Agree-and-Prove: Generalized Proofs Of Knowledge and Applications

Christian Badertscher[1], Daniel Jost[2], and Ueli Maurer[2]

[1] *University of Edinburgh, Scotland,* christian.badertscher@ed.ac.uk
[2] *ETH Zurich, Switzerland,* {dajost,maurer}@inf.ethz.ch

Abstract

Proofs of knowledge (PoK) are one of the most fundamental notions in cryptography and have been used as a building block in numerous applications. The appeal of this notion is that it is parameterized by generic relations which an application can suitably instantiate. On the other hand, in many applications, a more generalized proof system would be desirable that captures aspects not considered by the low-level abstraction boundary of PoKs. First, the context in which the protocol is executed is encoded using a static auxiliary input, which is insufficient to represent a world with more dynamic setup, or even the case where the relation to be proven does depend on a setup. Second, proofs of knowledge do by definition not take into account the statement derivation process. Yet, it often impacts either the complexity of the associated interactive proof or the effective zero-knowledge guarantees that can still be provided by the proof system. Some of this critique has been observed and partially addressed by Bernhard et al. (PKC'15), who consider PoK in the presence of a random oracle, and Choudhuri et al. (Eurocrypt'19), who need PoK schemes in the presence of a ledger functionality.

However, the theoretical foundation of a generalized notion of PoK with setup-dependent relations is still missing. As a first contribution, we introduce this new notion and call it *agree-and-proof*. Agree-and-prove rigorously extends the basic PoK framework to include the missing aspects. The new notion provides clear semantics of correctness, soundness, and zero-knowledge in the presence of generic setup and under dynamic statement derivation.

As a second contribution, we show that the agree-and-prove notion is the natural abstraction for applications that are in fact generalized PoKs, but for which the existing isolated notions do not reveal this intrinsic connection. First, we consider proofs-of-ownership of files for client-side file deduplication. We cast the problem and some of its prominent schemes in our agree-and-prove framework and formally analyze their security. Finally, leveraging our generalized zero-knowledge formalization, we devise a novel scheme that is provably the privacy-preserving analogon of the known Merkle-Tree based proof-of-ownership protocol. As a second application, we consider entity authentication and two-factor authentication. We thereby demonstrate that the agree-and-prove notion can not only phrase generalized PoKs, but also, along the same lines, proofs of possession or ability, such as proving the correct usage of a hardware token.

1 Introduction

The concept of an *interactive proof* in which a prover's goal is to convince a verifier of the validity of a given statement is a fundamental theoretical concept in complexity theory and is established as a cornerstone in cryptography as well. Especially the task of proving to a party that one knows a certain piece of information, without necessarily revealing it, is an essential task in cryptography and in the design of cryptographic protocols. The formal concept capturing the essence of this task is called *proof of knowledge* [GMR85, TW87, FFS88, BG93] and has turned out to be a building block with countless applications. In a nutshell, the task of a prover is to convince the verifier that he knows a witness w for a statement x satisfying a relation $R(w, x)$. Part of the elegance of this

definition, fostering its wide applicability, is that it neither makes any particular assumption about the statements or witnesses, i.e., the definition is independent of how statements are generated, nor about the relation except that for cryptographic applications relations based on hardness assumptions are typically considered. It is clear that proofs of knowledge are a rather low-level building block, and thus not directly suitable to define the overall security goal of a concrete cryptographic application, for at least two reasons: First, it assumes that the statement and witness are static objects, and the statement is given to the parties as input. If the prover is required to be efficient, the witness is also given as input. However, most real world settings are much more dynamic by design and seem to require a more general building block of the above spirit. Typically we have two parties, both with a certain prior state and access to some setup, such as a random oracle, to approach each other and first (interactively) *agree* on a statement, and only then *prove* the agreed statement. Clearly non-trivial theoretical questions and definitional issues arise about the interplay between these two phases. Aside of such definitional challenges, the first phase (which we call agreement phase) might have in general impact on the obtained security guarantees: on one hand, an involved agreement phase (with or without setup) might be followed by a more efficient proof, as for example concretely showcased by Baum and Nof [BN19] recently, thanks to the derivation of a particular type of statement admitting efficiency gains (again, this could depend on a setup in general). On the other hand, an agreement phase does pose additional challenges such as to retaining the desired zero-knowledge guarantees. Hence, the agreement phase plays a crucial role that cannot be neglected.

Second, proofs of knowledge are formalized in a rather static world where the parties do not have access to any shared setup, such as a common reference string (CRS), a random oracle (RO), or simply a database storing user-logins. While such a setup can be and has been partially represented as an auxiliary input, it does not admit making the goal of the proof, i.e., the relation, depend on this setup and thus the state of the world. As a consequence, the security of applications that ought to be instantiations of some generalized proof-of-knowledge notion have been phrased using rather ad-hoc security definitions. For instance, the basic security of password-based authentication and identification schemes naturally appears to be captured as having to know the password; yet it is frequently described in a property based manner assuming that the password is drawn according to some distribution. Similarly, in the realm of cloud file storage, the security of schemes where a client aims to convince the server that he knows a specified file (e.g., client-side deduplication), has been formalized using a min-entropy based security definition [HHPSP11], which is not clear how this maps to practice. While all these examples follow a generic and dynamic agree-and-prove paradigm, their respective tailored security definitions suffer from the major disadvantage that they do not simply inherit advanced properties, most prominently the zero-knowledge property, that are precisely understood by traditional interactive proofs of knowledge.

The potential need for a more general formalization has already been envisioned by Goldreich [Gol06]. However, the shortcoming that traditional definitions of proofs of knowledge do not account for setup was only made explicit and partially addressed by two important recent works. First, in the work by Bernhard, Fischlin and Warinschi [BFW15] proofs of knowledge in the random oracle model are formalized including some dependency of the chosen statement on the random oracle. Their model does, however, not account for aspects such as programmability in either the knowledge-extraction or the zero-knowledge simulation experiments which, as we see in this work, can be rather involved and tricky to define. Furthermore, their model still treats the statement and witness as simple inputs to parties rather than permitting a generic negotiation that might be dependent on the random oracle. And finally, they consider classical NP-relations that do not depend on the random oracle which is a restriction that seems desirable to overcome. In a recent work, Choudhuri, Goyal, and Jain [CGJ19] provide a comprehensive treatment of secure multi-party computations in the presence of a ledger functionality. Not surprisingly, their treatment does unveil the need for proofs (of knowledge) in the presence of setups more complex than for example a random oracle. As in the above case [BFW15], their definition is tailored to this case, i.e., standard

zero-knowledge proofs are extended to allow all entities, i.e., prover, verifier, and the distinguisher (for the real and ideal transcripts in the zero-knowledge experiment) oracle access to the ledger functionality. In this setting, Choudhuri et al. [CGJ19] point out the important and subtle issue that several of the standard proof techniques, such as rewinding, are not easily applicable. The reason is that the setup does not only assist the honest parties but can significantly enrich the adversarial capabilities unless properly tamed by a clever protocol, as given in [CGJ19].

We note that this is only one of many subtleties that may occur in general interactive proofs in the presence of setup. It indeed seems to be a subtle task to formally capture the relevant probabilistic experiments (for correctness, soundness, and zero-knowledge) because they have to deal with (1) the omnipresent dependency—even of the relation to be proven—on the state of the setup, (2) the kind of access of the different entities to the setup (including the simulator), (3) the question how the state of the setup has been generated before a proof is executed, and (4) that different entities might have different side information regarding the state of the setup.

It is therefore important to devise the theoretical foundations of this more general concept that addresses all these subtle points above. We define in this paper a new concept that does include all the above mentioned missing elements in existing formalizations of proofs (or arguments) of knowledge. The concept we aim for abstracts from specific applications, gives clear interpretations for the correctness, soundness, and zero-knowledge properties in the presence of setups and interactive statement derivation, and is therefore suitable as a unifying cryptographic concept behind the above mentioned scenarios, including entity authentication and client-side deduplication.

1.1 Our Contributions

Agree-and-prove. Based on the above motivation, we introduce a new notion called *Agree-and-Prove* in Section 2. We generalize proofs (and arguments) of knowledge to dynamic settings where the prover and verifier, based on a setup and their initial state, first have to agree on a statement (agreement phase), of which the prover then convinces the verifier in a second phase (proof phase). The agree-and-prove notion is parametrized by an arbitrary setup functionality, where in particular not only the agreed statement but also the associated relations can depend on. We formulate both cases of programmable vs. non-programmable setups. Moreover, we define the equivalent of zero-knowledge and consider both, prover and verifier zero-knowledge which is needed in dynamic settings where both parties potentially have information they do not want to reveal. Finally, for the sake of generality, our definitions of zero-knowledge are parametrized using explicit leakage functions, accommodating for protocols that leak limited information. We conclude the definitional section on how the new, stand-alone notion can be understood in the context of composability.

We exemplify the applicability of the agree-and-prove notion as a template to define the security of (or within) cryptographic protocols following two main application scenarios. This leads to contributions of independent interest as we outline below.

Application to proofs of ownership. In Section 3, we dive into the application of proofs-of-ownership of files that aim to achieve secure client-side deduplication in a cloud storage system. In a nutshell, these schemes consist of a client convincing a server that it has a file (already stored in the server's database), but without uploading the entire file. The main security concern is thereby that the client cannot falsely convince the server. Overall, this problem is arguably a generalized proof of knowledge of the file. We therefore capture proofs of ownership as a particular instantiation of agree-and-prove, where the setup is the server database (plus possibly further tools such as a random oracle) and where the parties have to first agree on a file identifier (together with additional control information), and only then run an efficient proof phase. This view captures proofs of ownership naturally as an instantiation of a higher-level concept.

In this setting, we show that a naive hash-based scheme, where the hash value of a file should imply knowledge, is secure if the hash function is modeled as a private random oracle, i.e., exclusively

accessible to the prover and verifier. However, with a publicly accessible random oracle, as we argue formally, the scheme completely breaks which is the theoretical statement explaining the apparent insecurity observed in practice when using a concrete hash function [HHPSP11]. We further show how to retain security in this global random oracle setting by employing a stronger proof phase in which a Merkle-Tree based proposal as in [HHPSP11] is executed. This exemplifies that the particular statement derivation relative to the above setup does have an influence on the complexity of the associated proof—yet the overall agree-and-prove interface to an application, in this case providing the abstraction of a secure proof of ownership, remains identical.

We point out that compared to previous definitions in this space, including [HHPSP11, XZ14], our formalism is not tailor-made for a particular application, but justified by a higher-level abstraction. As a consequence, our formalization of proof of ownership does not require that we have to start from a distribution on files (i.e., following an entropy-based approach to knowledge). In our language, this is an additional assumption and can be formalized by a stronger setup, where files in the database have an intrinsic min-entropy. Stated differently, the weaker entropy-based definition put forth in [HHPSP11] (avoiding an extraction-based notion) can be seen as a proof of ownership in our sense with a stronger setup. We point out that while our soundness definition contains an extraction process, a stronger setup can significantly improve the power of the agreement phase (and reduce the complexity of the proof phase) up to the extent that extraction becomes trivial but soundness is still satisfied. Such situations occur in Sections 3.1 and 4.2 of this work.

Privacy-preserving proofs of ownership. We extend proofs of ownership to a privacy-aware setting in Section 3.3. Consider a situation where a set of clients (e.g. employees of the same company) share a secret key under which they apply client-side encryption of the files before uploading them to a server. We present a novel scheme that allows an employee to prove that he knows the plaintext of a ciphertext without having to know the randomness that was used during the encryption. We prove that our protocol does not reveal more information to both client or server than what is generally necessary for the task of client-side deduplication. Analogously to above, in comparison with previous approaches to privacy in this context [XZ14, GMO15], we formulate a cryptographic definition of privacy for proofs of ownership that is justified by a generalized zero-knowledge definition for agree-and-prove schemes.

Overall, our construction is designed as the privacy-preserving analogon of the above Merkle-Tree based solution. The thereby added privacy layer enables a modular analysis with a clear separation into the two tasks of proving ownership and protecting the privacy, which we believe is a desirable simplification compared to more "interleaved" approaches such as [XZ14, GMO15]. Furthermore, our construction is secure under standard cryptographic assumptions and compared to [GMO15] does not use random oracles.

Application to client authentication. In Section 4, a second application of agree-and-prove is presented. First, it is shown how password-based authentication naturally fits as an instantiation of the notion. Then, it is discussed how advanced security properties arising in the context of password-based authentication, such as protection from precomputed rainbow-tables, can be taken into account. Finally, we present a direct instantiation of Agree-and-Prove that captures two-factor authentication. The fact that the knowledge-relation can depend on the setup is thereby leveraged to demonstrate that the agree-and-prove notion can not only (as expected) formalize proofs and arguments of knowledge, but is in fact the cryptographic tool to capture in a similar spirit *proofs of possession* or *proofs of ability* such as the possession and use of a hardware-token.

2 Agree-and-Prove: Definition

In this section, we introduce our notion of an agree-and-prove scheme. Such a scheme is intended to capture a setting where two parties, the prover and the verifier, dynamically want to agree on a statement of whose validity the prover then wants to convince the verifier. The statement is not fixed beforehand and can in particular depend on the environment in which they execute the protocol as well as the parties' prior knowledge.

2.1 The Scenario

Analogous to a proof-of-knowledge scheme, an agree-and-prove scheme is only well defined with respect to a goal it should achieve. While in proof of knowledge such a goal is simply given by an NP-relation, it is now generalized for agree-and-prove schemes.

First, we consider in our notion *setup* that models some assumptions on the world in which we execute the protocol. Such a setup can simply consist of a CRS or a random oracle, but also can model further assumption such as a file database assigning files to certain identifiers in the case of a proof of ownership. Second, characterizing which statement the parties should agree on—in dependence of the setup and the parties' prior knowledge—is an integral part of specifying the goal of an agree-and-prove scheme. This is characterized by an *agreement relation*. Third, the *proof relation* characterizes what it means to satisfy the statement they agreed on (for which we simply use the common term proof). This relation generalizes the NP-relation of the proof-of-knowledge formalization, as it can capture notions of knowledge as well as more general properties about the relation between the statement and the setup.

We formally define this intuition below: An agree-and-prove scenario captures what is the assumed setting in which the protocol is executed and specifies the goal of the scheme.

Definition 2.1 (Agree-and-Prove Scenario)**.** An agree-and-prove scenario Ψ is a triple $\Psi := (\mathcal{F}, \mathcal{R}^{\mathcal{O}_\mathcal{F}(\cdot,\cdot,\cdot)}, C^{\mathcal{O}_\mathcal{F}(\cdot,\cdot,\cdot)})$, consisting of the following components:

- A *setup functionality* $\mathcal{F}$, which is a PPT ITM that consists of an initialization procedure init and then provides an oracle $\mathcal{O}_\mathcal{F}(i, \mathsf{q}, arg)$, where $i \in \{\mathsf{I}, \mathsf{P}, \mathsf{V}\}$ denotes a role, q denotes a keyword, and arg denotes the argument for this query. For technical reasons, the setup functionality keeps track of all queries (including the answer) by the prover, exposing them as an oracle $\mathcal{O}_\mathcal{F}(\texttt{QUERIES})$.

- An *agreement relation* $C^{\mathcal{O}_\mathcal{F}(\cdot,\cdot,\cdot)}$, which is a PPT oracle machine taking a unary encoding of the security parameter κ, two auxiliary inputs and a statement as inputs, and producing a decision bit as output.

- A *proof relation* $\mathcal{R}^{\mathcal{O}_\mathcal{F}(\cdot,\cdot,\cdot)}$, which is a PPT oracle machine taking a unary encoding of the security parameter κ, a statement x, and a witness w as inputs, and outputting a decision bit.

Observe that the setup functionality, as depicted in Figure 1, contains three oracles that operate on shared state and randomness. Both the prover and the verifier have their own oracles $\mathcal{O}_\mathcal{F}(\mathsf{P}, \cdot, \cdot)$ and $\mathcal{O}_\mathcal{F}(\mathsf{V}, \cdot, \cdot)$, respectively. This allows us, for instance, to express that if the setup contains a login database, then only the verifier has access to the passwords. In addition, there is also the third oracle $\mathcal{O}_\mathcal{F}(\mathsf{I}, \cdot, \cdot)$ capturing the information and prior influence that third parties can have about the setup. For example, the setup can either be a shared private key, or it can be a public CRS, where only in the latter case the oracle $\mathcal{O}_\mathcal{F}(\mathsf{I}, \cdot, \cdot)$ can access it. Some leakage about the information obtained through this oracle might also be passed to the parties as prior knowledge, capturing that for instance a dishonest prover might obtain hashes from other parties without knowing the respective queries.

Generic Setup Functionality $\mathcal{F}$

- init: Setup-Functionality initialization procedure
- $\mathcal{O}_{\mathcal{F}}(i, \mathsf{q}, arg)$: Interaction of setup with the participants, where
  - $i \in \{\mathsf{I}, \mathsf{P}, \mathsf{V}\}$ denotes a role.
  - q is a keyword.
  - arg is the argument for this query.
- $\mathcal{O}_{\mathcal{F}}(\texttt{QUERIES})$: Recorded queries of role P. Upon invocation, the oracle returns a list of $(\mathsf{q}, arg, reply)$ triples corresponding to all the queries made by role P so far.

Figure 1: A generic setup functionality $\mathcal{F}$, consisting of a initialization procedure init and then provides an oracle $\mathcal{O}_{\mathcal{F}}(i, \mathsf{q}, arg)$, where $i \in \{\mathsf{I}, \mathsf{P}, \mathsf{V}\}$ denotes a role, q denotes a keyword, and arg denotes the argument for this query. Furthermore, $\mathcal{F}$ keeps track of all the prover's queries.

2.2 The Protocols

For a given agree-and-prove scenario we can now define the notion of a corresponding agree-and-prove scheme. Such a scheme consists of two pairs of protocols for the prover and verifier, (P_1, V_1) and (P_2, V_2), where the former pair agrees on the statement for which the latter one then will execute the necessary proof. More concretely, the prover and verifier P_1 and V_1, respectively, output the statement they agreed on at the end of the first phase, or chooses to abort the protocol by outputting $\perp$ in case they could not agree. If they do agree on a statement, then at the end of the second phase the prover and verifier P_2 and V_2, respectively, output whether the proof has been successful or not.

