Rotatable Zero Knowledge Sets:
Post Compromise Secure Auditable Dictionaries with application to Key Transparency⋆

Brian Chen[1] , Yevgeniy Dodis[2], Esha Ghosh[3], Eli Goldin[2], Balachandar Kesavan[1], Antonio Marcedone[1] , and Merry Ember Mou[1]

[1] Zoom Video Communications `{brian.chen,surya.heronhaye,`
`antonio.marcedone,merry.mou}@zoom.us`
[2] New York University `{dodis@cs.nyu.edu,eg3293@nyu.edu}`
[3] Microsoft Research `esha.ghosh@microsoft.com`

Abstract. *Key Transparency* (KT) systems allow end-to-end encrypted service providers (messaging, calls, etc.) to maintain an auditable directory of their users' public keys, producing proofs that all participants have a consistent view of those keys, and allowing each user to check updates to their own keys. KT has lately received a lot of attention, in particular its privacy preserving variants, which also ensure that users and auditors do not learn anything beyond what is necessary to use the service and keep the service provider accountable.

Abstractly, the problem of building such systems reduces to constructing so-called append-only Zero-Knowledge Sets (aZKS). Unfortunately, existing aZKS (and KT) solutions do not allow to adequately restore the privacy guarantees after a server compromise, a form of Post-Compromise Security (PCS), while maintaining the auditability properties. In this work we address this concern through the formalization of an extension of aZKS called *Rotatable ZKS* (RZKS). In addition to providing PCS, our notion of RZKS has several other attractive features, such as a stronger (extractable) soundness notion, and the ability for a communication party with out-of-date data to efficiently "catch up" to the current epoch while ensuring that the server did not erase any of the past data.

Of independent interest, we also introduce a new primitive called a *Rotatable Verifiable Random Function* (VRF), and show how to build RZKS in a modular fashion from a rotatable VRF, ordered accumulator, and append-only vector commitment schemes.

Keywords: Key Transparency, Zero-Knowledge Sets, Verifiable Random Functions, Post-Compromise Security.

1 Introduction

End-to-end encrypted communication systems (E2EE), including encrypted chat services (such as WhatsApp [49], Signal [42], Keybase [23], iMessage [2]) and encrypted calls (Zoom [7], Webex [48], Teams [35]), are becoming increasingly common in today's world. E2EE systems require each user to publish a public key, and use the corresponding secret key along with their communication partners' public keys to compute a shared secret which can be used to secure the communication. To enable this, service providers (such as Apple, Zoom, Meta, Microsoft, etc.) need to maintain a directory that maps each user to their public keys, a Public Key Infrastructure (PKI) analogous to the one in place to secure the web. The end-to-end guarantees depend on the authenticity of these public keys, as otherwise a malicious service provider (or one who is hacked or compelled to act maliciously) can replace an honest user's identity public key with another public key whose secret key is known to the provider, and thus implement a meddler-in-the-middle (MitM) attack without the communicating users ever noticing.

KEY TRANSPARENCY. To mitigate this issue, many E2EE communication systems provide users with "security codes", i.e. digests of the communication partners' identity public keys rendered as lists of digits or words, or QR codes. To detect potential meddler-in-the-middle attacks, the communicating users are expected to manually check these codes, either by reading them aloud (in calls), scanning them with their phone apps, or otherwise sharing them out-of-band. It is well understood that this has

---

⋆ This is the full version of the article with the same title published in the Proceedings of ASIACRYPT 2022, Springer, © IACR 2022.

severe usability challenges [3, 20, 21, 47]. *Key Transparency* (KT)[4] systems augment these checks with a fully automated solution that improves both usability and security.

KT systems enable service providers to maintain an auditable directory that maps each user's identifier (such as a username, phone number or email address) to their identity public keys (analogously to how Certificate Transparency [28] allows to monitor PKI certificates). Providers compute and advertise a short (signed) "commitment" com to the whole directory, and update it (creating a new *epoch*) whenever users join the directory or update their keys. When users query a particular *label* label (a key in the map, such as a username), they get the corresponding *value* val (i.e. public key) and a *proof* π that this (label, val) pair is consistent with com.[5] Clients are then encouraged to periodically monitor the directory to make sure their own identifier maps to the correct keys, thus detecting any attempt to MitM their communications.

Assuming cryptographic soundness of such proofs, to ensure that all clients receive the same answer when they query for the same label, it is enough to ensure they all have the same commitment com. To achieve this, KT relies on clients gossiping the commitment [32], or on public and untamperable ledgers such as blockchains [24]. While the implementation of such *gossiping schemes* is not part of the design (and definition) of KT, and they have seen little practical deployment[6] [16, 31], improvements in this respect seem feasible, and even the potential for users to independently check might deter the server from misbehaving.

