require computational completeness and assume that the datasets are provided by a PPT adversary. Then this translates to finding a collision for the hash function which happens with negligible probability if the hash function is collision-resistance. Hence, `VerifyUpdate` will output 1 with probability $1 - \text{negl}(\lambda)$. Further note that the commitment h_{D_i} to D_i satisfies completeness by construction and correctness of the Merkle tree.

Proof of Unlearning. Finally, consider the algorithm `ProveUnlearn`. If a data point d was unlearned in iteration i , then its hash is present in the set \mathcal{H}_{U_i} . The proof of unlearning π_d consists of the tree path to d in the Merkle tree of \mathcal{H}_{U_i} . Let com_i be the commitment for this iteration, then by correctness of the tree path algorithm, `VerifyUnlearn` will output 1 with probability 1. \square

C. Security Proof

Proof (of Theorem 2). Let \mathcal{A} be an adversary against unlearning security (as defined in Figure 2) of our instantiation. We will first argue that for all \mathcal{A} there exists an extractor \mathcal{E} that outputs the underlying datasets D_i . This follows directly from

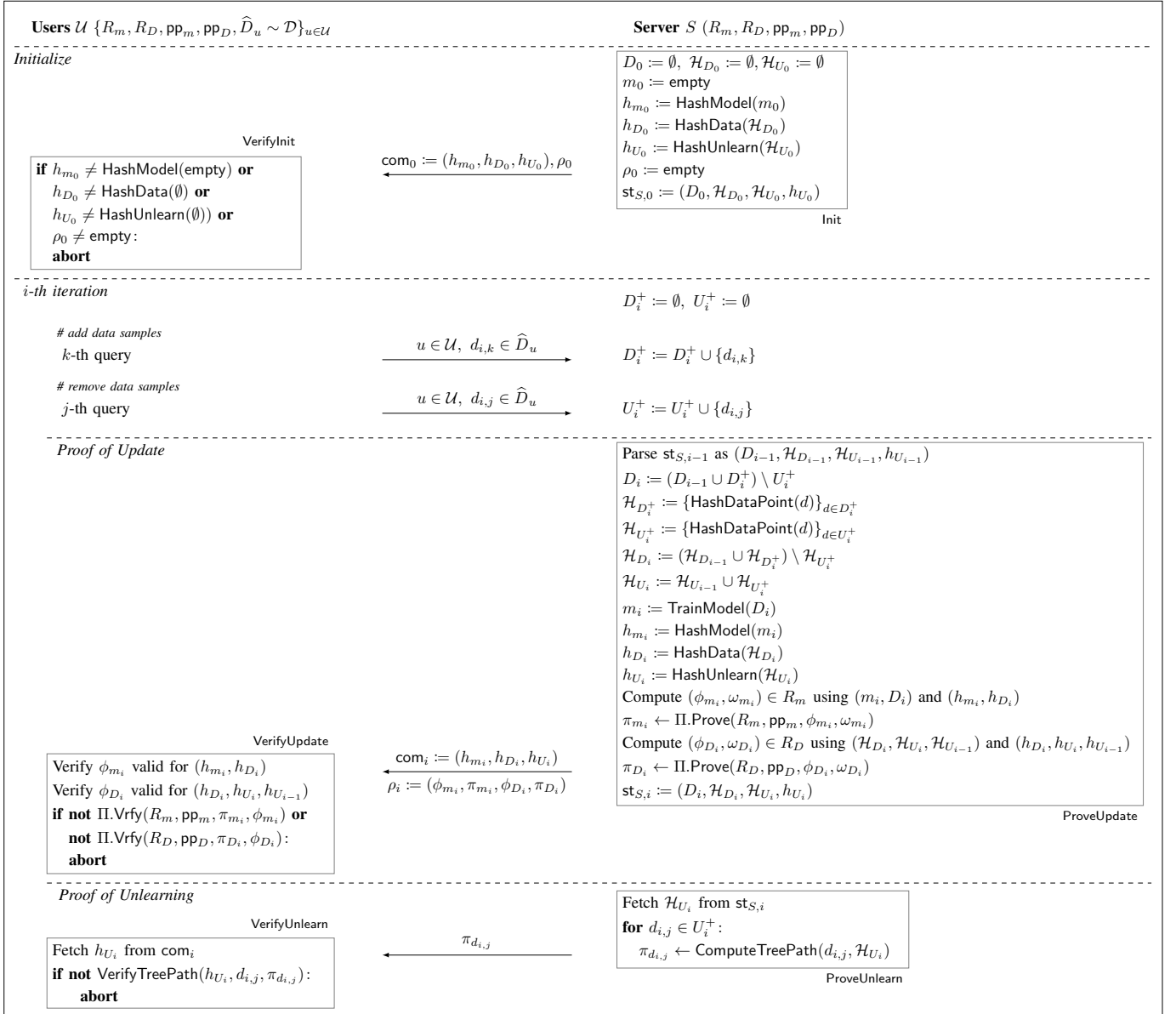


Fig. 7: **Instantiated Protocol Φ** . We instantiate the protocol with two primitives: a SNARK Π , and a hash function Hash.

the knowledge soundness of the SNARK for relation R_m since D_i is the witness used in the circuit (cf. Figure 3).