Definition 2.2 (Agree-and-Prove Scheme)**.** An *agree-and-prove scheme* is a quadruple $\mathcal{S} := (P_1, P_2, V_1, V_2)$, consisting of the following four interactive PPT oracle machines:

- A (honest) first phase prover $P_1^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)}$ taking a unary encoding of the security parameter κ and an auxiliary input aux_p as inputs. It produces a statement x_p or $\perp$ as output, as well as a state st_p.
- A (honest) first phase verifier $V_1^{\mathcal{O}_{\mathcal{F}}(\mathsf{V},\cdot,\cdot)}$ taking a unary encoding of the security parameter κ and an auxiliary input aux_v as inputs. It produces a statement x_v or $\perp$ as output, as well as a state st_v.
- A (honest) second phase prover $P_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)}$ taking a state st_p as input, as well as a unary encoding of the security parameter κ, and producing as output a bit that indicates whether the proof has been accepted.
- A (honest) second phase verifier $V_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{V},\cdot,\cdot)}$ taking a state st_v as input, as well as a unary encoding of the security parameter κ, and producing as output a bit that indicates whether it accepts or rejects.

Observe that both the prover and the verifier can keep state between the two phases. Furthermore, note that both the prover and the verifier get an auxiliary input aux_p and aux_v, respectively, as input which models the parties' prior knowledge about the world, and where aux_p typically contains a witness (among other). Finally, note that in a slight abuse of notation, we treat an empty output at the end of the agreement phase as $x = \perp$.

Remark (*On variations of the computational model*)*.* We formulate the above algorithms as interactive and PPT for the sake of concreteness and since our presented applications live in this world. However, as for the traditional notions, various computational models and properties can be considered for

agree-and-prove such as allowing unbounded provers in Definition 2.2 or considering computational instead of information-theoretic soundness in Section 2.3 or different runtime requirements for extractors. Also, intermediate computational classes such as unbounded provers with limited calls to the setup (e.g., random oracle calls) would be possible to consider. On another dimension, one can restrict the number of messages exchanged or number of queries made in the proof phase. An obvious example would be to restrict the prover to send only a single message in the second phase which would overall establish (a generalized notion) of non-interactive proofs.

We move on to define the execution of an agree-and-prove scheme:

Definition 2.3. Let aux_p and aux_v denote two bit-strings, let $\mathcal{F}$ denote a setup functionality, and let $\mathcal{S} := (P_1, P_2, V_1, V_2)$ denote an agree-and-prove scheme. Then,

$$((x_p, st_p); (x_v, st_v); T) \leftarrow \big\langle P_1^{\mathcal{O}_\mathcal{F}(\mathsf{P},\cdot,\cdot)}, V_1^{\mathcal{O}_\mathcal{F}(\mathsf{V},\cdot,\cdot)} \big\rangle \big((1^\kappa, aux_p); (1^\kappa, aux_v)\big)$$

denotes the execution of the agreement phase between the honest first phase prover P_1 and the honest first phase verifier V_1. Note that we use the notation $(a; b; T) \leftarrow \langle A, B\rangle(x, y)$ to denote the interactive protocol execution of interactive algorithms A and B invoked on the inputs x and y, respectively, and where a and b are the resulting outputs of A and B, respectively, and where T denotes the communication transcript. Moreover,

$$(v; v'; \cdot) \leftarrow \big\langle P_2^{\mathcal{O}_\mathcal{F}(\mathsf{P},\cdot,\cdot)}, V_2^{\mathcal{O}_\mathcal{F}(\mathsf{V},\cdot,\cdot)} \big\rangle \big((1^\kappa, st_p); (1^\kappa, st_v)\big)$$

denotes the execution of the proof phase between the honest second phase prover P_2 and verifier V_2, with v and v' being the decision bit of the prover and verifier, respectively.

2.3 The Basic Security Notion

In this section, we define the agree-and-prove security notion that generalizes the traditional security requirements expected from proofs of knowledge.

2.3.1 Prior Knowledge or Context

Recall from the previous section that both parties take an auxiliary input. While the setup models the world which we assume the protocol to be executed in, those auxiliary inputs model the parties' prior knowledge (a similar concept was used in [Gol06, Section 4.7.5] on identification schemes). In the security experiment, those inputs will be generated by a respective algorithm.

Definition 2.4 (Input Generation Algorithm)**.** An input generation algorithm $I^{\mathcal{O}_\mathcal{F}(\mathsf{I},\cdot,\cdot)}$ for an agree-and-prove scenario $\Psi := (\mathcal{F}, \mathcal{R}^{\mathcal{O}_\mathcal{F}(\cdot,\cdot,\cdot)}, C^{\mathcal{O}_\mathcal{F}(\cdot,\cdot,\cdot)})$ is a PPT oracle machine taking a unary encoding of the security parameter κ as input and producing a pair of bit-strings (aux_p, aux_v), specifying the auxiliary inputs for the prover and verifier respectively, as output.

Note that this algorithm gets oracle access to the setup functionality via its own oracle $\mathcal{O}_\mathcal{F}(\mathsf{I}, \cdot, \cdot)$. This allows us to capture the prior knowledge and context in which the protocol is executed as part of the setup functionality itself and therefore as part of the agree-and-prove scenario. The input generation algorithm is then universally quantified over in the security definition, making a clean separation between the part we do make assumptions about (the functionality) and the part which we do not make assumption about, such as the prior knowledge or context as derived from the functionality (cf. also Section 2.5).

2.3.2 Programmability and Non-Programmability

There are many cases in which one would like to formalize that an extractor can program the setup (e.g., a backdoor in a CRS model). He should, however, be only allowed to do so in a "correct", i.e., undetectable, manner, as otherwise he might for instance force the prover and verifier to disagree on the statement and abort, thereby making the extraction game trivial. To this aim, we introduce the notion of a setup generation algorithm to formally capture (valid) programmability.

Definition 2.5 (Setup Generation Algorithm)**.** A setup generation algorithm SGen is a PPT taking a unary encoding of the security parameter κ as input. It outputs (the description of) a setup functionality $\mathcal{F}'$ and a trapdoor td as output.

We say that the setup generation algorithm SGen is *admissible* for an agree-and-prove scenario $\Psi := (\mathcal{F}, \mathcal{R}^{\mathcal{O}_{\mathcal{F}}(\cdot,\cdot,\cdot)}, C^{\mathcal{O}_{\mathcal{F}}(\cdot,\cdot,\cdot)})$, if for every PPT oracle machine $\mathcal{A}$ the following advantage

$$\mathsf{Adv}^{\mathsf{AP\text{-}Setup}}_{\Psi,\mathsf{SGen},\mathcal{A}} := \Pr^{\mathcal{F}.\mathsf{init}(1^\kappa)}[\mathcal{A}^{\mathcal{O}_{\mathcal{F}}(\cdot,\cdot,\cdot)}(1^\kappa) = 1] - \Pr^{(\mathcal{F}',td)\leftarrow\mathsf{SGen}(1^\kappa);\ \mathcal{F}'.\mathsf{init}(1^\kappa)}[\mathcal{A}^{\mathcal{O}_{\mathcal{F}'}(\cdot,\cdot,\cdot)}(1^\kappa) = 1]$$

is negligible in κ.

We formulate the security games that potentially require programmability in proofs using this generated setup instead of the real one. The extractor then gets a trapdoor td (e.g. for the generated CRS) that he can use for the extraction during the prove phase (or in the zero-knowledge case to simulate proofs). Other than that, the generated setup is directly used in the security game.

On the other hand a non-programmable setup corresponds to restricting setup generation algorithms that do not produce any leakage, which is, in accordance with the above definition, essentially equivalent to just taking the real setup functionality $\mathcal{F}$.

Finally, one can also easily model a mixture of programmable and non-programmable setups by considering the list $\mathcal{F} := (\mathcal{F}_1, \ldots, \mathcal{F}_k)$ as one setup functionality and a corresponding setup-generation algorithm $\mathsf{SGen}(1^\kappa) := (\mathsf{SGen}_1(1^\kappa), \ldots, \mathsf{SGen}_k(1^\kappa))$ where for each declared non-programmable setup $\mathcal{F}_i$, SGen_i is required to produce no leakage.

2.3.3 The Security Definition

Based on the notion of an input generation algorithm I and a setup generation algorithm SGen, we now define the security game. Before giving the definition, we explain and motivate the security conditions appearing in Figure 2 in the following paragraphs.

Correctness. The correctness experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Corr},\Psi}_{\mathcal{S},I}$ formalizes the following aspects: First, if the honest prover and verifier interact, then they need to agree on a valid statement, where the validity is given by the agreement relation $C^{\mathcal{O}_{\mathcal{F}}(\cdot,\cdot,\cdot)}$ of the agree-and-prove scenario (c.f. Section 2.1). This relation takes into account the parties' prior knowledge aux_p and aux_v, respectively, as well as the setup functionality $\mathcal{F}$. It also decides whether aborting is allowed or not. Second, if they do not abort, then correctness enforces that the honest prover does convince the verifier, i.e., it enforces that the proof succeeds. Note that we do not require the honest prover to explicitly output a witness for the proof relation—the fact that he in principal knows such a witness is covered by the soundness condition.

Soundness. The extraction experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Ext},\Psi}_{\mathcal{S},\mathsf{SGen},I,E,\hat{P}}$ formalizes that every (potentially dishonest) prover that can convince the verifier with probability at least $p(\kappa)$ of a statement x must know a witness w that satisfies the proof relation $\mathcal{R}^{\mathcal{O}_{\mathcal{F}}(\cdot,\cdot,\cdot)}(1^\kappa, x, w)$. Analogous to a proof of knowledge, we phrase this via the existence of an extractor. More precisely, this extraction property refers to the proof phase of the protocol, formalizing that the above guarantee holds for every valid statement x which the prover manages to agree on with the verifier. To reflect this in the security game, the agreement phase is executed exactly once, and cannot be rewound, thereby fixing the statement x,

Experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Corr},\Psi}_{\mathcal{S},I}$

Input: 1^κ, where $\kappa \in \mathbb{N}$
$\mathsf{flag}_{\mathrm{Corr}} \leftarrow 1$
Execute $\mathcal{F}.\mathsf{init}(1^\kappa)$
$(aux_p, aux_v) \leftarrow I^{\mathcal{O}_{\mathcal{F}}(\mathsf{I},\cdot,\cdot)}(1^\kappa)$
$((x_p, st_p); (x_v, st_v); \cdot) \leftarrow \langle P_1^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)}, V_1^{\mathcal{O}_{\mathcal{F}}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, aux_p); (1^\kappa, aux_v))$
if $x_p \neq x_v$ **or** $\neg C^{\mathcal{O}_{\mathcal{F}}(\cdot,\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x_v)$ **then**
  $\mathsf{flag}_{\mathrm{Corr}} \leftarrow 0$; **return**
else if $x_v \neq \bot$ **then**
  $(v; v'; \cdot) \leftarrow \langle P_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)}, V_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, st_p); (1^\kappa, st_v))$
  $\mathsf{flag}_{\mathrm{Corr}} \leftarrow (v' = \mathsf{accept}) \wedge (v' = v)$

Experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Ext},\Psi}_{\mathcal{S},\mathsf{SGen},I,E,\hat{P}}$

Input: $(1^\kappa, p)$, where $\kappa \in \mathbb{N}$ and $p\colon \mathbb{N} \to [0,1]$
$(\mathcal{F}', td) \leftarrow \mathsf{SGen}(1^\kappa)$
Execute $\mathcal{F}'.\mathsf{init}(1^\kappa)$
$(aux_p, aux_v) \leftarrow I^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa)$
$((\cdot, st_p); (x, st_v); T) \leftarrow \langle \hat{P}_1^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}, V_1^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, aux_p); (1^\kappa, aux_v))$
if $x = \bot$ **then**
  $\mathsf{flag}_{\mathrm{Ext}} \leftarrow 1$; **return**
Define $\mathsf{succ} := \Pr\left[v' = \mathsf{accept} \;:\; (\cdot; v'; \cdot) \leftarrow \langle \hat{P}_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}, V_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}\rangle(1^\kappa, st_p; (1^\kappa, st_v))\right]$
if $\mathsf{succ} \leq p(\kappa)$ **then**
  $\mathsf{flag}_{\mathrm{Ext}} \leftarrow 1$; **return**
$w \leftarrow E^{\mathcal{O}_E}(1^\kappa, x, aux_p, st_p, T)$
$\mathsf{flag}_{\mathrm{Ext}} \leftarrow (\mathcal{R}^{\mathcal{O}_{\mathcal{F}'}(\cdot,\cdot,\cdot)}(1^\kappa, x, w) = 1)$

---

where:
$\mathcal{O}_E := \mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot), \mathcal{O}_{\mathcal{F}'}(\mathtt{QUERIES}), \mathcal{R}^{\mathcal{O}_{\mathcal{F}'}(\cdot,\cdot,\cdot)}(1^\kappa,\cdot,\cdot), \mathcal{O}_{\mathsf{BBR}}(\hat{P}_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}(st_p))$

Figure 2: Security experiments for an Agree-and-Prove scheme. Top: The correctness experiment. Bottom: The extraction experiment to formalize soundness (the case where an honest verifier $V = (V_1, V_2)$ interacts with a dishonest prover $\hat{P} = (\hat{P}_1, \hat{P}_2)$).

the prover's and verifier's state st_p and st_v respectively, and the state of the setup functionality. (See the two final remarks on the experiment below for a more formal notion of "state"). It is important to note that our definition simultaneously captures validity and soundness[1] (justifying the common term soundness), as we let the extractor run w.r.t. any derived statement. That is, if V_1 accepts an invalid statement (without witness), there exists trivially no extractor that provokes $\mathsf{flag}_{\mathrm{Ext}} = 1$.

Back to extraction, with respect to this overall state after phase one, the extractor has to provide a witness w (within a reasonable time bound along the lines of Goldreich [Gol06, Definition 4.7.1]). To achieve extraction, the extractor gets the statement x, the prover's state st_p, and the communication transcript T of the agreement phase. Furthermore, he gets black-box rewinding access to the dishonest prover (communication) strategy $\hat{P}_2$, access to the prover's oracle of the setup functionality $\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)$, and access to the list of setup queries made by the prover, which is provided by the oracle $\mathcal{O}_{\mathcal{F}}(\mathtt{QUERIES})$. In contrast to a traditional proof of knowledge where the relation is deterministic and publicly known, we also provide an oracle to the extractor with black-box access to the predicate defining the proof relation (which in general could depend on the

[1] Note that in traditional proof-of-knowledge games, the extraction game is called validity and only valid statements $x \in L$ for some language L, are considered, whereas soundness requires an extra condition to capture security for the case that $x \notin L$.

randomness of the setup functionality). We refer the reader to the discussion after Definition 2.6 for the rationale behind these choices in comparison with traditional proof-of-knowledge systems.

Two formal considerations about the extraction experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Ext},\Psi}_{\mathcal{S},\mathsf{SGen},I,E,\hat{P}}$ are in order:

- For sake of concreteness, we understand the black-box rewinding oracle $\mathcal{O}_{\mathsf{BBR}}(\hat{P}_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)}(st_p))$ as a stateful *message-specification function* along the lines of Goldreich [Gol06, Definition 4.7.1]: more formally, when invoked with a random tape r, the oracle creates the machine $\hat{P}_2(st_p)$ in its initial configuration with random tape r. It then provides black-box access to the communication behavior by accepting incoming messages, performing the state transitions of $\hat{P}_2$ until it outputs the next message to be sent. Note that during this computation, oracle calls to $\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)$ might be made (which can neither be intercepted nor undone unless the setup functionality would allow this form of resetting).

- The formal expression[2]

$$\mathsf{succ} := \Pr\left[v' = \mathsf{accept} \;:\; (\cdot; v'; \cdot) \leftarrow \langle \hat{P}_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot)}, V_2^{\mathcal{O}_{\mathcal{F}}(\mathsf{V},\cdot,\cdot)} \rangle \big((1^\kappa, st_p); (1^\kappa, st_v)\big)\right]$$

  is associated to the following probability space: first, the start configuration of both machines of prover and verifier is the initial configuration with the specified input tape. The start configuration of machine $\mathcal{F}$ is the configuration at the end of the first phase (i.e., a snapshot). The probability space is then formed over the random coins of prover and verifier, and over the coins, i.e., positions on the random tape, of $\mathcal{F}$, that have not been read up to and until the above start configuration of machine $\mathcal{F}$.

It follows the definition of the security requirements of an agree-and-prove scheme.

Definition 2.6 (Agree-and-Prove Security)**.** Let $p\colon \mathbb{N} \to [0,1]$. An agree-and-prove scheme $\mathcal{S}$, for an agree-and-prove scenario Ψ, is secure up to soundness error p, if the following conditions hold, where the experiments are defined in Figure 2.

- *Correctness*: For all input generation algorithms I, the experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Corr},\Psi}_{\mathcal{S},I}$ returns with $\mathsf{flag}_{\mathrm{Corr}} = 1$ for all κ with probability 1.

- *Soundness*: There exists an extractor algorithm E and an admissible setup generation algorithm SGen, such that for all dishonest provers $\hat{P} = (\hat{P}_1, \hat{P}_2)$ and input generation algorithms I, the experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Ext},\Psi}_{\mathcal{S},\mathsf{SGen},I,E,\hat{P}}$ on input $(1^\kappa, p)$ returns with $\mathsf{flag}_{\mathrm{Ext}} = 1$ except with negligible probability. Furthermore, for some $c > 0$, the expected number of steps of extractor E within the experiment $\mathsf{Exp}^{\mathsf{AP\text{-}Ext},\Psi}_{\mathcal{S},\mathsf{SGen},I,E,\hat{P}}$ on input $(1^\kappa, p)$ is required to be bounded by $\kappa^c/(\mathsf{succ} - p(\kappa))$ (where the experiment ensures that $\mathsf{succ} > p(\cdot)$).

We next discuss some of the motivation and rationale behind the definition.

Discussion of selected elements. We first observe that providing the prover with the transcript of the agreement phase implies that in the proof phase we do not necessarily have a full-fledged proof or argument of knowledge of a witness as it, or parts of it, could already be contained in the agreement phase, thereby allowing for a more efficient proof phase.