AUDITING. Although the basic functionality already goes a long way towards holding the server accountable for providing incorrect keys to users, clients would incur a high burden if they had to check the server's consistency at every epoch, especially clients whose keys do not change often as the directory evolves. To mitigate that, most KT systems provide additional *auditing functionality*, where more resourceful parties (called *auditors*) can continuously check that certain properties of the directory are maintained across updates (such as the fact that old keys are never erased, and newer ones are simply appended). Technically, when updating the old commitment com to directory D with a newer commitment com' to D', the server can issue a certain proof π_S asserting that $D \subseteq D'$ (and, ideally, revealing nothing more beyond $|D' \setminus D|$). While any user can be an auditor, in practice it is envisioned that relatively few external auditors would continuously monitor the server in this way, and most clients would rely on that assurance. This also justifies relatively large update proofs (with size proportional to $|D' \setminus D|$). Such KT systems are called *auditable*. In addition to keeping the server honest, auditable KTs might ease the need of clients to check their keys at every epoch, if trusted auditors exist. For example, if a client checked earlier that their keys were correct w.r.t. some (audited) old value com', and later got the current value of com from a trusted auditor, they can be sure their keys are still correct w.r.t. com, thus eliminating the need to ask the server to prove this fact again.

To the best of our knowledge, Keybase [27] is the first deployment of an auditable public key directory; they published the first KT digest on April 2014 [26]. Keybase was created as a more user-friendly and secure replacement for PGP, so their KT favors full transparency and auditability over privacy guarantees. For example, Keybase publicly advertises [25] how many devices each user has the Keybase client app installed on, and how often their keys change (i.e., the app is reinstalled). While this is an acceptable tradeoff for many, this privacy leakage can also be a concern, as surfaced in [29], which studied the privacy concerns of using Keybase for US journalists and lawyers. There could be other important business reasons for requiring privacy as well. A business might not want to use a KT system if doing so means revealing to the world how much churn the company has. If the KT system is used to authenticate group membership as well, revealing which groups a user is part of could leak the organizational structure of the business and facilitate social engineering attacks. In fact, Google and Zoom advocate for adding privacy to KT systems [7, 19]. In addition to being privacy-conscious (which is a good practice anyway), these industry leaders are also concerned about current and future laws and regulations, such as GDPR. Indeed, once a major system is in play, it is extremely hard to change it when a new privacy law/regulation comes into effect. For example, creating a publicly visible and immutable trail of a user's encryption key changes in a Key Transparency directory would likely cause

---

[4] KT is known under various names in the literature, such as *auditable registries, verifiable key directories, auditable directories* etc. For the purpose of this manuscript, we will stick to using KT.

[5] Additionally, if no (label, val) pair exists for a given label, the proof π becomes an *absence* proof for this label.

[6] While Keybase posts its KT digests to a blockchain, official Keybase clients do not check them.

a GDPR violation. Similarly, if a user asks the provider to delete their account and all traces, doing so would be very hard without privacy built-in.

PRIVACY-PRESERVING KT. Motivated by these and other considerations, new KT schemes were developed with privacy. Broadly speaking, privacy can be divided in two categories: *content-privacy* and *metadata-privacy*. Content-privacy hides public keys and usernames from unauthorized parties (e.g., auditors and other users who wouldn't otherwise be able to query for those usernames). KT systems supporting content privacy include [22, 31, 44–46]. Metadata-privacy also hides information such as when each user first registered in the KT, when and how often their keys change, correlations between multiple updates, etc. on top of content-privacy. We denote metadata-hiding KT schemes as *privacy-preserving KT* (ppKT) [8, 19, 32]. In ppKT, both external auditors and users should learn as little as possible beyond the data they are actively querying. For example, KT commitments and proofs for a certain user identifier should not reveal information about other users' keys and how often they are changing. Similarly, auditors should enforce that no data is ever deleted from the directory, while learning as little as the total number of keys being updated.