We will now prove the theorem by the sequence of games given in Figure 8 and analyze the probability that these games will output 1.

Game G_0 . Let G_0 be the original game GameUnlearn and \mathcal{E} be the extractor from the SNARK. Recall that the adversary must output a sequence of tuples $((k, d, \pi_d), \{\text{com}_i, \rho_i\}_{i \in [0, \ell]})$ for some $\ell \in \mathbb{N}$, where $\text{com}_i = (h_{m_i}, h_{D_i}, h_{U_i})$ for $i \in [0, \ell]$ and $\rho_i = (\phi_{m_i}, \pi_{m_i}, \phi_{D_i}, \pi_{D_i})$ for $i \in [\ell]$. We iterate over the winning conditions and return 0 as soon as one of them is violated (cf. Figure 8). For book-keeping we also compute all sets of unlearned data points U_i and U_i^+ from D_i . Note that

this is only a conceptual change at this point and we have

$$\Pr[G_0 \Rightarrow 1] = \Pr[\text{GameUnlearn}_{\mathcal{A}, \mathcal{E}, \Phi, \mathcal{D}}(1^\lambda) \Rightarrow 1].$$

Game G_1 . In G_1 , we compute the model m_i for each iteration from the datasets provided by \mathcal{A} and check whether the hash of this model corresponds to the hash h_{m_i} in the commitment. If this is not the case, the game outputs 0. We claim

$$|\Pr[G_1 \Rightarrow 1] - \Pr[G_0 \Rightarrow 1]| \leq \text{negl}(\lambda).$$

To prove the claim we argue in the following steps:

- First, π_{m_i} proves that the adversary knows a model m'_i and a dataset D'_i such that m'_i was computed on D'_i and that $h_{m_i} = \text{HashData}(m'_i)$ as well as $h_{D_i} = \text{HashData}(\mathcal{H}'_{D_i})$, where \mathcal{H}'_{D_i} is the set of all hashed

<pre> G₀-G₂ 00 (pp_m, td_m) ← Π.Setup(1^λ, R_m) 01 (pp_D, td_D) ← Π.Setup(1^λ, R_D) 02 ((k, d, π_d), {(h_{m_i}, h_{D_i}, h_{U_i}), ρ_i}_{i∈[0,ℓ]}; {D_i}_{i∈[0,ℓ]}) ← (A E)(R_m, pp_m, R_D, pp_D) 03 04 # Compute history of unlearned datasets 05 U₀ := ∅ 06 for i ∈ [ℓ]: 07 U_i⁺ := D_{i-1} \ D_i 08 U_i := U_{i-1} ∪ U_i⁺ 09 10 # Verify commitments 11 for i ∈ [0, ℓ]: 12 H_{D_i} := {HashDataPoint(d)}_{d∈D_i} 13 h'_{D_i} := HashData(H_{D_i}) 14 if h'_{D_i} ≠ h_{D_i}: 15 return 0 16 17 # Verify initialization 18 if (h_{m₀}, h_{D₀}, h_{U₀}) ≠ (HashModel(empty), HashData(∅), HashUnlearn(∅)) or ρ₀ ≠ empty: 19 return 0 </pre>	<pre> 20 # Verify proof of update 21 for i ∈ [ℓ] 22 Parse ρ_i as (φ_{m_i}, π_{m_i}, φ_{D_i}, π_{D_i}) 23 Verify φ_{m_i} valid for (h_{m_i}, h_{D_i}) and φ_{D_i} valid for (h_{D_i}, h_{U_i}, h_{U_{i-1}}) 24 if not Π.Vrfy(R_m, pp_m, π_{m_i}, φ_{m_i}) or not Π.Vrfy(R_D, pp_D, π_{D_i}, φ_{D_i}): 25 return 0 26 27 # Re-compute model and compare to h_{m_i} 28 for i ∈ [ℓ]: 29 m_i := TrainModel(D_i) // G₁-G₂ 30 if h_{m_i} ≠ HashModel(m_i) // G₁-G₂ 31 return 0 // G₁-G₂ 32 33 # Verify proof of unlearning 34 if not VerifyTreePath(h_{U_k}, d, π_d): 35 return 0 36 37 # Check membership of d in U_i 38 for i ∈ [k, ℓ]: // G₂ 39 if d ∉ U_i: // G₂ 40 return 0 // G₂ 41 42 # Adversary wins if point unlearned & re-added 43 if k < ℓ and d ∈ U_k⁺ and d ∈ D_ℓ: 44 return 1 45 return 0 </pre>
---	---