Moreover, providing the extractor with the prover's input, state, and the setup queries from the first round also entails a couple of implications: we formalize naturally that it is sufficient for the prover to *know* a witness in order to pass the test—in contrast to more traditional definitions of proofs of knowledge requiring that the *communication needs to prove that he knows* one.

[2] We would like to stress that the formal evaluation of the expression has no side-effects on the state of any of the involved entities.

For instance, consider a shared URF $U(\cdot)$ between the prover and the verifier as a setup. If the statement is that the prover knows the pre-image x of y under some one-way permutation f, i.e, x such that $f(x) = y$ for x, y known to a verifier, then we would consider sending the correct evaluation under $U(x)$ as convincing, as a prover cannot guess $U(x)$ without querying it except with negligible probability. On the other hand, x cannot be extracted from the communication transcript $U(x)$. We consciously opted for this relaxed definition of knowledge to allow for broader applicability of the concept and because we believe it to capture the essence of a more general understanding of knowledge. For example, in the case of proof-of-ownership of files it is crucial that the communication complexity can be significantly smaller than the file.

2.4 Zero Knowledge

Analogous to a proof of knowledge, we can also require the agree-and-prove scheme to be zero knowledge. That is, whatever a (potentially dishonest) verifier can compute after interacting with the honest prover can also be computed by an appropriate simulator. Since we consider an interactive agreement phase where both parties get private information and a different view on the setup functionality, it however also makes sense to consider prover zero-knowledge. That is, we can phrase that both a verifier, as well as a prover should not learn anything about the other party's input nor about the other party's view on the setup functionality.

While a zero knowledge agree-and-prove protocol certainly represents the optimal case it is often already desirable to limit and explicitly quantify the leakage. To this end, we introduce the notion of a leakage oracle that the simulator is allowed to invoke. Furthermore, in the classical ZK definition, it is assumed that the verifier is always allowed to learn the statement and whether the prover has a valid witness. Since in our agree-and-prove notion the statement and the witness are not a priori fixed, this also has to be modeled as an explicit leakage. The classical zero-knowledge definition is then obtained by considering a leakage oracle that only reveals this information.

Definition 2.7. A *leakage oracle* $\mathcal{L}$ for a setup functionality $\mathcal{F}$ is an oracle PPT ITM that consists of a initialization procedure init and an oracle $\mathcal{L}^{\mathcal{O}_{\mathcal{F}}(\mathsf{P},\cdot,\cdot),\mathcal{O}_{\mathcal{F}}(\mathsf{V},\cdot,\cdot)}(1^\kappa, aux, query)$, allowing the simulator to ask certain queries *query* which are evaluated on the other party's input *aux* and view on the setup.

We now proceed to define the classical property that the scheme is zero-knowledge, up to some explicit leakage, with respect to the dishonest verifier. Our definition follows the spirit of the standard (standalone) simulation paradigm where two different settings are compared and should be indistinguishable: one is the real protocol execution, and the other one is the execution where the actions of the dishonest party are simulated by a simulator (having access to the leakage oracle). The distinguishing metric is formalized by a distinguisher D that is given the output of the dishonest verifier, the protocol outputs of the honest party, and access to the context information, i.e., it is given the auxiliary input and access to interface I of the setup.[3] Note that the outputs of the dishonest verifier in this case (denoted out_1, out_2 in the security game) can contain anything the malicious strategy decides to output[4] (in particular, the entire transcript and information about the setup).

In summary, the definition captures that if the AaP scheme is run as a sub-system in a context specified by the input-generation algorithm, then the resulting trace it leaves by means of transcript and output of the honest party does not leak more than what is specified by the ideal leakage oracle and therefore is in line with the stand-alone simulation paradigm used in traditional zero-knowledge.

[3] The motivation what information to give to the distinguisher will become apparent in Section 2.5 when we discuss compositional aspects of the notion. We point out that the interface of the honest party to the setup, in this case P, is never exposed to any attacker or the distinguisher because the honest party is running the protocol and only the actual protocol outputs are visible to an "environment".

[4] We point out that both outputs are needed: for example if the protocol aborts after the first phase, we require the agree-phase not leak anything beyond what is specified by the leakage oracle.

Experiment $\mathsf{Exp}^{\mathsf{AP\text{-}ZK\text{-}V},\Psi,\mathcal{S},\mathcal{L}_P}_{\hat{V},I,S,\mathsf{SGen},D}$

Input: 1^κ, where $\kappa \in \mathbb{N}$
$b \twoheadleftarrow \{0,1\}$
if $b = 0$ **then**
  $\mathcal{F}' \leftarrow \mathcal{F}$
  Execute $\mathcal{F}'.\mathsf{init}(1^\kappa)$
  $(aux_p, aux_v) \leftarrow I^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa)$
  $((x, st_p); out_1; \cdot) \leftarrow \langle P_1^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}, \hat{V}_1^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, aux_p); (1^\kappa, aux_v))$
  $(v; out_2; \cdot) \leftarrow \langle P_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}, \hat{V}_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, st_p); (1^\kappa, out_1))$
else
  $(\mathcal{F}', td) \leftarrow \mathsf{SGen}(1^\kappa)$
  Execute $\mathcal{F}'.\mathsf{init}(1^\kappa)$ and $\mathcal{L}_P.\mathsf{init}(1^\kappa)$
  $(aux_p, aux_v) \leftarrow I^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa)$
  $(x, v; out_1, out_2) \leftarrow S^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot),\mathcal{L}_P^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot),\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}(1^\kappa, aux_p,\cdot)}(1^\kappa, aux_v, td)$
$b' \leftarrow D^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x, v, out_1, out_2)$
$\mathsf{flag}_{\mathrm{Guessed}} \coloneqq b = b'$

Experiment $\mathsf{Exp}^{\mathsf{AP\text{-}ZK\text{-}P},\Psi,\mathcal{S},\mathcal{L}_V}_{\hat{P},I,S,\mathsf{SGen},D}$

Input: 1^κ, where $\kappa \in \mathbb{N}$
$b \twoheadleftarrow \{0,1\}$
if $b = 0$ **then**
  $\mathcal{F}' \leftarrow \mathcal{F}$
  Execute $\mathcal{F}'.\mathsf{init}(1^\kappa)$
  $(aux_p, aux_v) \leftarrow I^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa)$
  $(out_1; (x, st_v); \cdot) \leftarrow \langle \hat{P}_1^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}, V_1^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, aux_p); (1^\kappa, aux_v))$
  if $x \neq \bot$ **then**
    $(out_2; v; \cdot) \leftarrow \langle \hat{P}_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot)}, V_2^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}\rangle((1^\kappa, out_1); (1^\kappa, st_v))$
  else
    $out_2, v \leftarrow \bot$
else
  $(\mathcal{F}', td) \leftarrow \mathsf{SGen}(1^\kappa)$
  Execute $\mathcal{F}'.\mathsf{init}(1^\kappa)$ and $\mathcal{L}_V.\mathsf{init}(1^\kappa)$
  $(aux_p, aux_v) \leftarrow I^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa)$
  $(out_1, out_2; x, v) \leftarrow S^{\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot),\mathcal{L}_V^{\mathcal{O}_{\mathcal{F}'}(\mathsf{P},\cdot,\cdot),\mathcal{O}_{\mathcal{F}'}(\mathsf{V},\cdot,\cdot)}(1^\kappa, aux_v,\cdot)}(1^\kappa, aux_p, td)$
$b' \leftarrow D^{\mathcal{O}_{\mathcal{F}'}(\mathsf{I},\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x, v, out_1, out_2)$
$\mathsf{flag}_{\mathrm{Guessed}} \coloneqq b = b'$

Figure 3: The zero-knowledge security experiments for an AaP scheme. The first experiment phrases verifier zero-knowledge, whereas the second one phrases prover zero-knowledge. The distinguisher is given the auxiliary input, both parties' outputs of the interaction.

Definition 2.8. Let $\mathcal{L}_P$ denote a leakage oracle. An agree-and-prove scheme $\mathcal{S}$, for an agree-and-prove scenario Ψ, is *verifier zero-knowledge up to leakage* $\mathcal{L}_P$ if for all dishonest verifiers $\hat{V} = (\hat{V}_1, \hat{V}_2)$ and all input generation algorithms I, there exists an efficient simulator S and an admissible setup generation algorithm SGen such that for all efficient distinguishers D it holds that the experiment $\mathsf{Exp}^{\mathsf{AP\text{-}ZK\text{-}V},\Psi,\mathcal{S},\mathcal{L}_P}_{\hat{V},I,S,\mathsf{SGen},D}$ on input 1^κ returns with $\mathsf{flag}_{\mathrm{Guessed}} = 1$ with probability at most negligibly larger than $\frac{1}{2}$. The experiment is defined in Figure 3.

Note that in the experiment depicted in Figure 3 we assumed that a dishonest verifier $\hat{V}_1$ is not restricted to output something of the form (x, st_v) at the end of the agreement phase, but can produce an arbitrary output instead. Moreover, observe that a dishonest verifier is not forced to abort, but in principle can always try to execute the proof phase with the honest prover.

Now we also define the symmetrical property that the scheme is zero-knowledge with respect

to the prover. When defining this notion, care has to be taken that observing the honest verifier aborting after the agreement phase does not leak information either. In the following definition it is assumed that the honest verifier refuses to execute V_2 in case V_1 ended with an abort, i.e., at most signaling the abort to anyone. Again, an AaP protocol run must be indistinguishable from a simulated run where the simulator has access to a specific leakage oracle, and must simulate the protocol interaction with the malicious prover.

Definition 2.9. The scheme is said to be *prover zero-knowledge up to leakage* $\mathcal{L}$, if the same property as in Definition 2.8 holds for all dishonest provers $\hat{P} = (\hat{P}_1, \hat{P}_2)$ in the experiment $\mathsf{Exp}^{\mathsf{AP\text{-}ZK\text{-}P},\Psi,\mathcal{S},\mathcal{L}_V}_{\hat{P},I,S,\mathsf{SGen},D}$, which is defined in Figure 3.

2.5 On the Composability of the Notion

In this section, we discuss some aspects of composability of the agree-and-prove notion. More precisely, we show how the standalone security definition can be embedded in a larger context, how the setup functionality of the standalone definition should be understood, and when it can be shared among different agree-and-prove instances. More formally, we show under which circumstances an agree-and-prove scheme can be securely used as a subroutine of a larger protocol when either the setup functionality is assumed to be per instance or shared among different protocol instances.

There exist several flavors of composable properties, such as sequential (self) composition, concurrent composition, and universal composition. While a universally composable analysis is outside the scope of this paper, we aim to show that our notion implies that any such scheme can be safely used as a subroutine of a more complex protocol that runs potentially many instances of agree-and-prove sequentially. Note thereby that the purpose of our sequential composition results differs from the one about traditional proofs of knowledge (with auxiliary inputs). For proofs of knowledge, the sequential composition theorem mainly serves as a technical tool to prove amplification, i.e., that the soundness of a PoK can be amplified using sequential repetitions without harming the zero-knowledge property, and the statement lives within the definitional framing of a PoK itself. In contrast, agree-and-prove is geared towards modeling more general executions (similar to proofs of retrievability [BJO09]), where a priori no semantics of amplification has to exist. Hence, a composition statement about AaP has to be embedded in a more general framework with the primary purpose of validating the rationale of the definitional aspects of the standalone definition.

In the following, we consider a hybrid MPC-style execution model of protocols. Let us briefly recap the basic model of Canetti [Can00] with static corruptions. In this model, parties can communicate with each other, as well as invoke functionalities as subroutines. This communication proceeds in rounds, where in any round each party can either send a message to all other parties, or invoke a functionality. We thereby assume the existence of a special adversary machine that fully controls all the dishonest parties. Let $\pi_1, \dots, \pi_n$ be n agree-and-prove protocols, with two distinct parties running π_i in their respective roles. As we only care about sequential composition, we assume that the honest parties only produce a single output (consisting of the statement and whether the proof has been accepted or not) at the very end of the protocol execution. Moreover, let $\rho^{\pi_1,\dots,\pi_n}$ be some arbitrary protocol that sequentially uses them as subroutine calls. In particular, sequentiality implies that if both prover and verifier are honest, then the i-th invocation is only started once the (i-1)-th finished. In the case where each AaP protocol π_i has its own instance of a setup functionality $\mathcal{F}$, this functionality of the standalone definition thereby translates to the following trusted party assumed to be present in this hybrid world:

- During the execution of the corresponding AaP protocol, both prover and verifier can make subroutine calls in their respective roles P and V. To this end, we assume that both parties explicitly have to initiate and terminate the interaction, and the protocol is said to be running once all the honest parties instructed it to do so, and analogously for termination.

- During the execution, the adversary can furthermore make calls in the name of the corrupted party.
- Before and after the execution (but not during), everybody can make subroutine calls in role I.

That is, the roles P, and V of an agree-and-prove setup functionality $\mathcal{F}$ models the prover and verifiers access during the protocol execution, whereas the role I models the effect the context (both honest and dishonest parties) can have outside the interaction.

In order to show that the standalone security definitions imply security in the execution environment above, we have to show an appropriate reduction to an input-generation algorithm, an adversary, and a distinguisher for the game-based notions. That is, we have to show that our notion of an input-generation algorithm is powerful enough to capture any preceding execution, and the distinguisher for the zero-knowledge games the succeeding execution. It is straightforward to see that a corresponding input-generation algorithm is obtained by internally executing ρ, the AaP instances $1, \dots, i-1$ (including their setup functionalities), and the adversary up to the point where the i-th instance starts—and then provide the correct input to the honest party and its current state to the dishonest party.

Proposition 2.10. *Suppose that we have n instances of an agree-and-prove protocol π_1, $\pi_2, \dots, \pi_n$, that are executed sequentially by some higher level protocol $\rho^{\pi_1,\dots,\pi_n}$ and where each instance has its independent copy of the setup functionality. Then, for each $1 \le i \le n$, the execution up to the beginning of π_i (including the executions of π_j for $1 \le j < i$) can be encoded as an input-generation algorithm for the i-th instance, and the succeeding execution (including π_j for $j > i$) as a distinguisher, such that the security properties of the i-th instance reduce to the stand-alone security definition presented above.*

Proof (Sketch). The input-generation algorithm internally executes ρ, the AaP instances $1, \dots, i-1$ (including their setup functionalities), as well as the adversary. Whenever one of the parties, or the adversary calls the i-th setup functionality in role I, the input-generation algorithm does accordingly. The round in which the i-th instance would start by being invoked as a subroutine of ρ, the algorithm sets *aux* of the honest party (or parties) to the corresponding input, and the dishonest party's *aux* to its current state and halts. Now consider the following dishonest party $\hat{P}$ or $\hat{V}$ that simply continues the execution of the adversary using the passed state, and outputs its internal state at the end of the protocol execution. The distinguisher can then resume the execution by using the honest party's output as well as the adversary's state to execute ρ with the remaining AaP instances.

Now consider briefly the security properties of the i-th AaP instance. First, in the case of correctness both parties are honest. It is easy to see that the sketched input-generation algorithm provides the same inputs as the honest parties get in the overall execution. Hence, violating the correctness in the overall execution would immediately imply it in the standalone experiment. Analogously for soundness: here, the transformed input generation algorithm and dishonest prover lead to the same execution as the i-th one in the overall experiment. Hence, if the prover could convince the verifier of a wrong statement in the overall experiment, so could he in the standalone experiment. Finally, an analogous argument can be made about the simulator for the zero-knowledge properties. There, the hybrid-world execution can be compared to an ideal-world execution with the simulator providing outputs for both parties, having only access to the dishonest party's interface and the leakage oracle—as well as the (potential) trapdoor of the functionality. □

Finally, we now consider the case of shared setup functionalities among different protocols. The natural question arising is: which property does a setup functionality need to satisfy, such that reusing it is sound? First of all, we restrict ourselves to non-programmable setup functionalities, since shared programmable setup leads to several complications as for instance pointed out by [CDG+18] and a more detailed treatment on the level of allowed programmability is outside the scope of this paper. Furthermore, since the role I models the interaction with the setup before and

after the protocol instance, a natural choice is that this role must be powerful enough to carry out the influence of another protocol instance, i.e., the input-generation algorithm must be able to simulate the other instances using the same setup functionality, while only having access to $\mathcal{F}$ in role I, which is for instance trivially true for a strong global random oracle. Under those restrictions, we show that the setup functionality can be securely reused by different agree-and-prove (instances) as follows.

Proposition 2.11. *Suppose that we have n instances of an agree-and-prove protocol $\pi_1, \pi_2, \ldots, \pi_n$ that share a non-programmable setup functionality $\mathcal{F}$, such that the role I is as powerful as the other two roles, i.e., if every command for P and V can also be executed in role I with the same effect. Then, for a higher level protocol $\rho^{\pi_1,\ldots,\pi_n}$ that executes them sequentially, the security reduces to the stand-alone security definitions.*

Sketch. The proof follows the one of Proposition 2.10 with an analogous reduction of the environment to an input-generation algorithm, adversary, and distinguisher. In particular, whenever one of the parties emulated by the input-generation algorithm accesses the setup functionality using either P or V, then it simply executes it in role I. Some care has to be taken with respect to the zero-knowledge properties, where we require that in each instance, the dishonest prover (or verifier) might learn at most as much as specified by an independent instance of the leakage oracle $\mathcal{L}_V$ (or $\mathcal{L}_P$). This now follows from our assumption on non-programmability, since from the point of view of a simulator for the ith instance, the previous executions (be it simulations or not) are just programs whose accesses to the setup can be emulated using (the sufficiently powerful) interface I (and in particular, no further information, such as a trapdoor is needed). □

3 Application to Proof-of-Ownership of Files

File deduplication is a cornerstone of every cloud storage provider. Client-side, rather than server-side, deduplication furthermore provides the additional benefit of reducing the bandwidth requirements and improving the speed. In such a scheme, the client—instead of just uploading the file—first tries to figure out whether the server already possesses a copy of the file, and if so simply requests the server to also grant him access to the file. Several commercial providers implemented client-side deduplication using a naive scheme of identifying the file using hash values. This allowed users to covertly abuse the storage as a content distribution network, prompting the storage providers to disable client-side deduplication [MSL+11].