Unlike KT systems without any privacy, in which the key directory data structure can be built entirely on symmetric key primitives like Merkle Trees [11], practical KT systems (with either content-privacy or metadata-privacy) achieve privacy through asymmetric primitives such as Verifiable Random Functions (VRFs) [34].[7] Ignoring some important details, given a (label, val) pair, the server holding the VRF's secret key will use a pseudorandom label $y = \mathsf{VRF}(\mathsf{label})$ in place of the original label. Then: (a) pseudorandomness of y ensures that no information about the original label is leaked; (b) verifiability of y ensures that it can be convincingly opened to the original label; and (c) uniqueness of $y = \mathsf{VRF}(\mathsf{label})$ ensures that each label can be used only once.[8]

KEY ROTATION AND POST COMPROMISE SECURITY. With the growing popularity and user-base of E2EE communication systems, ppKT is very close to real-world, large-scale deployments [7, 15, 19]. However, as with any real world system, a ppKT system will likely get compromised at some point, so there should be a robust plan to recover from such a compromise, should it happen. One subtle observation in this regard is that current ppKT systems all require the server to maintain a secret key sk (e.g., the secret key to the VRF, as explained above), in addition to simply storing the users' data. Thus, recovering from such compromise necessitates updating the secret/public keys of the server, which is called key rotation. In addition, even if no evidence of actual compromise is ever found, periodically rotating secret keys is considered an industry best practice and sometimes mandated by regulations [38, 39]. For ppKT, rotating the key would ideally ensure that compromise of the server would only violate the privacy of past records (which is unavoidable, as the server stores this data anyway), but not of future records.[9] In other words, the primary goal of key rotation is to achieve what is known as *post compromise security* (PCS): the privacy of ppKT systems should be seamlessly restored in case of (possibly silent) key compromise. This is the main question we address in this work:

How easy is it to add PCS to a ppKT, while maintaining high efficiency?

A NAIVE SOLUTION. To see why this question is non-trivial, let us look at a naive attempt to add key rotation to any existing ppKT, such as SEEMless [8]. The first idea is to simply pick a fresh key pair (sk_t, pk_t) for a ppKT with every rotation number t, and basically view the final database D as a disjoint union of t smaller databases $D_1, \ldots, D_t$, where D_i corresponds to the key pair (sk_i, pk_i). On the surface, this seems to maintain the efficiency of the base ppKT, since the server can figure out which "mini-database" D_i contains a given record (id, pk_{id}) and provide a proof only for this value of i. Unfortunately, this does not work, as the server also needs to provide $(i - 1)$ absence proofs that id does not belong to any of the previous databases $D_1, \ldots, D_{i-1}$. Otherwise, the server could insert (id, pk_{id}) in database i, (id, pk'_{id}) in database i', and provide different clients with different answers to the same

---

[7] Informally, a VRF [34] is similar to a standard pseudorandom function (PRF), except the secret key owner is also committed to the entire function in advance, and can selectively open some of its outputs in a verifiable manner.

[8] Property (c) is why VRF is needed, and regular commitments to label do not work.

[9] The effect of compromise on authenticity/auditability is rather minimal anyway, as the key used to sign the commitments would typically be authenticated using the web PKI, and thus can be revoked upon compromise using existing techniques. Moreover, learning the secret server state doesn't help break the binding of the commitment to the entire set of current records in the directory.

query, even if a good base ppKT is used. And in the case of id not belonging to the entire database D, the server must provide t such absence proofs. Given that clients might need to lookup many identifiers at once and that providers will have to handle a large volume of queries simultaneously, this multiplicative slowdown is unacceptable for practical use.

A better approach — and indeed the approach we take in this work — is to transfer the entire database D when switching from sk_{t-1} to sk_t, thereby initializing $D_t = D_{t-1}$, and then growing D_t when new data items are appended. This ensures that the efficiency of key lookup, the most frequent and important operation in ppKT, is indeed inherited from the base ppKT to which we are adding PCS, because it is always done w.r.t. the latest public key pk_t. Of course, now the server also needs to prove that it honestly initialized $D_t = D_{t-1}$, so that the users or auditors performing this (potentially expensive, but *rare*) check are convinced that no data was added, removed, or modified. Moreover, this check should be done in a privacy-preserving way, so that auditors learn as little as possible about the database $D = D_{t-1} = D_t$ at this moment, beyond the fact that it was correctly "copied" during key rotation.

Unfortunately, none of the existing ppKT systems appear friendly to such *(key) rotation proofs*, while generic zero-knowledge proofs would be prohibitively inefficient given large database sizes in typical ppKT systems. As our main technical contribution, we overcome this difficulty by designing a specialized, but still highly efficient, ppKT system which supports efficient key rotation, and hence provides PCS against (possibly silent) server compromises.

1.1 Our Contributions

Before our concrete solution, we list our contributions from the modeling and definitions perspective.

MODELING AND DEFINITIONS. First, much like earlier works on ppKT [8, 19, 32] we abstract the primitive that we need. Our primitive, which we term *Rotatable Zero Knowledge Set* (RZKS), is a natural extension of the so-called append-only zero-knowledge set (aZKS) from [8].