Fig. 8: Games G_0 - G_2 for the proof of Theorem 2.

values in D'_i , i.e., $\mathcal{H}'_{D'_i} := \{\text{HashDataPoint}(d)\}_{d \in D'_i}$. By soundness of the SNARK, the adversary can only forge a proof for an invalid statement with negligible probability, so we can assume the proof was generated honestly with a witness. By knowledge soundness, the extractor is able to compute this witness such that $D'_i = D_i$.

- Second, we claim that then $m'_i = m_i$. This is true unless the adversary finds a collision in the hash function such that $\text{HashModel}(m'_i) = \text{HashModel}(m_i) = h_{m_i}$, which we assume to happen only with negligible probability.

Game G_2 . In G_2 , we check whether the data point d output by \mathcal{A} is contained in the underlying datasets U_i of the k -th and all subsequent iterations. We will show that

$$|\Pr[G_2 \Rightarrow 1] - \Pr[G_1 \Rightarrow 1]| \leq \text{negl}(\lambda).$$

For this we first look at the second SNARK proof π_{D_i} . It proves that the adversary knows three sets \mathcal{H}'_{D_i} , $\mathcal{H}'_{U_{i-1}}$ and $\mathcal{H}'_{U_i^+}$ such that (1) h_{D_i} resp. $h_{U_{i-1}}$ are the Merkle tree roots of \mathcal{H}'_{D_i} resp. $\mathcal{H}'_{U_{i-1}}$, (2) h_{U_i} is the Merkle tree root of $\mathcal{H}'_{U_i} := \mathcal{H}'_{U_{i-1}} \cup \mathcal{H}'_{U_i^+}$ and (3) the intersection $\mathcal{H}'_{D_i} \cap \mathcal{H}'_{U_i}$ is empty. Again by soundness of the SNARK, the adversary can only prove an invalid statement with negligible probability.

Now we look at the proof of unlearning π_d . Recall that this proof consists of the tree path from the hashed data point d to the root h_{U_k} from the k -th commitment. Since the adversary can only win if the proof verifies successfully, we know that in this case the hash value of d , in the following denoted by $h_d := \text{HashDataPoint}(d)$, must be a leaf in the Merkle tree constructed from \mathcal{H}'_{U_k} . Unless the adversary finds another data point d' such that $\text{HashDataPoint}(d')$ maps to the same

hash value h_d —which happens with negligible probability—the point d must be contained in U_k .

Since the $(k+1)$ -th iteration requires to compute $\pi_{D_{k+1}}$ which proves that the value $h_{U_{k+1}}$ is obtained from $\mathcal{H}_{U_k} \cup \mathcal{H}'_{U_{k+1}^+}$, data point d must also be contained in U_{k+1} . Continuing with this argument, we can show that d is contained in all datasets U_i for $i \in \{k, \dots, \ell\}$.

Finally, we show that $\Pr[G_2 \Rightarrow 1] \leq \text{negl}(\lambda)$. For this, recall the third property of the SNARK proof π_{m_i} discussed above, namely that the intersection $\mathcal{H}'_{D_i} \cap \mathcal{H}'_{U_i}$ is empty. Together with the fact that the root of the Merkle tree constructed from \mathcal{H}'_{D_i} is h_{D_i} , the dataset \mathcal{H}'_{D_i} which was used for the creation of the proof must have been obtained from the corresponding dataset D_i (unless the adversary has found a collision in the hash function). We now look at the ℓ -th iteration. As shown above, we know that $d \in U_\ell$ and $h_d \in \mathcal{H}'_{U_\ell}$. The final winning condition requires that $d \in D_\ell$ and thus $h_d \in \mathcal{H}'_{D_\ell}$. Thus, the intersection of the two sets is not empty which means that the SNARK proof π_{D_ℓ} must belong to an invalid statement, which contradicts the previous statement and proves the final claim.

Collecting the probabilities yields

$$\Pr[\text{GameUnlearn}_{\mathcal{A}, \Phi, \mathcal{D}}(1^\lambda)] \leq \text{negl}(\lambda),$$

which concludes the proof of Theorem 2. \square