As a response, Halevi, Harnik, Pinkas, and Shulman-Peleg [HHPSP11] introduced the first rigorous security treatment of client-side deduplication, formalizing the primitive of a proof of ownership. While intuitively their notion formalizes that a client can only claim a file he knows, Halevi et al. formalized the proof-of-ownership concept as an entropy-based notion, rather than a proof-of-knowledge based notion.

In this section, we show that our agree-and-prove notion is the natural candidate for formalizing the security of client-side deduplication. Besides the basic requirements, we also present a privacy preserving scheme that is applicable if users additionally employ client-side encryption.

3.1 Proof-of-Ownership with a Local RO

We first abstract this application as an agree-and-prove scenario which includes the setup and the relation we want to prove. We finally give a description of the scheme.

Setup. We describe the setup in very simple terms. We want to deal with an array of pairs $L = (\mathrm{id}_i, F_{\mathrm{id}_i})_{i\in[n]}$, where $\mathrm{id}_i, F_{\mathrm{id}_i} \in \{0,1\}^*$ and for all i, id_i is unique in L. The setup of the Proof-of-Ownership scenario is thus a functionality $\mathcal{F}_{\mathsf{DB,RO}}$ that first expects such a list from the input-generation algorithm (recall that the input-generation algorithm also defines the state

of the prover). The setup gives the verifier access to the list L. The setup further provides a (non-programmable) random oracle to the prover and the verifier (but not to the input generation algorithm). The description can be found in Figure 4.

Agreement and Proof Relations. Our goal is to show that a very simple File-Ownership protocol is indeed a valid agree-and-prove scheme with the above setup. The statement that prover and verifier agree on is a file identity and the relation to be proven is that the prover knows the file with the corresponding identity. More formally, the agreement relation is defined via the condition

$$C^{\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\cdot,\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x) = 1 \;:\leftrightarrow\; x = \bot \vee \exists i : L(i) = (x, \cdot), \tag{1}$$

which can be efficiently implemented by C by calling $\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{V}, \mathtt{getFile}, x)$ and verifying that the answer is some $F \neq \bot$.

Accordingly, the proof relation is defined via the condition

$$\mathcal{R}^{\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\cdot,\cdot,\cdot)}(1^\kappa, x, w) = 1 \;:\leftrightarrow\; (x, w) \in L \tag{2}$$

which can be efficiently implemented by $\mathcal{R}$ by calling $\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{V}, \mathtt{getFile}, x)$ and verifying that the returned value F equals w.

The scheme. The scheme is described in Figure 4. The agreement phase consists of the prover stating the identity of the file, of which ownership is to be proven, and providing a hash of it. The verifier checks the hash of the claimed file and upon success, informs the prover. The agreed statement is $x = \mathrm{id}$ such that there is an index i with $L(i) = (\mathrm{id}, \cdot)$. As we will see in the analysis, after this agreement phase no further proof phase is needed. Hence, the prover P_2 halts and V_2 outputs 1 if and only if V_1 did successfully derive the statement.

Analysis. In order for the verifier to accept, in the agreement phase the prover has to send (id, h) such that $H(F_{\mathrm{id}}) = h$. Since the auxiliary input from I does not depend on H, there are two possibilities: either the prover queried the random oracle at position F_{id}, in which case he has the file, or he guessed the hash. The latter can, however, only happen with negligible probability. Hence we get the following statement.

Theorem 3.1. *The agree-and-prove scheme from Figure 4, for the above described agree-and-prove scenario capturing proof-of-ownership with a local RO, is secure (with soundness error $p(\kappa) := 0$).*

Proof. By the definition of V_1, the experiment $\mathsf{Exp}_{\mathcal{S},I}^{\mathsf{AP\text{-}Corr},\Psi}$ returns with $\mathsf{flag}_{\mathrm{Corr}} = 1$ with probability 1 for all input generation algorithms I, as the verifier outputs a statement $x \neq \bot$ only if the correctness predicate is satisfied. Now consider the following extractor E. Given (id, h) from the transcript T, with $x = \mathrm{id} \neq \bot$, the extractor fetches the list L of random oracle queries from the prover by calling $\mathcal{O}_{\mathcal{F}}(\mathtt{QUERIES})$. For each pair $(q_i, r_i) \in L$, E checks whether $r_i = h$ and (q_i, h) satisfies the proof relation. If such a pair is found, the extractor returns this q_i, otherwise it aborts. It remains to analyze the probability of the experiment $\mathsf{Exp}_{\mathcal{S},\mathsf{SGen},I,E,\hat{P}}^{\mathsf{AP\text{-}Ext},\Psi}$ returning with $\mathsf{flag}_{\mathrm{Ext}} = 1$. First, observe that the verifier decides at the end of the agreement phase. Hence, in the experiment we have that succ is either zero or one. If $\mathsf{succ} = 0$, we trivially have $\mathsf{flag}_{\mathrm{Ext}} = 1$. If $\mathsf{succ} = 1$, assume that the extractor does not succeeds. In that case, we have that P_1 sent (id, h) such that $H(F_{\mathrm{id}}) = h$ without having any information about $H(F_{\mathrm{id}})$, since he neither queried it himself nor got any information about it from I (which cannot query it) nor V_1. Given that h is of the length κ, this can happen with probability at most $2^{-\kappa}$. □

Prover Protocols P_1 and P_2

Prover $P_1(1^\kappa, aux_p)$:

1: Parse aux_p as (id, F). If this fails, set $st_p = \mathsf{Fail}$ and halt.

2: Call $h \leftarrow \mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{P}, \mathtt{ROeval}, F)$ and send (id, h) to V_1.

3: Upon receiving ok define $st_p = \mathsf{Done}$ and statement $x := \mathrm{id}$ Otherwise, set $st_p = \mathsf{Fail}$ and $x := \bot$.

Prover $P_2(1^\kappa, st_p)$:

1: Halt with 1 (accept) if $st_p = \mathsf{Done}$ and otherwise with 0.

Verifier Protocols V_1 and V_2

Verifier $V_1(1^\kappa, aux_v)$:

1: Upon receiving a message (id, h) from P_1, obtain $F \leftarrow \mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{V}, \mathtt{getFile}, \mathrm{id})$ and obtain $h' \leftarrow \mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{V}, \mathtt{ROeval}, F)$. Then do the following:
   - If $h = h'$, define statement $x := \mathrm{id}$ and $m, st_v \leftarrow \mathsf{ok}$.
   - If $h \neq h'$ define statement $x := \bot$ and $m, st_v \leftarrow \bot$.

   Send m to P_1.

Verifier $V_2(1^\kappa, st_v)$**, assume** $x \neq \bot$:

1: If $st_v = \mathsf{ok}$ then output 1, otherwise output 0.

File-Ownership setup $\mathcal{F}_{\mathsf{DB,RO}}$

- init: $H \leftarrow []$ and L is the empty array.
- $\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{I}, \mathtt{defineDB}, L')$: If L is the empty array, then do the following: If L' is an array of pairs $(\mathrm{id}_i, F_{\mathrm{id}_i})_{i \in [n]}$, where $\mathrm{id}_i, F_{\mathrm{id}_i} \in \{0,1\}^*$ and for all i, id_i is unique in L', then set $L \leftarrow L'$. Output L in any case.
- $\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{V}, \mathtt{getFile}, \mathrm{id})$: If $i \in \{\mathsf{I}, \mathsf{V}\}$ and $\mathrm{id} \in L$ then return F_{id}. In any other case, return $\bot$.
- $\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(i, \mathtt{ROeval}, x)$: If $i \in \{\mathsf{P}, \mathsf{V}\}$ do the following: if $H[x]$ is not yet defined, first choose $y \twoheadleftarrow \{0,1\}^\kappa$ and set $H[x] := y$; finally, return $H[x]$. If $i = \mathsf{I}$, return $\bot$.

Figure 4: The description of the prover protocols (left) and the verifier protocols (right), and the concrete setup functionality (bottom).

3.2 Proof-of-Ownership with a Global RO

While the above approach is sound if prover and verifier share a random function among each other (which is only used locally in the agree-and-prove context), it is not considered secure in practice, since one does usually have to assume that access to such a random function is not exclusive to the prover. In this section, we discuss the alternative, where the random oracle is accessible by all three roles and in particular by the input generation algorithm.

The new scenario. We modify the setup slightly to allow all roles access to the random oracle, i.e., even an input generation algorithm could obtain the RO outputs and hence hashes might be part of the prior knowledge. The resulting setup functionality $\mathcal{F}_{\mathsf{DB,GRO}}$ is defined analogous to the one from Figure 4, except that also `ROeval` queries are also admitted for the role I. The relations remain the same as in equations (1) and (2), except that they are with respect to the setup $\mathcal{F}_{\mathsf{DB,GRO}}$.

3.2.1 Insecurity of the Simple Scheme

It is easy to see that with a global RO, the scheme in Section 3.1 loses its guarantees. To be more concrete, the scheme has the following property in a GRO setting, that basically says that the scheme can only be secure if the file identifier and the hash are sufficient to efficiently recover the file corresponding to the identifier. This results in a trivial scheme, as anyone can efficiently obtain knowledge about any file in L.

Lemma 3.2. *Consider the scheme of Section 3.1 in the GRO setting. There exists an input generation algorithm I and a dishonest prover strategy $(\hat{P}_1, \hat{P}_2)$ for which the extraction problem of Figure 2 is at least as hard as the extraction problem that must recover the file F only given its identifier* id *and its hash h and with access to the setup.*

Poof Sketch. Consider the following input generation algorithm $I^{\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{I},\cdot,\cdot)}$ for the agree-and-prove scenario from above: I (id, F_{id}) and the corresponding hash $h \leftarrow H[F_{\text{id}}]$. It defines $aux_p \leftarrow (\text{id}, h)$. It is clear that a dishonest prover $\hat{P}_1$ can convince V_1 by sending (id, h). To conclude the attack, we simply let $\hat{P}_2 := P_2$ (i.e. simply halts). Note that any extractor E in the security game is hence called on input $(1^\kappa, \text{id}, (\text{id}, h), (\text{id}, h))$. To conclude the statement, note that no setup-query is made by $\hat{P}_1$ and $\hat{P}_2$ simply halts. This means that the prover-oracle can be replaced by a local sub-routine. We can therefore construct a simpler extractor $\tilde{E}^{\mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{P},\cdot,\cdot)}(1^\kappa, aux_p)$ that simply executes E on input $(1^\kappa, \text{id}, (\text{id}, h), (\text{id}, h))$ and emulates the prover-oracle access towards E. E and E' have the same efficiency and the output distribution given the file identifier id and the hash h are identical. ☐

Note that the above stated provable insecurity reflects practice. For instance, consider the case where a cloud storage provider uses an proof-of-ownership protocol to perform client side deduplication based on above protocol with a standard hash function. In this setting, malicious parties can covertly abuse it as a file sharing platform by only having to exchange small hash values, with which they can then download the entire file from the cloud storage prover, as for instance pointed out by Mulazzani et al. [MSL+11].

3.2.2 A Secure Alternative

The natural way to obtain a secure protocol in this setting, is to let the prover prove his knowledge in the second phase. While one way to do so would be to simply send the entire file, we consider here a more efficient protocol proposed in [HHPSP11]. We also extend the structure of the protocol to be privacy preserving in the next section.

The scheme. In the agreement phase, the prover still sends the identity to the verifier. In the verification phase both prover and verifier first encode F using an erasure code to obtain $X = E(F)$ and then split X into blocks of size b. The verifier then chooses uniformly at random a subset (of size n) of the blocks for which the prover has to demonstrate knowledge. We assume here an erasure code (E, D) as of Appendix A.2 which can restore the original data item, as long as at most an α fraction of the symbols of the encoding X are missing, for some fixed $\alpha \in (0, 1)$.

Instead of simply sending those blocks, the protocol makes use of a Merkle-Tree as of Appendix A.1. That is, both the prover and verifier already compute $X = E(F)$ in the agreement phase, and then calculate the Merkle-Tree using $\mathsf{GenMT}^h(X, b)$ (where we assume here h be a random oracle for sake of simplicity). The prover then additionally sends the root value and the number of leaves ℓ of a Merkle-Tree along with the file identifier, and the verifier only accepts the statement if they match. During the second phase, the verifier can then check the correctness of the blocks by only using the control information consisting of the root and the number of leaves of the tree (instead of the entire file), thereby keeping its state small at the expense of some communication overhead.

A formal description of the corresponding prover and verifier protocols is given in Figure 5.

Analysis. Assume there is a prover who knows less than a $1 - \alpha$ fraction of the blocks, and thus cannot recover it with the erasure code. If we ask this prover to send us block b_i, for an i chosen uniformly at random by the verifier, then we will catch him with probability at least α. So if we ask him for a uniformly drawn subset of n blocks, we catch him with probability $1 - (1 - \alpha)^n$. We make this intuition precise, building on results from [HHPSP11], in the following security statement and its proof.

Theorem 3.3. *The described agree-and-prove scheme, specified as pseudo-code in Figure 5, for the scenario capturing proof-of-ownership with a global RO is secure up to soundness error* $p(\kappa) := (1 - \alpha)^{n(\kappa)}$.

Prover Protocols P_1 and P_2

Parameters: Size of Merkle-Tree leafs b and number of challenges n.

Prover $P_1(1^\kappa, aux_p)$**:**

1: Parse aux_p as (id, F). Abort if not possible.

2: Compute
 - $X \leftarrow E(F)$
 - $(T, \ell) \leftarrow \mathsf{GenMT}^{\mathcal{O}_{\mathcal{F}_{\mathsf{DB,GRO}}}(\mathsf{P}, \mathtt{ROeval}, \cdot)}(X, b)$.

 and let v_{root} be the root of T.

3: Send $(\mathrm{id}, (v_{\mathrm{root}}, \ell))$ to V_1.

4: Upon receiving ok from V_1, define the state $st_p \leftarrow T$ and statement $x = \mathrm{id}$, otherwise, set $st_p \leftarrow \mathsf{Fail}$ and $x := \perp$.

Prover $P_2(1^\kappa, st_p)$**:**

1: If $st_p = \mathsf{Fail}$, output 0 and halt.

2: Upon receiving the message containing a set $I = \{i_1, \ldots, i_n\} \subseteq [\ell]$ of leaf indexes, do the following:
 - $\forall j \in [n] : p_{i_j} \leftarrow \mathsf{MTGetPath}(T, i_j)$.
 - Send $(p_{i_1}, \ldots, p_{i_n})$ to V_2. Output 1 and halt.

Verifier Protocols V_1 and V_2

Parameters: Size of Merkle-Tree leafs b and number of challenges n.

Verifier $V_1(1^\kappa, aux_v)$**:**

1: Upon receiving a message $(\mathrm{id}, (v', \ell'))$ from P_1, obtain $F \leftarrow \mathcal{O}_{\mathcal{F}_{\mathsf{DB,RO}}}(\mathsf{V}, \mathtt{getFile}, \mathrm{id})$ and compute
 - $X \leftarrow E(F)$ and
 - $(T, \ell) \leftarrow \mathsf{GenMT}^{\mathcal{O}_{\mathcal{F}_{\mathsf{DB,GRO}}}(\mathsf{V}, \mathtt{ROeval}, \cdot)}(X, b)$.

 if $F \neq \perp$ and let v_{root} be the root of T (and $\perp$ otherwise).

2: If $v_{\mathrm{root}} \neq \perp$ and $v' = v_{\mathrm{root}}$ and $\ell' = \ell$, set $st_v \leftarrow (v_{\mathrm{root}}, \ell)$, send ok to P_1, and define the statement-state pair $(x = \mathrm{id}, st_v)$.
 In any other case, set $x := \perp$ and send x to P_1.

Verifier $V_2(1^\kappa, st_v)$**, assume** $x \neq \perp$**:**

1: Choose uniformly at random a subset $I = \{i_1, \ldots, i_n\} \subseteq [\ell]$ of n leaf indexes and send I to P_2.

2: Upon receiving siblings paths $(p_{i_1}, \ldots, p_{i_n})$ do:
 - If $\forall j \in [n] : \mathsf{MTVerifyPath}(v_{\mathrm{root}}, \ell, p_{i_j})$ returns 1, then output 1. Output 0 in any other case. Halt.

Figure 5: The description of the prover protocols (left) and the verifier protocols (right) of the Merkle-Tree based scheme.

Proof Sketch. Correctness is clearly satisfied. Security, in a nutshell, follows by a standard hardness amplification argument, i.e., if the prover provides valid Merkle paths for n uniformly at random chosen leaves with probability greater than $(1-\alpha)^n$, then the same prover will essentially provide a valid path for a single one with at least probability $1-\alpha$. Or phrased differently, he provides a valid path for at least an $1-\alpha$ fraction of the leaves. For this reasoning to be made precise, we can rely on the established fact stated in Lemma A.1. This means that our overall extractor first lets the extractor program $K'^{P_2}(v, \ell, n)$—which is guaranteed by Lemma A.1— run on input the Merkle-tree root v and the number of leaves ℓ from the transcript T, as well as the protocol parameter n (number of leaf indexes queried), until K' succeeds in restoring an $(1-\alpha)$ fraction X' of the encoding of $E(F)$ (matching the control information v,ℓ). We finally apply and output $F \leftarrow D(F')$ if F is a valid witness for the statement $x = \mathrm{id}$.

The security and runtime bounds of our overall extractor follows from (1) the guarantees given in Lemma A.1, in particular that an $(1-\alpha)$ fraction of the leaves is restored in an expected number of steps $\mathtt{T}_U$ that is inverse proportional to the soundness gap δ, (2) the assumed erasure code that allows to decode the obtained output of E to F, since an $(1-\alpha)$ fraction gets restored of $X = E(F)$, where X is the underlying data of the Merkle-Tree, and (3) the hash-collision property, which is information theoretic in the case of an RO, which guarantees that the restored file F is the file with identifier id and Merkle-root v except with negligible probability. □

3.3 On Including Privacy and Zero-Knowledge

Consider a company that wants to use an external cloud provider for file storage. To protect the confidentiality of their trade secrets they most likely want to opt for client side encryption of all the files. As naturally each file might be distributed among many employees, and the provider charges for the overall storage requirement, file deduplication is highly desirable. While all employees (or at least certain subgroups) might share the same key, coordinating on the randomness used to encrypt

each file is not practical and deterministic encryption does often not provide the required level of security. Thus, neither server-side deduplication nor the naive client-side deduplication on the ciphertext are feasible.