At a high level, aZKS is a primitive where a prover can incrementally commit to a dictionary D, and later prove (in zero-knowledge) a statement of the form that a certain (label, val) pair belongs to the dictionary, or that a certain label does not belong to the dictionary (for any val). Moreover, there is at most one val for any label, and this val cannot be modified once it is assigned. To model the incremental nature of aZKS, the prover can also prove the "append-only" property to the auditors, such that two commitments com and com′ correspond to two dictionaries D and D', where D is a subset of D', in almost[10] zero-knowledge.

Our RZKS notion extends aZKS in several ways. First, and most importantly from the perspective of PCS, we allow a new algorithm for key rotation. Syntax-wise, it is the same as the append algorithm of aZKS: given a (possibly empty) set S of fresh $\{(\text{label, val})\}$-pairs to be appended to the current database, we update the commitment com to D to a new commitment com′ to $D' = D \cup S$, and output a proof π_R that this operation was done "consistently". However, unlike the regular append operation given by proof π_S, the proof size and time for the rotation operation is allowed to be proportional to the entire database D', as opposed to the number of appended elements $|S|$. What we gain though is the PCS property: unlike with the regular append, compromising the server's state (including D) before the rotation does not help the attacker learn any new information about newly appended elements S, or any elements appended in the future (including those by the standard append operation). (As a bonus, it also wipes out the minimal leakage of regular append mentioned in Footnote 10.)

Second, and of independent interest, we extend the aZKS functionality to support what we call *extension proofs*. Such proofs allow a party to verify that a given newer commitment $\text{com}_{t'}$ commits to a given older commitment com_t (and, therefore, also implies that both $\text{com}_{t'}$ and com_t commit to the same sequence $\text{com}_1, \text{com}_2, \ldots, \text{com}_{t-1}$), for any $t' > t$ (as opposed to the append-only proofs in aZKS only supporting $t' = t + 1$). Here, t and t' are the total number of appends and rotations that were performed to produce the dictionary corresponding to each commitment. These extension proofs are *extremely* efficient (only logarithmic in the number of epochs t'), as they can be instantiated using Merkle Tree append-only proofs [8, 45].

---

[10] According to a well-defined leakage profile. For [8], the only such leakage reveals if a label known to be missing in D is later inserted in D', which seems acceptable for the main application to KT.

Note that, by themselves, extension proofs do not prove that the database has evolved consistently (for example, it is possible that $D_t \not\subseteq D_{t'}$). However, auditors still check that each successive epoch correctly performs append or rotation operations. As a result, extension proofs allow users to confirm that the commitments they receive are authentic and represent a consistently evolving database by occasionally verifying commitments with a trusted auditor, instead of verifying every commitment they receive, which would be a frequent and expensive operation. Concretely, suppose a user receives a series of commitments $\mathsf{com}_{t_1}, \ldots, \mathsf{com}_{t_n}$ from the server, possibly over a long period of time, along with extension proofs from each com_{t_i} to the next $\mathsf{com}_{t_{i+1}}$ (which the user verifies). Then, by verifying just com_{t_n} with an auditor, blockchain, or other gossiping mechanism, the user can guarantee the consistency of every previous com_{t_i} they received with those other sources' views. Furthermore, if the user trusts that the source they are verifying against has also verified that the database evolved consistently at each epoch, they can infer that each $D_{t_i} \subseteq D_{t_{i+1}}$. This also allows auditors and other clients to only gossip about the latest commitment com' and forget any previous commitments. If an older commitment $\mathsf{com}_{\mathsf{old}}$ is ever needed, the server can always provide $\mathsf{com}_{\mathsf{old}}$ and the extension proof from $\mathsf{com}_{\mathsf{old}}$ to com'.

Third, each query also explicitly indicates the epoch at which the queried pair was added to the RZKS directory, which can be verified without any increase in the proof size (obtaining this information in an aZKS would require multiple proofs). We believe that this information can be helpful in practical applications, as older records/keys are often considered more trustworthy than newer ones (the owner has had more time to react to a compromise), and quickly comparing the age of two records can be helpful for more complex applications of RZKS beyond standard KT[11]. Moreover, while previous ppKT do not allow to determine this efficiently, they do not hide this information either.

Finally, our notion of RZKS strengthens the soundness definition of aZKS presented in [8]. Namely, the latter mandates that an adversary cannot produce valid proofs of conflicting statements (for example, proving that the same key maps to different values, possibly in different epochs). Instead, we notice that the soundness of the SEEMless construction of [8] is proven in the Random Oracle model anyway, where we can achieve much stronger forms of soundness. Indeed, our RZKS notion demands a very strong form of *extractability-based* soundness. Roughly, we require the existence of an extractor, which, given any malicious commitment com produced by the attacker (and its random oracle queries), can extract the entire database D for which the attacker can later produce verifying membership proofs. We believe this stronger property makes it easier to reason about the security of applications of RZKS.

RZKS CONSTRUCTION. Finally, we show how to build an efficient RZKS system. Our starting point is the aZKS construction from SEEMless [8]. SEEMless uses — in a black-box way — a *verifiable random function* (VRF) [34] and cryptographic hash function to build their aZKS, and recommends the specific DDH-based VRF from the upcoming VRF standard [18].

Recall, a VRF allows the secret key owner (e.g., server) to compactly commit to an entire random-looking function f, but in a way that allows them to convincingly open individual function outputs $f(x)$, without compromising the randomness of yet unopened outputs $f(x')$ for $x' \neq x$. In the aZKS construction of [8], when appending a $(x = \mathsf{label}, v = \mathsf{val})$ pair to D, the server uses the VRF output $f(x)$ to decide where to put a commitment to v in some Merkle Tree T that it builds. If this place is occupied, the server knows that D already contains some v' associated with label x, and can reject the request. Otherwise, it inserts some commitment to v into the Merkle Tree T, and uses the new root of T as the modified commitment value com' to $D' = D \cup \{(x, v)\}$. Intuitively, the use of VRF ensures privacy, as it hides information about the labels that would otherwise be leaked by Merkle proofs of "neighboring" labels. On the other hand, VRF uniqueness and verifiability properties ensure that the server cannot cheat.

One can now consider how to extend the scheme above to support key rotation, provided that the underlying VRF can support what we call *VRF rotation proofs*. Intuitively, a RZKS rotation proof will switch the VRF key from f_1 to f_2, rebuild the Merkle Tree T_1 into T_2 using the same commitments to each of the values, and openly reveal the one-to-one correspondence between leaves of T_1 and T_2 associated with all keys x present in the original database D before the rotation. However, recall that the value x itself should be hidden from auditors verifying consistency of key rotation, which leads to

---

[11] For example, Keybase uses its KT dictionary to also store other statements signed by a user's device, such as when a user wants to add another user to a group: knowing that the statement was signed before the key that signed it is revoked/rotated is important for the security of the system.

the following problem we solve in this work. We need to design a VRF with a fast zero-knowledge proof showing that two VRF outputs y_1 and y_2 under two independent keys f_1 and f_2 correspond to the same secret input x: $y_1 = f_1(x)$ and $y_2 = f_2(x)$. We call this novel type of VRFs *rotatable*. We discuss them next, and defer the rest of the details of our final RZKS construction to Section 5.2, simply highlighting here its modularity: it is built from any rotatable VRF, commitments and other generic building blocks instantiable from Merkle Trees.[12]

ROTATABLE VRFs. Unfortunately, supporting an efficient ZK proof mentioned above is not sufficient for the type of rotatable VRFs we need for RZKS. To achieve PCS for RZKS, our VRF also needs to satisfy a novel type of "non-committing property": upon compromise, the attacker learns of a compact secret key sk for the VRF, which suddenly explains a lot of VRF outputs $\{y\}$ that the attacker saw prior to the corruption (but did not know the corresponding inputs $\{x\}$). More concretely, we use a simulation-based rotatable VRF definition, extending the earlier "simulatable VRF" notion of [10] to handle rotations. Under this notion, the simulator must in particular "win" in the following game (which is the most challenging part of our definition explaining the heart of the problem). The simulator must commit to a VRF public key pk, get a bunch of *random* strings $\{y\}$ as various VRF outputs of *unknown* inputs $\{x\}$, answer random oracle queries from the attacker, then learn the hidden set $\{x\}$, and finally produce a secret key sk that correctly explains that $f(x) = y$ for all matching (x, y) pairs of the corresponding sets $\{x\}$ and $\{y\}$.

This problem seems to relate to the area of non-committing encryption (NCE) [6, 37], where one compact secret is supposed to open many previously committed ciphertexts in a certain way. As with non-committing encryption [37], building standard model "non-committing" VRF is impossible, as one short secret key sk cannot "explain away" arbitrarily many random looking outputs y. On the other hand, given that several NCE schemes exist in the random oracle model (e.g., [5]), one might hope that the same simple ideas[13] will work in our VRF setting as well. Unfortunately, this does not appear to be the case, due to the inherently algebraic structure of VRF proofs. To understand this inherent tension, let us consider the concrete efficient VRF (from the VRF standard [18]) recommended by the authors of SEEMless. In this VRF, the secret key sk for the VRF is a random exponent α, the public key $pk = g^\alpha$ (for public generator g), and the VRF value $y = f(x) = F'(F(pk, x)^\alpha)$, where F is a random oracle and F' is an "extractor" meant to map a random group element to a random bit-string. (The proof π that $y = f(x)$ is the value $z = F(pk, x)^\alpha$ and the standard Fiat-Shamir variant of the Σ-protocol for the DDH tuple $(g, pk, F(pk, x), z)$ [13].)