In this section, we provide a private version of the proof-of-ownership scheme that enables client-side deduplication in this setting, if the cloud provider explicitly supports so. The goal is that the storage provider should not be required to be trusted, and thus essentially learn nothing during the protocol run. At the same time, the storage provider should only provide access to the files to those users that already possess it, thereby preventing a rogue employee from just downloading all of the company's files. The basic idea of the protocol is that we keep the overall structure of the previous protocol, but patch it using encryption and NIZKs.

Setup. The setup corresponds to a snapshot of the system at the moment where a user wants to run the protocol. That is, it contains a list of encrypted files indexed by their respective identifiers (where the files can again be chosen by the input-generation algorithm), which have already been uploaded, together with the corresponding control information needed to run the protocol. The control information consists of an ElGamal encrypted Merkle root of the plaintext (an unencrypted root would allow the server to test whether a file is equal to a given bit-string), the number of leaves in the tree, as well as a signature binding the control information to the file identifier.

The verifier can access the encrypted files, the control information, as well as the public ElGamal key and the signature verification key. The setup either provides the prover access to the public keys only (modeling an outsider), or additionally to the symmetric key, the ElGamal decryption key, and the signing key (modeling an insider). Finally, the setup also provides the necessary CRS for the NIZK proofs to all parties. See Figure 6. Looking ahead, we will assume that the setup be programmable (to program the CRS).

File-Ownership setup $\mathcal{F}_{\mathsf{priv}}$

- The setup is (implicitly) parametrized by a cryptographic hash-function familiy $\mathcal{H}$, an erasure code (E, D), the leaf-size b for the Merkle-Tree, a symmetric encryption scheme SE, the ElGamal encryption scheme $\mathsf{ElGamal}$, and four associated NIZK proof systems.

- init:
  1: $\mathsf{KeysAssigned} \leftarrow \mathsf{false}$
  2: Choose $h \twoheadleftarrow \mathcal{H}$
  3: $k^{\mathsf{SE}} \leftarrow \mathsf{SE.Gen}(1^\kappa)$, $(ek^{\mathsf{ElGamal}}, dk^{\mathsf{ElGamal}}) \leftarrow \mathsf{ElGamal.Gen}(1^\kappa)$, $(vk^{\mathsf{Sig}}, sk^{\mathsf{Sig}}) \leftarrow \mathsf{Sig.Gen}(1^\kappa)$
  4: $(crs^{pt}, crs^{con}, crs^{mt,h}) \leftarrow (\mathsf{NIZK}^{pt}.\mathsf{Gen}(1^\kappa), \mathsf{NIZK}^{con}.\mathsf{Gen}(1^\kappa), \mathsf{NIZK}^{mt,h}.\mathsf{Gen}(1^\kappa))$

- $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{I}, \mathtt{defineDB}, L')$: If L is the empty array then do the following: if L' is an array of pairs $(\mathrm{id}_i, F_{\mathrm{id}_i})_{i\in[n]}$, where $\mathrm{id}_i, F_{\mathrm{id}_i} \in \{0,1\}^*$ and for all i, id_i is unique in L', then set $L \leftarrow L'$. In any case, output L.
  Once L is defined, for $(\mathrm{id}, F_{\mathrm{id}}) \in L$ do:
  - $X_{\mathrm{id}} \leftarrow \mathsf{SE.Enc}(k, F_{\mathrm{id}})$
  - $(T_{\mathrm{id}}, \ell_{\mathrm{id}}) \leftarrow \mathsf{GenMT}^h(E(F_{\mathrm{id}}), b)$. Let $v_{\mathrm{root,id}}$ be the root of T_{id}.
  - $c_{\mathrm{root,id}} \leftarrow \mathsf{ElGamal.Enc}(ek^{\mathsf{ElGamal}}, v_{\mathrm{root,id}})$.
  - $\sigma_{\mathrm{id}} \leftarrow \mathsf{Sig.Sgn}(sk^{\mathsf{Sig}}, (\mathrm{id}, c_{\mathrm{root,id}}, \ell_{\mathrm{id}}))$
  - $D[\mathrm{id}] \leftarrow (X_{\mathrm{id}}, c_{\mathrm{root,id}}, \ell_{\mathrm{id}}, \sigma_{\mathrm{id}})$.

- $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{I}, \mathtt{assignKeys}, -)$: Set $\mathsf{KeysAssigned} \leftarrow \mathsf{true}$.

- $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{V}, \mathtt{getFile}, \mathrm{id})$: If $\mathrm{id} \in L$ then return $D[\mathrm{id}]$. Otherwise, return $\perp$.

- $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{P}, \mathtt{getKey}, -)$: Return $(k^{\mathsf{SE}}, dk^{\mathsf{ElGamal}}, sk^{\mathsf{Sig}})$ if $\mathsf{KeyAssigned}$. Otherwise, return $\perp$.

- $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{i}, \mathtt{getPub}, -)$: Return the description of h, $(crs^{pt}, crs^{eq}, crs^{mt,h}, crs^{mt,h})$, and $(ek^{\mathsf{ElGamal}}, vk^{\mathsf{Sig}})$

Figure 6: The setup for the privacy-preserving file-ownership setting.

Verifier Leakage $\mathcal{L}_V$

- init: queried ← false
- $\mathcal{L}_V^{\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{P},\cdot,\cdot),\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{V},\cdot,\cdot)}(1^\kappa, aux_v, query)$:

1: If queried, return $\perp$, else set queried ← true and continue.

2: Parse *query* as (id, v, dk). Return $\perp$ if not possible.

3: Obtain $(k^{\mathsf{SE}}, dk^{\mathsf{ElGamal}}, sk^{\mathsf{Sig}})$ via a call $\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{P}, \mathtt{getKey}, -)$. Return $\perp$ if not possible.

4: Call $D \leftarrow \mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{V}, \mathtt{getFile}, \text{id})$.
  - If $D = \perp$, return 0.
  - Else, parse D as $(X_{\text{id}}, c_{\text{root,id}}, \ell_{\text{id}}, \sigma_{\text{id}})$. Let $v_{\text{root,id}} \leftarrow \mathsf{ElGamal.Dec}(dk, c_{\text{root,id}})$ and return 1 if $v = v_{\text{root,id}}$. Return 0 otherwise.

Prover Leakage $\mathcal{L}_P$

- init: –
- $\mathcal{L}_P^{\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{P},\cdot,\cdot),\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{V},\cdot,\cdot)}(1^\kappa, aux_p, query)$:

1: Parse aux_p as (id, F) and get $(k^{\mathsf{SE}}, dk^{\mathsf{ElGamal}}, sk^{\mathsf{Sig}})$ by calling $\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{P}, \mathtt{getKey}, -)$. Return aborted if either one is not possible. Continue otherwise.

2: Compute $(T, \ell) \leftarrow \mathsf{GenMT}^h(E(F), b)$ and let v_{root} denote the root of T.

3: Obtain $(X_{\text{id}}, C', \ell', \sigma)$ from $\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\mathsf{V}, \mathtt{getFile}, \text{id})$
  - If this fails or $\ell \neq \ell'$, return $(\text{id}, \ell, 0)$
  - Else, $v'_{\text{root}} \leftarrow \mathsf{ElGamal.Dec}(dk^{\mathsf{ElGamal}}, C')$ and let $a := (v_{\text{root}} = v'_{\text{root}})$. Return (id, ℓ, a).

Figure 7: The description of leakage a dishonest prover can obtain about the verifier's view (left) and leakage a dishonest verifier can obtain about the prover's view (right), in a single run of the protocol.

Agreement and Proof Relations. The statement that prover and verifier agree on is simply the analogous statements from the previous section, i.e., the relation is defined via the condition

$$C^{\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\cdot,\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x) = 1 \;:\leftrightarrow\; x = \perp \vee \exists i : L(i) = (x, \cdot), \tag{3}$$

Accordingly, the proof relation is defined via the condition

$$\mathcal{R}^{\mathcal{O}_{\mathcal{F}_{\text{priv}}}(\cdot,\cdot,\cdot)}(1^\kappa, x, w) = 1 :\leftrightarrow (x, w) \in L \wedge \mathsf{KeysAssigned}, \tag{4}$$

where it is additionally checked that the prover not only knows the file but it also has the necessary keys. As before, both predicates can be efficiently evaluated using the available oracles.

The privacy. Assume a prover and verifier execute a privacy-preserving proof-of-ownership scheme. Clearly at the end of a successful protocol run, the verifier will have learned whether the prover had a file which was already present in his database. More specifically, he will learn which identifier this file had, which appears to be inevitable if we were to use the protocol to handle client-side deduplication for cloud storage. Analogously, the prover will learn whether for his input (id, F) it held that $F = F_{\text{id}}$, which also seems necessary in a setting where he needs to upload the entire file otherwise.

In the remainder of the section we design a privacy preserving version of the previous Merkle-Tree based scheme. Let us briefly discuss the implication of sticking to this overall structure on privacy. In the agreement phase of the previous protocol, the prover sent the identity together with the root of the Merkle-Tree. The verifier would accept if and only if he has a file with the corresponding identity that has the same root, and only in the proof phase the prover had to show that he knows the entire file. In our scheme, a dishonest prover that has the decryption key dk will be able to learn whether for an identifier id and a Merkle-Tree root v_{root} of his choice, there exists a file F_{id} with root v_{root}, leaking slightly more than an optimal protocol. A formal definition of the leakage machine $\mathcal{L}_V$ can be found in Figure 7.

On the other side, in our protocol a dishonest verifier will learn the file identifier the prover has (if the prover has a file and the decryption key) and also the length of the prover's file (the number of Merkle leaves). Furthermore, if a file with this identifier exists in his database, then he will also learn whether it is the same one. A formal definition of the leakage machine $\mathcal{L}_P$ can be found in Figure 7 (note that in this case *query* is merely a trigger-input to obtain the leakage).

The scheme. The scheme basically follows the approach of the previous scheme of using a Merkle tree (in this section, we assume a collision-resistant hash function h and not a random oracle), however encrypts all the nodes of the tree using ElGamal encryption and then proves the consistency using NIZK proofs. See Figure 8 for the formal description of the scheme. In the following, let $G = \langle g \rangle$ be a cyclic group of prime order q with a generator g, in which the decisional Diffie-Hellman assumption is assumed to hold, and let $h\colon G^2 \to G$ be a collision resistant function.

We first describe the agreement phase. While in the original protocol the prover sends the identity id to the server together with the root of the Merkle tree, in the privacy preserving scheme he sends the identity alongside a fresh ElGamal encryption $(c_0, c_1) := (g^r, g^{dkr} \cdot v_{\text{root}})$ of the root. If the verifier has a file with that identity, then they proceed to check whether it encrypts the same root as the corresponding one from the verifier's control information $(c_0', c_1') := (g^s, g^{dks} \cdot v'_{\text{root}})$, i.e, whether $v'_{\text{root}} = v_{\text{root}}$. To this end, the verifier chooses $t \in \mathbb{Z}_q^*$ uniformly at random and sends back $(d_0, d_1) := \left(g^{t(s-r)}, g^{t\cdot dk(s-r)} \cdot (v'_{\text{root}} \cdot v_{\text{root}}^{-1})^t\right)$ obtained from dividing the two encryptions. Note that t is used to blind the verifier's Merkle tree root, which would otherwise leak to the prover knowing dk. The prover can then check whether $(v'_{\text{root}} \cdot v_{\text{root}}^{-1})^t = 1$ by raising the first element by the decryption key dk, and inform the verifier accordingly. Observe that since G is of prime order, we have that $x^t = 1$, for $t \in \mathbb{Z}_q^*$, if and only if $x = 1$, and thus the prover's check succeeds if and only if $v'_{\text{root}} = v_{\text{root}}$. If the verifier does not have a file with identifier id, then he chooses $d_0 \in G$ and $t \in \mathbb{Z}_q^*$ uniformly at random and sends $(d_0, (d_0)^t)$ instead, to conceal this fact. With overwhelming probability $t \neq dk$ and, thus, the prover will abort assuming that the Merkle roots don't match.

To protect against dishonest behaviors, during the agreement phase, both parties additionally prove with each message that is has been computed correctly using a NIZK proof for the languages introduced below, which are parametrized in (a description of) the group G, the generator g, the group order q, the file identifier space $\mathcal{ID}$, and the signature scheme Sig including the verification-key space $\mathcal{VK}$ and the signature space Σ.

- For the first message from the prover to the verifier, let NIZK^{dk} be a NIZK proof system for the language $L^{dk} := \{x \mid \exists w\ (x, w) \in R^{pt}\}$, where R^{dk} is defined as follows: for $x = ek \in G$ and a witness $w = dk \in \mathbb{Z}_q$, $R^{dk}(x, w) = 1$ if and only if

  $$ek = g^{dk}.$$

  Hence, the prover shows that he knows the decryption key, and thus also the corresponding plaintext v_{root} of his first message.

- For the message from the verifier to the prover, let NIZK^{con} be a NIZK proof system for the language $L^{con} := \{x \mid \exists w\ (x, w) \in R^{con}\}$, where R^{con} is defined as follows: for $x = \left(c_0, c_1, d_0, d_1, \text{id}, \ell, vk\right) \in G^4 \times \mathcal{ID} \times \mathbb{N} \times \mathcal{VK}$ and a witness $(c_0', c_1', t, \sigma) \in G^2 \times \mathbb{Z}_q \times \Sigma$, $R^{con}(x, w) = 1$ if and only if

  $$\Big((d_0, d_1) = \big((c_0' \cdot c_0^{-1})^t, (c_1' \cdot c_1^{-1})^t\big) \wedge \mathsf{Sig.Vrf}(vk, \sigma, (\text{id}, c_0', c_1', \ell))\Big) \vee (d_0)^t = d_1.$$

- For the second message from the prover to the verifier, let NIZK^{eq} be a NIZK proof system for the language $L^{eq} := \{x \mid \exists w\ (x, w) \in R^{eq}\}$, where R^{eq} is defined as follows: for $x = \left(ek, d_0, d_1\right) \in \mathbb{Z}_q \times G^2$ and a witness $w = dk \in \mathbb{Z}_q$, $R^{eq}(x, w) = 1$ if and only if

  $$(ek, d_1) = (g^{dk}, d_0^{dk}).$$

Finally, in the prove-phase, the server selects again a number of leaf indexes and the prover replies with the encrypted siblings path together with NIZK's to prove that the path is correctly built, defined as follows.

Prover Protocols P_1 and P_2

Parameters: Size of Merkle-Tree leafs b and number of challenges n.

Prover $P_1(1^\kappa, aux_p)$**:**

1: Parse aux_p as (id, F). Abort if not possible.

2: Obtain h, $(crs^{dk}, crs^{con}, crs^{eq}, crs^{mt,h})$, and $(ek^{\mathsf{ElGamal}}, vk^{\mathsf{Sig}})$ from $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{P}, \mathtt{getPub}, -)$. Also, obtain $(k^{\mathsf{SE}}, dk^{\mathsf{ElGamal}}, sk^{\mathsf{Sig}})$ by calling $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{P}, \mathtt{getKey}, -)$. Abort if not possible (send $st_p := \mathsf{Fail}$ to V_1 and set $x := \perp$).

3: Compute
- $X \leftarrow E(F)$
- $(T, \ell) \leftarrow \mathsf{GenMT}^h(X, b)$,

and let v_{root} be the root of T.

4: Let $C \leftarrow \mathsf{ElGamal.Enc}(ek^{\mathsf{ElGamal}}, v_{\mathrm{root}})$ and $\pi^{dk} \leftarrow \mathsf{NIZK}^{dk}.\mathsf{Prove}\big(crs^{dk}, ek^{\mathsf{ElGamal}}, dk^{\mathsf{ElGamal}}\big)$ Send $(\mathrm{id}, C, \pi^{ek}, \ell)$ to V_1.

5: Upon receiving $D = (d_0, d_1)$ from V_1,
- If $\mathsf{NIZK}^{con}.\mathsf{Ver}\big(crs^{con}, (C, D, \mathrm{id}, \ell, vk^{\mathsf{Sig}}), \pi^{con}\big) = 0$ or $d_1 \neq (d_0)^{dk^{\mathsf{ElGamal}}}$ then set $st_p \leftarrow \mathsf{Fail}$ and $x := \perp$.
- Else, compute $\pi^{ek} \leftarrow \mathsf{NIZK}^{eq}.\mathsf{Prove}\big(crs^{eq}, (ek^{\mathsf{ElGamal}}, d_0, d_1), dk^{\mathsf{ElGamal}}\big)$, send π^{eq} to V_1, and set $st_p \leftarrow T$ and $x := \mathrm{id}$.

6: Define st_p and x if V_1 returns ok (and $st_p \leftarrow \mathsf{Fail}$ and $x := \perp$ otherwise).

Prover $P_2(1^\kappa, st_p)$**, assume** $st_p \neq \mathsf{Fail}$**:**

1: Upon receiving the message containing a set $I = \{i_1, \ldots, i_n\} \subseteq [\ell]$ of leaf indexes, compute $\forall j \in [n] : p_{i_j} \leftarrow \mathsf{MTGetPath}(T, i_j)$ and do for each such sibling path p_{i_j}:
- Parse $p_{i_j} = (v_{i_j,1}, \ldots, v_{i_j,k})$ as sequence of nodes (starting from the root and with at each level the left node being first), and let the encrypted sequence be $(c_{i_j,1}, \ldots, c_{i_j,k})$, where $c_{i_j,1} := C$ and $c_{i_j,t} \leftarrow \mathsf{ElGamal.Enc}(ek^{\mathsf{ElGamal}}, v_{i_j,t})$ for $2 \leq t \leq k$.
- For each non-leaf level $1 \leq d \leq (k-1)/2$, compute $\pi_{i_j,d} \leftarrow \mathsf{NIZK}^{mt,h}.\mathsf{Prove}\big(crs^{mt,h}, (ek^{\mathsf{ElGamal}}, n_d, l_d, r_d), dk^{\mathsf{ElGamal}}\big)$, where n_d denotes the node on the d-th level on which the path descends, and l_d and r_d is left and right children, respectively.
- Send $(\pi_{i_j,1}, \ldots, \pi_{i_j,(k-2)/2})$ and $(c_{i_j,1}, \ldots, c_{i_j,k})$ to V_2 and output 1. Halt.