When rotating the key pair (α, g^α) to a fresh key pair (β, g^β), first we need to ensure that there exists an efficient ZK proof showing that two random values y and y' satisfy the relation $y = F'(F(g^\alpha, x)^\alpha)$ and $y' = F'(F(g^\beta, x)^\beta)$. As the first obstacle, this seems hard due the outside extractor F'. Fortunately, this problem is trivially solved by getting rid of the "outer extractor" F', and thinking of the VRF as outputting a group element (rather than bit-string) $y = F(pk, x)^\alpha$. Indeed, the standard VRF proof in [18] shows that the above VRF is already secure. The next problem comes from the fact that the old VRF f and the new VRF f' have different public keys g^α and g^β hashed inside the "inner random oracle" F. Once again, it turns out that the VRF proof just needs some domain separation, and goes through if we redefine the output $y = F(salt, x)^\alpha$, where $salt$ is some unpredictable value which does not need to change with any rotation.[14]

This already gives us the ability to construct (at least "syntactically") the required ZK proof of rotation when moving from $sk = \alpha$ to $sk' = \beta$. Indeed, for any unknown x, if $y = F(x)^\alpha$ and $y' = F(x)^\beta$, the server can simply prove that the tuple $(g^\alpha, g^\beta, y, y')$ is a proper DDH tuple (using witness $w = \beta/\alpha$).[15] As we said, though, we also need to provide the PCS property mentioned above. And this appears hopeless at first glance. Indeed, the public key $pk = g^\alpha$ commits to α information-theoretically. Moreover, when programming the random oracle query $F(x)$, the simulator does not know yet which random output y corresponds to x. Hence, the simulator has no chance to correctly program $F(x) = y^{1/\alpha}$. For regular ("non-rotatable") VRFs, we would try to fake the Fiat-Shamir proofs for correctness. In fact,

---

[12] Namely, so called ordered accumulators, and append-only vector commitment schemes. See Section 5.1.

[13] Namely, to a posteriori program random oracle in a manner depending on the strings y, on appropriate inputs involving the secret key sk.

[14] For simplicity of exposition, we omit $salt$ from our description, but recommend that each application uses a fresh salt.

[15] Our final ZK proof will aggregate many such individual input rotation proofs into one compact proof.

this would extend to rotatable VRFs without the PCS property (i.e., without corruptions of α); but, of course, this is not very interesting from the application perspective. In the case of corruptions, however, the simulator is committed to the secret key α, and will be caught cheating with certainty.

OUR SOLUTION: GGM ANALYSIS. Interestingly, the difficulty of completing the simulation-based PCS proof for our tweaked construction $y = F(x)^\alpha$ does not seem to translate to an explicit attack on the resulting rotatable VRF. Rather, we cannot build a sufficiently adaptive simulator to prevent the type of attack in the previous paragraph. So we ask the question if the construction might be actually be secure, despite the natural proof breaking down. Somewhat surprisingly, we give supporting evidence that this is the indeed the case, by providing such a security analysis in the *generic group model* (GGM) of Shoup [41].[16]

Recall that in Shoup's GGM, all group elements have random bit-string representations, and the group operation $\star$ also has a random multiplication table $\star(a, b)$ (subject to associativity of multiplication). As such, most security assumptions in standard groups (e.g., DDH) will hold in the GGM unconditionally. But now the simulator can commit to the public key $pk = g^\alpha$ *without committing to α information-theoretically*. Intuitively, since the attacker does not know value α before the compromise, and has a bounded number of multiplication queries to explore, the simulator can simply choose a random value of α as the secret key, and will have enough freedom to "mess" with the multiplication table $\star(a, b)$ to simultaneously satisfy many equations of the form $y_i = F(x_i)^\alpha$ (as well as $pk = g^\alpha$). However, the formal proof of this statement is rather subtle, and forms one of the main technical novelties of this work. For example, the group laws mandate certain relationships that the attacker can always satisfy, so the simulator has to be extremely careful not to "overplay its hand" and program the multiplication table too aggressively. We present the full simulation proof in Section 4.4, and hope that our GGM proof technique will find applications for analyzing other "non-committing" algebraic primitives.

INTERPRETATION OF OUR RESULT. On a philosophical point, we suggest that the value of our GGM security proof should be understood in light of the fact that ROM-based proofs seem to be inherently stuck, at least for the natural rotatable VRF that we consider. Aside from the obvious consideration that we focused on finding a practical solution to a natural problem for which we could not find an explicit attack, we note that the requirements in our definition of rotatable VRFs are quite strong. Basically, the simulator has to answer all ideal queries without knowing any of the input/output behavior of the VRF, and then must produce a single secret key consistent with not only these ideal queries, but also all fake proofs (including rotation). From this perspective, we feel that it is quite surprising that we managed to overcome these difficulties at all, even relying on the GGM. The GGM proof can also be considered a sanity check that our scheme is likely to be secure under weaker models/assumptions, provided one correspondingly weakens our extremely demanding simulation security definition.

More generally, while the ROM model is obviously preferred to the GGM, practitioners do not mind relying on the GGM, provided it solves an interesting problem. Indeed, we can point to several examples of interesting primitives where standard analyses appear to be stuck, and the GGM provided meaningful answers to these questions. Most notably, Signal leverages in production a protocol which can only be proven secure in the GGM model to achieve group privacy [12, 43]. Other important examples include optimal structure-preserving signature schemes [1] and state-restoration soundness analysis of Bulletproofs [17].

2 Notation and Preliminaries

We use square brackets $[a_1, a_2, \ldots, a_n]$ to denote ordered lists of objects, and curly brackets $\{a, b, c, \ldots\}$ for sets. We represent maps $D = \{(a, b), (c, d), \ldots\}$ as sets of label-value pairs. If D is a set of pairs, we denote with $D_{(\cdot)} = \{a, c, \ldots\}$ the set of the first components of each pair (the domain of the corresponding map), and with $D^{(\cdot)} = \{b, d, \ldots\}$ the set of the second components (the range of the map). When clear from context, we slightly abuse notation and write $a \in D$ (instead of $a \in D^{(\cdot)}$) if there is a pair $(a, \cdot)$ in the set, and (when unique) we denote the corresponding value with $D[a]$. Similarly, we use $C[i]$ to denote the i-th element of list C (1-indexed), and $last(C)$ to denote its last element.

---

[16] We stress that we only use GGM for the ZK property of our construction. Our stronger extractability-based soundness is still proven in the random oracle model, and does not require the GGM.

We denote with λ the security parameter. Given two security games I and R, each parameterized by an algorithm $\mathcal{A}$ (the adversary), we define the advantage of $\mathcal{A}$ in distinguishing the two experiments as $\left| \Pr[I^{\mathcal{A}} = 1] - \Pr[R^{\mathcal{A}} = 1] \right|$. In each figure defining a security experiment, we denote with $\mathcal{A}^{\mathcal{O}\cdots}(a_1, \ldots, a_n)$ an execution of algorithm $\mathcal{A}$ on input $a_1, \ldots, a_n$ with access to all the oracles defined in that figure.

We use the following conventions to describe algorithms. When a hash function takes more than one input (or a pair), we assume that there is a well defined way to serialize and deserialize such a tuple into a bitstring. Given a boolean b, we use **ensure** b as shorthand for "if not b, **return** 0". We use "**parse** a as $(a_1, \ldots, a_n)$" to denote that an algorithm tries to unpack a tuple of objects, and if the tuple does not have the appropriate length the algorithm returns a dummy output/error. In a security game, we use "**assert** b" to denote that if b is false, the experiment is immediately terminated with a special return value $\perp$; during an oracle call, we use "**require** b" to indicate that if b is false the oracle call by the adversary is interrupted without output, and any effects on the state of this call are reverted.

In Appendix A, we recall the Diffie-Hellman assumption, and briefly discuss the Random Oracle Model and Generic Group Model assumptions that our work depends on.

3 Rotatable Zero Knowledge Set

In this section, we formally define Rotatable Zero Knowledge Sets (RZKS). The primitive Zero-Knowledge Set was introduced in [9,33] and extended to append-only ZKS (aZKS) in [8]. We extend the notion of aZKS from SEEMless to add new properties as well as strengthen the soundness guarantees in our new primitive: RZKS.

Definition 1 A Rotatable Zero Knowledge Set (RZKS) consists of a tuple of algorithms $\mathcal{Z} = (\mathcal{Z}.\mathsf{GenPP}, \mathcal{Z}.\mathsf{Init}, \mathcal{Z}.\mathsf{Update}, \mathcal{Z}.\mathsf{PCSUpdate}, \mathcal{Z}.\mathsf{VerifyUpd}, \mathcal{Z}.\mathsf{Query}, \mathcal{Z}.\mathsf{Verify}, \mathcal{Z}.\mathsf{ProveExt}, \mathcal{Z}.\mathsf{VerExt})$ defined as follows:

▷ $\mathsf{pp} \leftarrow \mathcal{Z}.\mathsf{GenPP}(1^\lambda)$: This algorithm takes the security parameter and produces public parameter pp for the scheme. All other algorithms take these pp as input implicitly, even when not explicitly specified.