Verifier Protocols V_1 and V_2

Parameters: Size of Merkle-Tree leafs b and number of challenges n.

Verifier $V_1(1^\kappa, aux_v)$**:**

1: Upon receiving a message $(\mathrm{id}, C, \pi^{dk}, \ell)$ from P_1, obtain h, $(crs^{dk}, crs^{con}, crs^{eq}, crs^{mt,h})$, and $(ek^{\mathsf{ElGamal}}, vk^{\mathsf{Sig}})$ from $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{P}, \mathtt{getPub}, -)$. Then, check $\mathsf{NIZK}^{dk}.\mathsf{Ver}\big(crs^{dk}, ek^{\mathsf{ElGamal}}, \pi^{dk}\big)$, and if this fails (or P_1 aborted) then set $x := \perp$ and send $\perp$ to P_1. Otherwise, continue.

2: Choose $t \twoheadleftarrow \mathbb{Z}_q^*$ uniformly at random and retrieve $(X_{\mathrm{id}}, C', \ell', \sigma)$ via a call to $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{V}, \mathtt{getFile}, \mathrm{id})$.
- If the file does not exist, or if $\ell \neq \ell'$, then choose $d_0 \twoheadleftarrow G$ u.a.r., set $D := \big(d_0, (d_0)^t\big)$, and compute $\pi^{con} \leftarrow \mathsf{NIZK}^{con}.\mathsf{Prove}\big(crs^{con}, (C, D, \mathrm{id}, \ell, vk^{\mathsf{Sig}}), ((1,1), t, \sigma_\perp)\big)$, for $\sigma_\perp \in \Sigma^{\mathsf{Sig}}$ arbitrary.
- Else, parse C as (c_0, c_1) and C' as (c'_0, c'_1) and compute $D := \big((c'_0/c_0)^t, (c'_0/c_0)^t\big)$. Additionally, compute $\pi^{con} \leftarrow \mathsf{NIZK}^{con}.\mathsf{Prove}\big(crs^{con}, (C, D, \mathrm{id}, \ell', vk^{\mathsf{Sig}}), (C', t, \sigma)\big)$.

Send (D, π^{con}) to P_1.

3: Upon receiving π^{eq}, verify it by evaluating $b \leftarrow \mathsf{NIZK}^{eq}.\mathsf{Ver}\big(crs^{eq}, (ek^{\mathsf{ElGamal}}, D), \pi^{eq}\big)$. If $b = 1$ set $st_v \leftarrow (C, \ell)$, send ok to P_1, and output $(x = \mathrm{id}, st_v)$ (otherwise, $x := \perp$ and define Fail to P_1).

Verifier $V_2(1^\kappa, st_v)$**, assume** $x \neq \perp$**:**

1: Choose uniformly at random a subset $I = \{i_1, \ldots, i_n\} \subseteq [\ell]$ of n leaf indexes and send I to P_2.

2: Upon receiving n sibling paths $(c_{i_j,1}, \ldots, c_{i_j,k})$ and associated proofs $(\pi_{i_j,1}, \ldots, \pi_{i_j,(k-2)/2})$, for each $j \in [n]$ do:
- Check that $c_{i_j,1} = C$
- For each non-leaf level $1 \leq d \leq (k-1)/2$ check $\mathsf{NIZK}^{mt,h}.\mathsf{Ver}\big(crs^{mt,h}, (ek^{\mathsf{ElGamal}}, n_d, l_d, r_d), \pi_{i_j,d}\big)$, where n_d denotes the node on the d-th level on which the path descends, and l_d and r_d is left and right children, respectively.

3: If and only if all tests succeed, output 1. In any other case, output 0. Halt.

Figure 8: The description of the prover protocols (left) and the verifier protocols (right) for the privacy setting.

- Let the language $L^{mt,h} := \{x \mid \exists w\ (x,w) \in R^{mt,h}\}$ be defined via the following relation $R^{mt,h}$: for $x = \big(ek, n_0, n_1, l_0, l_1, r_0, r_1\big) \in G^7$ and a witness $w = dk \in \mathbb{Z}_q$, $R^{mt,h}(x,w) = 1$ if and only if

$$ek = g^{dk}$$
$$\wedge\ \mathsf{ElGamal.Dec}(dk,(n_0,n_1)) = h\big(\mathsf{ElGamal.Dec}(dk,(l_0,l_1)), \mathsf{ElGamal.Dec}(dk,(r_0,r_1))\big).$$

The verifier furthermore checks that in each path, the ciphertext of the root is the one the prover sent in the agreement phase.

Analysis. The described agree-and-prove protocol achieves the same level of security as the plain Merkle-Tree based protocol, analyzed in the last section, but additionally provides the described level of privacy. This is summarized in the following theorem.

Theorem 3.4. *The agree-and-prove scheme from Figure 8, for the agree-and-prove scenario consisting of the setup functionality from Figure 6 and the relations from equations (3) and (4), is secure up to knowledge error $p(\kappa) := (1-\alpha)^{n(\kappa)}$.*

Furthermore, it is verifier zero-knowledge up to $\mathcal{L}_P$ and prover zero-knowledge up to $\mathcal{L}_V$, where $\mathcal{L}_P$ and $\mathcal{L}_V$ are both defined in Figure 7.

For ease of presentation, we split the proof into its separate properties. First, we show that it achieves the same level of security as the plain protocol.

Lemma 3.5. *The agree-and-prove scheme from Figure 8, for the agree-and-prove scenario consisting of the setup functionality from Figure 6 and the relations from equations (3) and (4), is secure up to knowledge error $p(\kappa) := (1-\alpha)^{n(\kappa)}$.*

Poof Sketch. Correctness is again trivially satisfied as the verifier V_1 only agrees on a file identifier id that is in the database.

Soundness follows along the same lines as in Theorem 3.3. We here only sketch how the soundness of the scheme with privacy can be reduced to the soundness of the basic scheme. First, consider the agreement phase. In the basic scheme, a the verifier only agrees on a statement if the prover sent a $(\mathrm{id}, v_{\mathrm{root}}, \ell)$ triple that matches his file F_{id}. We now first show that the same also holds in the scheme with privacy, and that the extractor learns $(\mathrm{id}, v_{\mathrm{root}}, \ell)$. To this end, consider the following setup generation algorithm SGen: it first runs E_1^{pt} from the knowledge extractor E^{pt} to obtain a CRS and the corresponding trapdoor. Then, it outputs the description of a setup functionality $\mathcal{F}'_{\mathsf{priv}}$ that works the same as $\mathcal{F}_{\mathsf{priv}}$ except that crs^{pt} is replaced by the one obtained from E_1^{pt}. Moreover, it outputs the CRS trapdoor as td. The extractor for the agree-and-prove scheme can then use td to extract dk^{ElGamal} from the NIZK proof that the prover initially has to send. Since we can extract the correct secret key, with overwhelming probability it also must hold that $\mathsf{KeysAssigned}$ is set to true, thus reducing the knowledge predicate to knowing the correct file. By the soundness of the NIZK proof for R^{eq} that the prover sends as the second message, we moreover know that V_1 only accepts if

$$g^{(s-r)t\,dk^{\mathsf{ElGamal}}} = g^{(s-r)t\,dk^{\mathsf{ElGamal}}} \cdot (v'_{\mathrm{root}}/v'_{\mathrm{root}})^t,$$

which, by $t \in \mathbb{Z}_q^*$ implies that $v'_{\mathrm{root}} = v'_{\mathrm{root}}$. In summary, V_1 only accepts if $\hat{P}_1$ sends the encryption of the same Merkle root, and the extractor furthermore knows this root. In the proof phase, we know, by the soundness of the NIZK for $L^{mt,h}$, that V_2 only accepts if and only if the Merkle proofs are correct. Hence, the extractor can internally run the one from the basic scheme, decrypting for him the nodes on the paths using dk^{ElGamal}, and thereby achieving the same success probability. □

Now, we prove that the protocol achieves the desired level of privacy for the honest prover, i.e., that a verifier cannot learn more whether the prover had a file with a matching Merkle root.

Lemma 3.6. *The agree-and-prove scheme from Figure 8, for the agree-and-prove scenario consisting of the setup functionality from Figure 6 and the relations from equations (3) and (4), is verifier zero-knowledge up to $\mathcal{L}_P$ as defined in Figure 7.*

Proof. We have to show that a dishonest verifier $\hat{V}$ cannot learn more than provided by the leakage oracle $\mathcal{L}_P$.

To this end, consider the following setup generation algorithm SGen: it first runs S_1^{dk}, S_1^{eq}, $S_1^{mt,h}$ and E_1^{con} to obtain the corresponding CRS crs^{dk}, crs^{eq}, $crs^{mt,h}$, and crs^{con}, and trapdoors τ^{dk}, τ^{eq}, $\tau^{mt,h}$ and ξ^{con}, respectively. Then, it outputs the description of a setup functionality $\mathcal{F}'_{\mathsf{priv}}$ that works the same as $\mathcal{F}_{\mathsf{priv}}$ except that it uses those CRS. Moreover, it outputs the trapdoor $td := (\tau^{dk}, \tau^{eq}, \tau^{mt,h}, \xi^{con})$.

Now, consider the following simulator $S_{\hat{V}}$ that internally emulated $\hat{V}$ and works as follows:

1. First, the simulator queries $\mathcal{L}_P$. If the return value is $\mathsf{aborted}$ it internally runs $\hat{V}$ without providing any further input. Otherwise, it obtained the leakage (id, ℓ, a) and continues.

2. It chooses a file $\tilde{F}$ and computes $(\tilde{T}, \tilde{\ell}) \leftarrow \mathsf{GenMT}^h(E(\tilde{F}), b)$, where the length of $\tilde{F}$ has been chosen such that $\tilde{\ell} = \ell$. Let $\tilde{v}_{\mathrm{root}}$ be the root of $\tilde{T}$.

3. For the first message of P_1, it computes $\tilde{C} \leftarrow \mathsf{ElGamal.Enc}(ek^{\mathsf{ElGamal}}, \tilde{v}_{\mathrm{root}})$ and $\tilde{\pi}^{ek} \leftarrow S_2(crs^{ek}, \tau^{ek}, ek^{\mathsf{ElGamal}})$. It then uses $(\mathrm{id}, \tilde{C}, \tilde{\pi}^{ek}, \ell)$ as the first message to $\hat{V}$.

   Observe that by the IND-CPA security of ElGamal, and the zero-knowledge property of the NIZK, this looks indistinguishable to $\hat{V}$ from the actual message from P_1.

4. The simulator receives the answer to the prover (D, π^{con}) from $\hat{V}$. It then runs the extractor $(c_0', c_1', t, \sigma) \leftarrow E_2^{con}(crs^{con}, \xi^{con}, (\tilde{C}, D, \mathrm{id}, \ell, vk^{\mathsf{Sig}}), \pi^{con})$.

   - If
     $$a = 1 \ \wedge\ (d_0, d_1) = \big((c_0' \cdot \tilde{c}_0^{-1})^t, (c_1' \cdot \tilde{c}_1^{-1})^t\big) \ \wedge\ \mathsf{Sig.Vrf}(vk^{\mathsf{Sig}}, \sigma, (\mathrm{id}, c_0', c_1', \ell)),$$
     then the simulator computes $\tilde{\pi}^{eq} \leftarrow S_2(crs^{eq}, \tau^{eq}, (ek^{\mathsf{ElGamal}}, D))$. If $a = 1$, then the simulator inputs $\tilde{\pi}^{eq}$ as the second message from P_1 to $\hat{V}$.
   - Else, the simulator inputs $\perp$ as the second message from P_1 to $\hat{V}$.

   We now argue that this message looks indistinguishable to $\hat{V}$ from the actual second message from P_1. First, assume that the extractor produced a witness satisfying the first condition. Then, by the unforgeability of the signature scheme C', is the verifier's correct encrypted root for id and moreover D has been computed correctly. Then, as previously seen in the soundness proof, the prover P_1 will send back a NIZK proof if and only if the two Merkle roots match, thus if and only if $a = 1$. If the extractor produced a witness not satisfying this condition, then it must satisfy $d_0^t = d_1$. Given the hardness of the discrete logarithm problem, we will have $t \neq dk^{\mathsf{ElGamal}}$ with overwhelming probability, as neither $\hat{V}$ nor the extractor know anything about dk^{ElGamal} beyond ek^{ElGamal}. Thus, with overwhelming probability P_1 would also answer $\perp$.

5. In the proof phase, the simulator will forge the Merkle proofs with respect to $\tilde{T}$. That is it will decrypt the appropriate nodes from $\tilde{T}$, and for each non-leaf level $1 \leq d \leq (k-1)/2$, it will generate $\tilde{\pi}_{i_j,d} \leftarrow S_2(crs^{mt,h}, \tau^{mt,h}, (ek^{\mathsf{ElGamal}}, n_d, l_d, r_d))$, where n_d denotes the node on the d-th level on which the path descends, and l_d and r_d is left and right children, respectively, in $\tilde{T}$.

   Again, by the IND-CPA security of ElGamal, and the zero-knowledge property of the NIZK, this looks indistinguishable to $\hat{V}$ from the actual message from P_2.

In summary, the simulator provides $\hat{V}$ inputs that look indistinguishable from the actual ones from P given the view of $\hat{V}$ and $\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{I}, \cdot, \cdot)$. □

Finally, we show that the protocol achieves the desired level of privacy for the honest verifier, i.e., that a prover (who has the decryption keys) cannot learn more whether for a given file identifier id and a root v_{root}, there is a matching entry in the database.

Lemma 3.7. *The agree-and-prove scheme from Figure 8, for the agree-and-prove scenario consisting of the setup functionality from Figure 6 and the relations from equations (3) and (4), is prover zero-knowledge up to $\mathcal{L}_V$ as defined in Figure 7.*

Proof. Consider the following setup generation algorithm SGen: it first runs E_1^{dk}, E_1^{eq}, $E_1^{mt,h}$ and S_1^{con} to obtain the corresponding CRS crs^{dk}, crs^{eq}, $crs^{mt,h}$, and crs^{con}, and trapdoors ξ^{dk}, ξ^{eq}, $\xi^{mt,h}$ and τ^{con}, respectively. Then, it outputs the description of a setup functionality $\mathcal{F}'_{\mathsf{priv}}$ that works the same as $\mathcal{F}_{\mathsf{priv}}$ except that it uses those CRS. Moreover, it outputs the trapdoor $td := (\xi^{dk}, \xi^{eq}, \xi^{mt,h}, \tau^{con})$.

Now, consider the following simulator $S_{\hat{P}}$ that internally emulated $\hat{P}$ and works as follows:

1. Upon obtaining $(\text{id}, C, \pi^{ek}, \ell)$ from $\hat{P}_1$ it verifies π^{ek}. If the verification fails, it only inputs $\perp$ to $\hat{P}_1$. Otherwise, it computes $dk^{\mathsf{ElGamal}} \leftarrow E_2^{dk}(crs^{dk}, \xi^{dk}, ek^{\mathsf{ElGamal}}, \pi^{eq})$, and $v_{\text{root}} \leftarrow \mathsf{ElGamal.Dec}(dk^{\mathsf{ElGamal}}, C)$. Then it queries $b \leftarrow \mathcal{L}_V^{\mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{P},\cdot,\cdot), \mathcal{O}_{\mathcal{F}_{\mathsf{priv}}}(\mathsf{V},\cdot,\cdot)}(1^\kappa, aux_v, (\text{id}, v, dk))$.

   Observe that since we managed to extract dk^{ElGamal}, the prover must have access to the keys (otherwise either the dishonest prover or the extractor solve the discrete logarithm problem) and, thus, $b \neq \perp$.

2. Then,

   - If $b = 1$, the simulator chooses $t \twoheadleftarrow \mathbb{Z}_q^*$ uniformly at random, sets $\tilde{D} := \big(g^t, (ek^{\mathsf{ElGamal}})^t\big)$ and computes $\tilde{\pi}^{con} \leftarrow S_2(crs^{con}, \tau^{con}, (C, \tilde{D}, \text{id}, \ell, vk^{\mathsf{Sig}}))$. Note that the statement is in the language, as setup has a signature such that $((g^s, g^{dk^{\mathsf{ElGamal}} s} \cdot v_{\text{root}}), t, \sigma)$ is a witness.
   - If $b = 0$, the simulator chooses $d_0 \twoheadleftarrow G$ and $t \twoheadleftarrow \mathbb{Z}_q^*$ uniformly at random, sets $D := (d_0, (d_0)^t)$, and computes $\tilde{\pi}^{con} \leftarrow \mathsf{NIZK}^{con}.\mathsf{Prove}\big(crs^{con}, (C, \tilde{D}, \text{id}, \ell, vk^{\mathsf{Sig}}), (C', t, \sigma)\big)$, for arbitrary C' and σ.

   The simulator then inputs $(\tilde{D}, \tilde{\pi}^{con})$ to $\hat{P}_1$.

3. Upon receiving π^{eq}, the simulator verifies the proof. If the proof verified and $b = 1$, then it inputs $\mathtt{ok}$ to $\hat{P}_2$, chooses $I = \{i_1, \ldots, i_n\} \subseteq [\ell]$ uniformly at random, and input I to $\hat{P}_2$.

It remains to show that the message $(\tilde{D}, \tilde{\pi}^{con})$ he inputs to $\hat{P}_1$ is indistinguishable by the one produced by V_1. Clearly, if no file with identifier id exists, or this file has a different number of leaves ℓ', then the simulator produces his answer exactly the same way as P_1. In case they do match, we need to consider two cases. First, assume that also the roots match, i.e, $v_{\text{root}} = v'_{\text{root}}$. The honest verifier then replies with $(g^{t(s-r)}, ek^{t(s-r)})$ for $t \twoheadleftarrow \mathbb{Z}_q^*$ and $s \twoheadleftarrow \mathbb{Z}_q$ uniformly at random. Independently of r, however, $t(s-r)$ is a uniform random element of $\mathbb{Z}_q$, which is indistinguishable from a $t \twoheadleftarrow \mathbb{Z}_q^*$ in a group of prime order. Second, consider the case that the roots don't match. In this case, the honest verifier V_1 replies with $(g^{t(s-r)}, g^{dk(s-r)t} \cdot (v'_{\text{root}}/v_{\text{root}})^t)$, whereas the simulator inputs $(d_0, (d_0)^t)$. In the following assume again that $t \twoheadleftarrow \mathbb{Z}_q$ (which is indistinguishable form $t \twoheadleftarrow \mathbb{Z}_q^*$). Observe now that $g^{t(s-r)}$ is a uniform random group element, independent of r and t and, thus, by the decisional Diffie-Hellman assumption we have

$$\big(dk, r, v_{\text{root}}, g^{t(s-r)}, g^{dk(s-r)t} \cdot (v'_{\text{root}}/v_{\text{root}})^t\big) \approx \big(dk, r, v_{\text{root}}, g^x, g^{dkx} \cdot (v'_{\text{root}}/v_{\text{root}})^t)\big)$$

for $x \twoheadleftarrow \mathbb{Z}_q$ uniformly at random. Since $v'_{\mathrm{root}}/v_{\mathrm{root}} \neq 1$ and thus a generator (in the group of prime order), we moreover have

$$\left(dk, r, v_{\mathrm{root}}, g^x, g^{dkx} \cdot (v'_{\mathrm{root}}/v_{\mathrm{root}})^t)\right) \approx \left(dk, r, v_{\mathrm{root}}, g^x, g^{dkx} \cdot g^y\right) \approx \left(dk, r, v_{\mathrm{root}}, g^x, g^{xy}\right)$$

for $x, y \twoheadleftarrow \mathbb{Z}_q$ chosen independently and uniformly at random. This in turn is indistinguishable from $(dk, r, v_{\mathrm{root}}, d_0, (d_0)^t)$ where $d_0 \twoheadleftarrow G$ and $t \twoheadleftarrow \mathbb{Z}_q^*$. □

This concludes the overall proof of Theorem 3.4.

4 Application to Client Authentication

In this section, we consider a different application of our agree-and-prove definition: client authentication. Client authentication is an integral part of any web service, where the server authenticates himself using the global certificate infrastructure, but clients are typically authenticated using passwords and, optionally, some second factor such as a hardware token.

For this reason, client authentication has gained a lot of attention from the security community, such as for instance [YWWD06, JK18, BHvOS12]. Furthermore, plenty of client authentication protocols have been proposed and studied over the years, such as [CJT02, YRY04, YY05, LLH06]. Those works however phrase security in a property based manner with rather particular attack models making them not directly applicable in an overall cryptographic analysis based on explicit hardness assumptions and reduction proofs. For instance, [YWWD06] phrases (among other) the desired property of *client authentication*, i.e., that the server must ensure that the communicating party is the registered client that claims to be at the end of the protocol, or the property that *Server Knows No Password*, i.e., that a server should not obtain information about the password.

Multi-factor authentication has also gotten some attention in the cryptographic community, e.g. the work by Shoup and Rubin on session-key distribution with smart cards [SR96], which arguably does not reflect the usual password plus second-factor based setting. To the best of our knowledge, there is no formal cryptographic model to analyze multi-factor authentication. In the following, we show how the agree-and-prove notion can be used to formalize the above mentioned properties in a sound and thorough manner.

4.1 Password-Based Authentication

We first consider the simple case of a client authenticating himself with a password exclusively to a server storing a corresponding login database. We first abstract this application as an agree-and-prove scenario which includes the setup, and the relation we want to prove.

Setup. The setup is parametrized in what we call a user-administration mechanism UAdmin, that allows a user to register to the service with a given password, or update is password, if the user already exists. Intuitively, such an algorithm abstracts away the maintenance of a login database. Note that this does not prescribe how login attempts are verified. For a given user-administration mechanism this will be the task of the associated agree-and-prove scheme.

Definition 4.1. A *user-administration mechanism* UAdmin consists of the following two PPT algorithms:

Initialization: The algorithm Init; on input a security parameter 1^κ, outputs an initial state db.

User registration: The algorithm Set; on input a state db, a username un, and a password pw, outputs a new state db'.

The setup then provides the input-generation algorithm the possibility to register users, and update their passwords, if desired. The verifier then gets the resulting database, to verify the login attempts. Moreover, to model potential intrusions, the input-generation algorithm can access the database as well, and to phrase the following relations it can also retrieve the list of all users with their passwords (which he set). The description can be found in Figure 9.

Password-Based Client-Authentication setup $\mathcal{F}_{\mathsf{PW\text{-}A}}$

- The setup is parameterized a user-administration mechanism UAdmin
- init: $db \leftarrow \mathsf{UAdmin.Init}(1^\kappa)$; initialize PW to an empty map.
- $\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\mathsf{I}, \mathtt{SetPassword}, un, pw)$: Set $db \leftarrow \mathsf{UAdmin.Set}(db, un, pw)$ and $\mathsf{PW}[un] \leftarrow pw$.
- $\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\mathsf{I}, \mathtt{GetUsers}, -)$: Return PW.
- $\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\mathsf{i}, \mathtt{GetDB}, -)$: If $i \in \{\mathsf{I}, \mathsf{V}\}$, return db. Otherwise, return $\perp$.

Figure 9: The generic setup functionality for a password-based client authentication scenario.

Agreement and Proof Relations. The obvious statement that a prover and verifier should agree on is a username which has been registered. More formally, the agreement relation is defined via the condition

$$C^{\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\cdot,\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x) = 1 \;:\leftrightarrow\; x = \perp \vee \mathsf{PW}[x] \text{ is defined,} \tag{5}$$

which can be efficiently implemented by C by calling $\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\mathsf{I}, \mathtt{GetUsers}, x)$.

The proof relation, on the other hand, guarantees that a prover who can convince the verifier of being a certain user must know the corresponding password. The relation is thus defined via the condition

$$\mathcal{R}^{\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\cdot,\cdot,\cdot)}(1^\kappa, x, w) = 1 \;:\leftrightarrow\; \mathsf{PW}[x] = w. \tag{6}$$

which again can be efficiently implemented by calling $\mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\mathsf{I}, \mathtt{GetUsers}, x)$.

4.1.1 A Simple Scheme

Consider the following naive user-administration mechanism $\mathsf{UAdmin}_{\text{plain}}$: The algorithm Init outputs an empty map $db \leftarrow []$. The algorithm Set on input and a state db, a username un, and a password pw does the following: If un already exists in db it overwrites the password; otherwise it creates a new entry $db(un) \leftarrow pw$. That is, it stores the passwords in plain. The scheme is described in Figure 10. The agreement phase consists of the prover stating its username, and the verifier checking in the login database whether such a user exists (aborting otherwise). In the verification phase, the prover sends his password in plain. The verifier accepts if and only if it matches the password in the database.

It is easy to see that this scheme achieves the required client authentication for $\mathsf{UAdmin}_{\text{plain}}$, as summarized in the following result.

Theorem 4.2. *The agree-and-prove scheme from Figure 10 for the agree-and-prove scenario defined, by $\mathsf{UAdmin}_{\text{plain}}$, $\mathcal{F}_{\mathsf{PW\text{-}A}}$ and the above described relations, is secure (with soundness error $p(\kappa) := 0$).*

Proof. It is trivial to see that the correctness relation is satisfied as the verifier accepts if and only if the username is in the database. Similarly, there exists a trivial knowledge extractor: it runs $\hat{P}_2$ and returns the sent password as the witness. □

Prover Protocols P_1 and P_2

Prover $P_1(1^\kappa, aux_p)$**:**

1: Parse aux_p as (un, pw). If this fails, set $st_p = \mathsf{Fail}$ and halt.

2: Send un to V_1.

3: Upon receiving ok define $st_p = pw$ and statement $x := un$. Otherwise, set $st_p = \mathsf{Fail}$ and $x := \perp$.

Prover $P_2(1^\kappa, st_p)$**:**

1: If $st_p = \mathsf{Fail}$, then halt with output 0. Otherwise first send st_p to V_2 and output whatever V_2 returns.

Verifier Protocols V_1 and V_2

Verifier $V_1(1^\kappa, aux_v)$**:**

1: Upon receiving a message un from P_1, obtain $db \leftarrow \mathcal{O}_{\mathcal{F}_{\mathsf{PW\text{-}A}}}(\mathsf{V}, \mathtt{GetDB}, -)$. Then do the following: , and otherwise set , $st_v \leftarrow \mathsf{ok}$. send ok to P_1 and output (x, st_v) the following:

- If $db(un)$ is defined, set $m \leftarrow \mathsf{ok}$, $st_v \leftarrow db(un)$, and statement $x \leftarrow un$.
- Else, set $m \leftarrow \mathsf{Fail}$, $st_v \leftarrow \mathsf{Fail}$, and statement $x \leftarrow \perp$.

Send m to P_1.

Verifier $V_2(1^\kappa, st_v)$**, assume** $x \neq \perp$**:**

1: Upon receiving pw from P_2, check whether $st_v = pw$. Output 1 if so, and 0 otherwise and send the decision to P_2.

Figure 10: The description of the prover protocols (left) and the verifier protocols (right) for the password-based authentication where the setup stores the passwords in plain.

4.1.2 On Reflecting further Aspects

We now briefly discuss how our basic agree-and-prove notion could be extended to account for additional security properties relevant in client authentication.

On protecting the passwords. The above scheme is slightly oversimplified, as an intruder breaking into the server can learn all the passwords and then successfully impersonate all users. Especially, it does not satisfy "Server Knows No Password" property from [YWWD06].

We would like to remark, however, that in our modeling this is a security property of the setup and thereby orthogonal to the privacy provided by the authentication protocol. As this, it could be simply expressed as an additional security property of the user-administration mechanism, where various variants appear plausible: one could either request that an adversary having access to the login database cannot guess a password (computational property), or that he cannot learn any information about the passwords (decisional property).

What our agree-and-prove notion can guarantee is that the verifier does not learn anything about the password (which is not already revealed by the database), during the execution of the protocol. For instance, consider a login database that stores a hash of the passwords. In the random oracle model this does not reveal more to the verifier than allowing him to verify password guesses.

On accounting for pre-processing. In practice, simply storing hashes of the password is not considered good practice. Passwords should rather be seeded with a separate randomly chosen seed for each password in order to thwart well-know pre-processing attacks such as rainbow tables [Hel80]. From a cryptographic point of view, such pre-computation attacks are best captured by considering the auxiliary-input random-oracle model [Unr07, DGK17], where the adversary can get a bounded amount of advise about the random oracle from a computationally unbounded entity.

To account for such pre-processing in our agree-and-prove framework, one would best split the input-generation algorithm into two phases: a first part that is computationally unbounded and has access to the random oracle, and a second phase that afterwards registers the users by choosing their username and respective passwords. The seeds would then be chosen at the time of user registration and thus be independent of the pre-processing phase.

4.2 Two-Factor Authentication

In this section we demonstrate that agree-and-prove cannot only capture proofs of knowledge, but also proofs of possession, such as demonstrating access to a hardware token as commonly used in a two-factor authentication.

The scheme we consider in this section combines both factors: it checks that the prover knows the correct password, and has access to the corresponding token. We thereby consider the following type of hardware token, analogous to [SR96]: upon producing the token a public/secret key pair of a PKE scheme is chosen. The secret key is then securely embedded into the token—that provides a decryption oracle—and the public key is stored for verification. In order to verify access to the token, the verifier encrypts a random challenge and checks that correct decryption is returned.

Setup. As in the previous section, the setup is parametrized in a user-administration mechanism UAdmin and the setup provides the input-generation algorithm the possibility to register users and set their passwords. The input-generation algorithm can also *assign* a certain username to the prover, thereby granting him access to the corresponding token. In addition, the input generation algorithm gets query access to all the tokens—modeling that the prover might have had temporary access to those tokens in the past. The verifier again gets the login database, as well as all public keys corresponding to the secret keys embedded in the tokens. The description can be found in Figure 11.

The relations. The agreement relation requires that the parties either have to agree on a valid username x or abort. For correctness, we require that the honest parties additionally only agree on a username if the prover possesses the corresponding token.

$$C^{\mathcal{O}_{\mathcal{F}_{\text{2-FA}}}(\cdot,\cdot,\cdot)}(1^\kappa, aux_p, aux_v, x) = 1 :\leftrightarrow x = \bot \vee (\mathsf{PW}[x] \text{ is defined} \wedge \mathsf{Assigned}[x]), \tag{7}$$

which can be efficiently implemented using oracle access to $\mathcal{F}_{\text{2-FA}}$.

The proof relation for two-factor authentication checks two conditions: it checks *knowledge* of the password and *access* to the token. Knowledge of the password is as usually phrased as the witness w which the knowledge extractor has to extract. Access, or possession, of the token on the other hand cannot be phrased as a witness extraction problem—in the end we do not want to require the extractor to extract the internal state of a secure hardware token. Rather, it is simply a property of the setup that is checked by the relation. We thus can define the relation via the condition

$$\mathcal{R}^{\mathcal{O}_{\mathcal{F}_{\text{2-FA}}}(\cdot,\cdot,\cdot)}(1^\kappa, x, w) = 1 :\leftrightarrow \mathsf{PW}[x] = w \wedge \mathsf{Assigned}[x]. \tag{8}$$

The scheme. The scheme is described in Figure 11. It makes black-box use of a secure password-based agree-and-prove scheme $(P^{pwd} = (P_1^{pwd}, P_2^{pwd}), V^{pwd} = V_1^{pwd}, V_2^{pwd})$—where all queries to the setup are handed to $\mathcal{F}_{\text{2-FA}}$ in Figure 11 whose capabilities is a superset of $\mathcal{F}_{\text{PW-A}}$—and in addition in the proof phase also checks that the prover has access to the token by requesting him to decrypt the encryption of a random challenge.

Analysis. The described agree-and-prove protocol achieves the same level of security with respect to the knowledge of the password as the underlying password based scheme (P^{pwd}, V^{pwd}). Moreover it successfully ensures possession of the token assuming the used PKE scheme is secure. This can be summarized in the following statement.

Theorem 4.3. *If the agree-and-prove scheme* (P^{pwd}, V^{pwd}) *is secure up to soundness error* $p(\kappa)$ *for the password-based authentication agree-and-prove scenario, and the PKE scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is IND-CCA1 secure, then the agree-and-prove scheme from Figure 11, for the described two-factor authentication agree-and-prove scenario, is secure up to soundness error* $p(\kappa)$ *as well.*

Prover Protocols P_1 and P_2

Parameters: password-based authentication scheme (P^{pwd}, V^{pwd}).

Prover $P_1(1^\kappa, aux_p)$:

1: Invoke $(x, st_p^{pwd}) \leftarrow P_1^{pwd}(1^\kappa, aux_p)$ and let un denote the user it agreed on, or halt if this fails.

2: Query $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{P}, \texttt{IsAssigned}, un)$. If the answer is 1, send ok to V_1 and set $st_p := (un, st_p^{pwd})$. Otherwise, send Fail to V_1 and set $st_p := \mathsf{Fail}$ (and the statement to $\perp$).

Prover $P_2(1^\kappa, st_p)$:

1: If $st_p = \mathsf{Fail}$ or the decision 0 was received from V_2 then output 0 (and halt). Otherwise, parse st_p as (un, st_p^{pwd}) and invoke $P_2^{pwd}(st_p^{pwd})$.

2: Upon receiving a challenge ch from V_2, then call $res \leftarrow \mathcal{O}_{\mathcal{F}_U^{2\text{FA}}}(\mathsf{P}, \texttt{EvalToken}, un, ch)$ and send res to V_2. Finally, output the decision obtained from V_2.

Verifier Protocols V_1 and V_2

Parameters: password-based authentication scheme (P^{pwd}, V^{pwd}) and PKE scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.

Verifier $V_1(1^\kappa, aux_v)$:

1: Invoke $(x, st_v) \leftarrow V_1^{pwd}(1^\kappa, aux_v)$ and if ok was received by P_1, define (x, st_v) as its own statement and state pair (and otherwise $x := \perp$).

Verifier $V_2(1^\kappa, st_v)$, assume $x \neq \perp$:

1: Invoke $a^{pwd} \leftarrow V_2^{pwd}(st_v)$. If $a^{pwd} = 0$, then return 0 and send the decision to P_2 and halt.

2: Otherwise, choose a random challenge $ch \in \{0,1\}^\kappa$, query $pk \leftarrow \mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{V}, \texttt{GetPublicKey}, x)$, compute $c \leftarrow \mathsf{Enc}(pk, ch)$, and send c to P_2.

3: Upon receiving res from P_2, check $res = ch$ and return 1 if so, and 0 otherwise. Send the decision to P_2.

Two-Factor Setup $\mathcal{F}_{2\text{-FA}}$

- The setup is parameterized a user-administration mechanism UAdmin and a PKE scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$.
- init: $db \leftarrow \mathsf{UAdmin.Init}(1^\kappa)$; initialize PW, Keys, and TKN to empty maps, and $\mathsf{Assigned}$ to a map pre-initialized to false.
- $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{I}, \texttt{SetPassword}, un, pw)$:
  1: If $\mathsf{Keys}[un]$ is not defined yet, create an empty function table $\mathsf{TKN}_{un}(\cdot)$, sample $(pk, sk) \leftarrow \mathsf{Gen}(1^\kappa)$ and store $\mathsf{Keys}[un] \leftarrow (pk, sk)$.
  2: Set $db \leftarrow \mathsf{UAdmin.Set}(db, un, pw)$ and $\mathsf{PW}[un] \leftarrow pw$.
- $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{I}, \texttt{GetUsers}, -)$: Return PW.
- $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{i}, \texttt{GetDB}, -)$: If $i \in \{\mathsf{I}, \mathsf{V}\}$, return db. Otherwise, return $\perp$.
- $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{i}, \texttt{TokenEval}, un, x)$: If $i = \mathsf{I}$ and $\mathsf{Keys}[un]$ is defined, or $i = \mathsf{P}$ and $\mathsf{Assigned}[un]$, then let $(pk, sk) \leftarrow \mathsf{Keys}[un]$ and return $\mathsf{Dec}(sk, x)$. In any other case, return $\perp$.
- $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{P}, \texttt{IsAssigned}, un)$: If $\mathsf{Assigned}[un]$ then return 1, otherwise 0.
- $\mathcal{O}_{\mathcal{F}_{2\text{-FA}}}(\mathsf{i}, \texttt{GetPublicKey}, un)$: If $i \in \{\mathsf{I}, \mathsf{V}\}$ and $\mathsf{Keys}[un]$ is defined, then let $(pk, sk) \leftarrow \mathsf{Keys}[un]$ and return pk. Otherwise, return $\perp$.

Figure 11: The description of the prover protocols (left) and the verifier protocols (right), and the concrete setup functionality (bottom) for the two-factor authentication.

Proof. Correctness follows from the correctness of (P^{pwd}, V^{pwd}) and the fact that P_1 checks access to the token and aborts otherwise. For soundness, the extractor can internally make use of the one for (P^{pwd}, V^{pwd}) to successfully extract the password.

It remains to show that any dishonest prover with success probability at least $p(\kappa)$ must have access to the token. This follows, however, directly from the IND-CCA1 security, and the fact that the challenge is chosen uniformly at random and cannot be guessed with non-negligible performance. Note that IND-CCA1 is needed because the input-generation algorithm can query the token for arbitrary ciphertexts beforehand. Stated differently, if the dishonest prover has no access to the token, successfully answering the challenge implies an adversary against the IND-CCA1 game and hence is a negligible probability event in κ. $\square$

References

[BFW15] David Bernhard, Marc Fischlin, and Bogdan Warinschi. Adaptive proofs of knowledge in the random oracle model. In Jonathan Katz, editor, *Public-Key Cryptography – PKC 2015*, pages 629–649, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[BG93] Mihir Bellare and Oded Goldreich. On defining proofs of knowledge. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 390–420, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[BHvOS12] J. Bonneau, C. Herley, P. C. v. Oorschot, and F. Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *2012 IEEE Symposium on Security and Privacy*, pages 553–567, May 2012.

[BJO09] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[BN19] Carsten Baum and Ariel Nof. Concretely-efficient zero-knowledge arguments for arithmetic circuits and their application to lattice-based cryptography. Cryptology ePrint Archive, Report 2019/532, 2019. https://eprint.iacr.org/2019/532.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, Jan 2000.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 280–312, Cham, 2018. Springer International Publishing.

[CGJ19] Arka Rai Choudhuri, Vipul Goyal, and Abhishek Jain. Founding Secure Computation on Blockchains. In *Eurocrypt*, pages 1–44, 2019.

[CJT02] Hung-Yu Chien, Jinn-Ke Jan, and Yuh-Min Tseng. An efficient and practical solution to remote authentication: Smart card. *Computers & Security*, 21(4):372 – 375, 2002.

[DGK17] Yevgeniy Dodis, Siyao Guo, and Jonathan Katz. Fixing cracks in the concrete: Random oracles with auxiliary input, revisited. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, pages 473–495, Cham, 2017. Springer International Publishing.

[FFS88] Uriel Feige, Amos Fiat, and Adi Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, Jun 1988.

[GMO15] Lorena González-Manzano and Agustin Orfila. An efficient confidentiality-preserving proof of ownership for deduplication. *J. Netw. Comput. Appl.*, 50(C):49–59, April 2015.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM.

[Gol06] Oded Goldreich. *Foundations of Cryptography: Volume 1*. Cambridge University Press, New York, NY, USA, 2006.

[GOS06] Jens Groth, Rafail Ostrovsky, and Amit Sahai. Perfect non-interactive zero knowledge for np. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, pages 339–358, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Hel80] M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, July 1980.

[HHPSP11] Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 491–500, New York, NY, USA, 2011. ACM.

[JK18] C. Jacomme and S. Kremer. An extensive formal analysis of multi-factor authentication protocols. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 1–15, July 2018.

[LLH06] I-En Liao, Cheng-Chi Lee, and Min-Shiang Hwang. A password authentication scheme over insecure networks. *Journal of Computer and System Sciences*, 72(4):727 – 740, 2006.

[MSL+11] Martin Mulazzani, Sebastian Schrittwieser, Manuel Leithner, Markus Huber, and Edgar Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[SR96] Victor Shoup and Avi Rubin. Session key distribution using smart cards. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 321–331, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[TW87] Martin Tompa and Heather Woll. Random self-reducibility and zero knowledge interactive proofs of possession of information. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, SFCS '87, pages 472–482, Washington, DC, USA, 1987. IEEE Computer Society.

[Unr07] Dominique Unruh. Random oracles and auxiliary input. In Alfred Menezes, editor, *Advances in Cryptology - CRYPTO 2007*, pages 205–223, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[XZ14] Jia Xu and Jianying Zhou. Leakage resilient proofs of ownership in cloud storage, revisited. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 97–115, Cham, 2014. Springer International Publishing.

[YRY04] Eun-Jun Yoon, Eun-Kyung Ryu, and Kee-Young Yoo. Efficient remote user authentication scheme based on generalized elgamal signature scheme. *IEEE Transactions on Consumer Electronics*, 50(2):568–570, May 2004.

[YWWD06] Guomin Yang, Duncan S. Wong, Huaxiong Wang, and Xiaotie Deng. Formal analysis and systematic construction of two-factor authentication scheme (short paper). In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, pages 82–91, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[YY05] Eun-Jun Yoon and Kee-Young Yoo. New authentication scheme based on a one-way hash function and diffie-hellman key exchange. In Yvo G. Desmedt, Huaxiong Wang, Yi Mu, and Yongqing Li, editors, *Cryptology and Network Security*, pages 147–160, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

A Preliminaries

In this section, we introduce the basic cryptographic primitives used throughout this work.

A.1 Merkle Trees

Merkle Tree. We use Merkle Trees in this works as succinct "commitments" to bitstrings $X = (X_1, \ldots, X_k)$, where $|X_i| = b$, with b the size parameter of the Merkle Tree leafs. Based on a hash-function $h : \{0,1\}^* \to \{0,1\}^\kappa$, a Merkle Tree is built by building pairs of elements (X_i, X_{i+1}), called the Merkle Tree leaves, and computing their hashes $v_i = h(X_i, X_{i+1})$ yielding a list $(v_1, \ldots, v_{k'})$, where v_i can be seen as a node of the two "children" (X_i, X_{i+1}) it was computed from. The process is recursively applied on each arising list until one element (i.e., a list of size 1) remains. The structure of this process is a Tree T, where the final element of the above process is the (Merkle Tree) root v_{root}, the leafs are the original data items, and the intermediate nodes are the computed hash values. We succinctly denote this process by $(T, \ell) \leftarrow \mathsf{GenMT}^h(X, b)$ where ℓ denotes the number of leaves when choosing a leaf size b.

An important property is that leaf values can be "opened" by providing a siblings path, $p \leftarrow \mathsf{MTGetPath}^h(T, i)$, which is the path from the leaf X_i to the root v_{root} that is complemented with all the siblings of each intermediate node. Knowing the root of the Merkle v_{root} and the number ℓ of leafs, a siblings path p can be efficiently verified, which we denote by $\mathsf{MTVerifyPath}^h(v_{\text{root}}, \ell, p)$, by checking the length of the path and by recomputing the hashes on p and comparing with v_{root}.

Merkle-Tree extractor. Merkle trees are often used for efficiency reasons to prove that one knows a certain set of leaf values. We make use of the following lemma proven in [HHPSP11]: Let $\mathsf{MTProve}^h(v, \ell, n)$ denote the following interactive verifier protocol: the verifier gets as input the root v and the number of leaves ℓ of a Merkle Tree $\mathsf{GenMT}^h(X, b)$, where X is the underlying bitstring X and b the size of the leaves. The verifier picks n leaf indexes $r_i \in [\ell]$ uniformly at random and asks the prover these leaf values and their corresponding siblings path and accepts if and only if all paths verify successfully. For this protocol, the following holds:

Lemma A.1 ([HHPSP11], Lemma 1)**.** *There exists a black-box extractor E with oracle access to a Merkle-Tree prover (for the above protocol), that has the following properties:*

1. *For every prover P and $v \in \{0,1\}^*$, $\ell, n \in \mathbb{N}$, and $\delta \in (0,1)$, $K^P(v, \ell, n, \delta)$ makes at most $n^2 \ell(\log(\ell) + 1)/\delta$ calls to oracle P.*
2. *Let $(T, \ell) \leftarrow \mathsf{GenMT}^h(X, b)$ for parameters as above and let v denote the root of T, and fix a prover $P^* := P'(h, X, n)$. Then, if P^* has probability at least $(1-\alpha)^n + \delta$ of convincing the verifier $\mathsf{MTProve}^h(v, \ell, n)$, where $\alpha, \delta \in (0,1)$ then with probability at least $1/4$ (over its internal randomness), the extractor $K^{P^*}(v, \ell, n, \delta)$ outputs values for at least an $(1-\alpha)$-fraction of the leaf values of T (together with valid siblings paths for all those leaves).*

The extractor K works as follows:

1. ***for** $i = 1$ **to** n, **for** $j = 1$ **to** ℓ:*
   - *1.1 **repeat for** $\lceil n(\log(s) + 1)/\delta \rceil$ times:*
     - *Choose at random $r_1, \ldots, r_n \in [\ell]$ and invoke $P(r_1, \ldots, r_{i-1}, j, r_{i+1}, \ldots, r_n)$*
2. ***Output** all the sibling paths for all the leaves for which P returned valid siblings path with respect to control information v and ℓ.*

For a proof of this lemma, we refer the reader to [HHPSP11]. Furthermore, we note that all statements hold using black-box access to the hash-function h (which could therefore be replaced by a random oracle). Finally, the following equivalent description of extractor K is slightly more preferable to our setting as it does not rely on the knowledge of δ: Let $\mathtt{T}_U$ be a runtime bound.

1. Execute the following subprocess $K'(v, \ell, n)$ for $\mathtt{T}_U$ times:

   1.1 **for** $i = 1$ **to** n, **for** $j = 1$ **to** ℓ:

   - Choose at random $r_1, \ldots, r_n \in [\ell]$ and invoke $P(r_1, \ldots, r_{i-1}, j, r_{i+1}, \ldots, r_n)$

2. **Output** all the sibling paths for all the leaves for which P returned valid siblings path (w.r.t. control information v and ℓ).

The distribution of queries on which P is evaluated remains the same when switching the order of iterations and hence does not change the output distribution of this process. The probabilistic guarantees provided by Lemma A.1 therefore hold whenever $\mathtt{T}_U \geq \lceil n(\log(s) + 1)/\delta \rceil$. Note that the parameter b is not of primary interest in our asymptotic treatment, but it can be used to trade efficiency of the extractor versus communication complexity in the proof (i.e., increase b to decrease ℓ). In this work, ℓ will be effectively bounded by the fact that an efficient prover will otherwise not conclude the agreement phase (and hence no proof phase will take place).

A.2 Erasure Codes

An (n, k, d) erasure code over an alphabet Σ with error symbol $\perp \notin \Sigma$, is a pair of (efficient) algorithms (E, D) that satisfy the following requirement for all $F \in \Sigma^k$: Let $\bar{F} := E(F) \in \Sigma^n$ and define the set

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

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

A.3 Signature and Encryption Schemes

We introduce the basic notation for the standard cryptographic primitives for completeness. Recall that throughout the text, κ denotes the security parameter.

Signature scheme. We make use of an existentially unforgeable signature scheme under chosen message attacks (EU-CMA). For a signature scheme Sig we denote the key generation algorithm by $(vk^{\mathsf{Sig}}, sk^{\mathsf{Sig}}) \leftarrow \mathsf{Sig.Gen}(1^\kappa)$, the signing algorithm by $\sigma \leftarrow \mathsf{Sig.Sgn}(sk^{\mathsf{Sig}}, m)$, and the verification algorithm by $\mathsf{Sig.Vrf}(vk^{\mathsf{Sig}}, \sigma, m)$.

Symmetric encryption schemes. We further make use of symmetric encryption schemes with indistinguishable ciphertexts under chosen ciphertext attacks (IND-CPA). For a symmetric encryption scheme SE we denote the key generation algorithm by $k^{\mathsf{SE}} \leftarrow \mathsf{SE.Gen}(1^\kappa)$, the encryption algorithm by $c \leftarrow \mathsf{SE.Enc}(k^{\mathsf{SE}}, m)$ and the decryption algorithm by $m' \leftarrow \mathsf{SE.Dec}(k^{\mathsf{SE}}, c)$.

Public-key encryption schemes and ElGamal. A generic PKE scheme is denoted by the triple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ for key generation, encryption, and decryption, respectively. We make further specific use of the ElGamal public key encryption scheme. More specifically, ElGamal is applied on a message space that is identified with a cyclic group $G = \langle g \rangle$ of prime order q, $2^{\kappa-1} < q < 2^\kappa$, where g is a known generator. We further assume throughout this work that the decisional Diffie-Hellman assumption (DDH) holds in G (and thus ElGamal IND-CPA secure). We refer to the scheme by $\mathsf{ElGamal}$ and to its key generation algorithm by $(ek^{\mathsf{ElGamal}} := g^x, dk^{\mathsf{ElGamal}} := x) \leftarrow \mathsf{ElGamal.Gen}(1^\kappa)$, where $x \in \mathbb{Z}_q$ is chosen uniformly at random, to its encryption algorithm by $(c_1 := g^r, c_2 := g^{rx} \cdot m) \leftarrow \mathsf{ElGamal.Enc}(ek^{\mathsf{ElGamal}}, m)$, where $r \in \mathbb{Z}_q$ is chosen uniformly at random, and finally to its decryption algorithm by $m' := c_2 \cdot c_1^{-x} \leftarrow \mathsf{ElGamal.Dec}(dk^{\mathsf{ElGamal}}, (c_1, c_2))$.

A.4 Non-interactive Zero-Knowledge Proof Systems

We define non-interactive zero-knowledge proofs following Groth [GOS06].

Definition A.2. Let R be an efficiently computable binary relation and consider the *language* $L := \{x \mid \exists w\ (x, w) \in R\}$. A *non-interactive proof system* for L (or for R) consists of the following three PPT algorithms:

Key generation: The algorithm Gen on input a security parameter 1^κ, outputs a *common reference string* crs.

Proving: The algorithm Prove on input a common reference string crs, a *statement* x, and a *witness* w, outputs a *proof* π.

Verification: The algorithm Ver on input a common reference string crs, a statement x, and a proof π, outputs a bit b (where $b = 1$ means "accept" and $b = 0$ means "reject").

We require *perfect completeness*, i.e., for all crs in the range of Gen and for all $(x, w) \in R$, we have

$$\mathsf{Ver}\big(crs, x, \mathsf{Prove}(crs, x, w)\big) = 1$$

with probability 1.

Definition A.3 (Soundness)**.** Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver})$ be a non-interactive proof system for a language L and let $\mathcal{A}$ be a probabilistic algorithm. We define the *soundness advantage* of $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{NIZK\text{-}snd}}_{\mathcal{E},\mathcal{A}} := \Pr^{crs \leftarrow \mathsf{Gen}(1^\kappa);\ (x,\pi) \leftarrow \mathcal{A}(crs)}\big[x \notin L\ \wedge\ \mathsf{Ver}(crs, x, \pi) = 1\big].$$

The scheme $\mathcal{E}$ is *computationally sound* if $\mathsf{Adv}^{\mathsf{NIZK\text{-}snd}}_{\mathcal{E},\mathcal{A}}$ is negligible for all efficient $\mathcal{A}$ and *perfectly sound* if $\mathsf{Adv}^{\mathsf{NIZK\text{-}snd}}_{\mathcal{E},\mathcal{A}} = 0$ for all $\mathcal{A}$.

Definition A.4 (Computational zero-knowledge)**.** Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver})$ be a non-interactive proof system for a relation R and let $S = (S_1, S_2)$ be a pair of PPT algorithms, called *simulator*. Further let $S'(crs, \tau, x, w) = S_2(crs, \tau, x)$ for $(x, w) \in R$, and $S'(crs, \tau, x, w) = \mathtt{failure}$ for $(x, w) \notin R$. We define the *zero-knowledge advantage* of a probabilistic algorithm $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{NIZK\text{-}ZK}}_{\mathcal{E},S,\mathcal{A}} := \Pr^{crs \leftarrow \mathsf{Gen}(1^\kappa)}\big[\mathcal{A}^{\mathsf{Prove}(crs,\cdot,\cdot)}(crs) = 1\big] - \Pr^{(crs,\tau) \leftarrow S_1(1^\kappa)}\big[\mathcal{A}^{S'(crs,\tau,\cdot,\cdot)}(crs) = 1\big].$$

We call $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver}, S_1, S_2)$ a *non-interactive zero-knowledge (NIZK) proof system* for R if $\mathsf{Adv}^{\mathsf{NIZK\text{-}ZK}}_{\mathcal{E},S,\mathcal{A}}$ is negligible for all efficient $\mathcal{A}$.

Definition A.5 (Knowledge extraction)**.** Let $\mathcal{E} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver})$ be a non-interactive proof system for a relation R and let $E = (E_1, E_2)$ be a pair of PPT algorithms, called *knowledge extractor*. We define the *knowledge extraction advantages* of a probabilistic algorithm $\mathcal{A}$ as

$$\mathsf{Adv}^{\mathsf{NIZK\text{-}ext}_1}_{\mathcal{E},E,\mathcal{A}} := \Pr^{crs \leftarrow \mathsf{Gen}(1^\kappa)}\big[\mathcal{A}(crs) = 1\big] - \Pr^{(crs,\xi) \leftarrow E_1(1^\kappa)}\big[\mathcal{A}(crs) = 1\big],$$
$$\mathsf{Adv}^{\mathsf{NIZK\text{-}ext}_2}_{\mathcal{E},E,\mathcal{A}} := \Pr^{(crs,\xi) \leftarrow E_1(1^\kappa);\ (x,\pi) \leftarrow \mathcal{A}(crs);\ w \leftarrow E_2(crs,\xi,x,\pi)}\big[\mathsf{Ver}(crs, x, \pi) = 1\ \wedge\ (x, w) \notin R\big].$$

We call $(\mathsf{Gen}, \mathsf{Prove}, \mathsf{Ver}, E_1, E_2)$ a *non-interactive proof of knowledge system* for R if $\mathsf{Adv}^{\mathsf{NIZK\text{-}ext}_1}_{\mathcal{E},E,\mathcal{A}}$ and $\mathsf{Adv}^{\mathsf{NIZK\text{-}ext}_2}_{\mathcal{E},E,\mathcal{A}}$ are negligible for all efficient $\mathcal{A}$.