▷ $(\mathsf{com}, \mathsf{st}) \leftarrow \mathcal{Z}.\mathsf{Init}(\mathsf{pp})$: This algorithm takes as input the public parameters, and produces a commitment com to an empty datastore $\mathsf{D}_0 = \{\}$ and an initial server/prover state st. A datastore D will be a collection of $(\mathsf{label}_i, \mathsf{val}_i, t)$ tuples, where t is an integer indicating that the tuple has been added to the datastore as part of the t-th Update or PCSUpdate operation (we call this an epoch). Labels will be unique across the datastore (it can be thought of as a map). Each server state st will contain a datastore and a digest, which we will refer to as $\mathbf{D}(\mathsf{st})$ and $\mathbf{com}(\mathsf{st})$. Similarly, each commitment will include the epoch $\mathbf{t}(\mathsf{com})$ of the datastore to which it is referring. (Alternatively, these can be thought of as deterministic functions which are part of the scheme.)

▷ $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathcal{Z}.\mathsf{Update}(\mathsf{pp}, \mathsf{st}, S)$, $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathcal{Z}.\mathsf{PCSUpdate}(\mathsf{pp}, \mathsf{st}, S)$: Both algorithms take in the public parameters, the current state of the prover st, and a list $S = \{(\mathsf{label}_1, \mathsf{val}_1), (\mathsf{label}_2, \mathsf{val}_2), \ldots, (\mathsf{label}_n, \mathsf{val}_n)\}$ of new (label, value) pairs to insert (the labels must be unique and not already part of $\mathbf{D}(\mathsf{st})$). The algorithm outputs an updated commitment to the datastore, an updated internal state st', and a proof π_S that the update has been done correctly. Intuitively, com' is a commitment to the updated datastore $\mathbf{D}(\mathsf{st}')$ at epoch $\mathbf{t}(\mathsf{st}') = \mathbf{t}(\mathsf{st}) + 1$, which extends $\mathbf{D}(\mathsf{st})$ by also mapping each label_i in S to the pair $(\mathsf{val}_i, \mathbf{t}(\mathsf{st}'))$. As we will see, Update and PCSUpdate have different tradeoffs between their efficiency and the privacy guarantees they offer.

▷ $0/1 \leftarrow \mathcal{Z}.\mathsf{VerifyUpd}(\mathsf{pp}, \mathsf{com}_{t-1}, \mathsf{com}_t, \pi_S)$: This deterministic algorithm takes in two commitments to the datastore output at successive invocations of Update, and verifies the above proof.

▷ $(\pi, \mathsf{val}, t) \leftarrow \mathcal{Z}.\mathsf{Query}(\mathsf{pp}, \mathsf{st}, u, \mathsf{label})$: This algorithm takes as input a state st, an epoch $u \leq \mathbf{t}(\mathsf{st})$, and a label. If a tuple $(\mathsf{label}, \mathsf{val}, t) \in \mathbf{D}(\mathsf{st})$ and $t \leq u$, it returns val, t and a proof π. Else, it returns $\mathsf{val} = \perp$, $t = \perp$ and a non-membership proof π. In both cases, proofs are meant to be verified against the commitment com_u output during the u-th update.

▷ $1/0 \leftarrow \mathcal{Z}.\mathsf{Verify}(\mathsf{pp}, \mathsf{com}, \mathsf{label}, \mathsf{val}, t, \pi)$: This deterministic algorithm takes a $(\mathsf{label}, \mathsf{val}, t)$ tuple, and verifies the proof π with respect to the commitment com. If $\mathsf{val} = \perp$ and $t = \perp$, this is considered a proof that label is not part of the data structure at epoch $\mathbf{t}(\mathsf{com})$.

▷ $\pi_E \leftarrow \mathcal{Z}.\mathsf{ProveExt}(\mathsf{pp}, \mathsf{st}, t_0, t_1)$: This algorithm takes the state of the prover and two epochs t_0, t_1, and returns a proof π_E that the datastore after the t_1-th update is an extension of the datastore after the t_0-th update. Proofs are meant to be verified against the commitments com_{t_0} and com_{t_1} output by Update during the t_0-th and t_1-th update.

▷ $1/0 \leftarrow \mathcal{Z}.\mathsf{VerExt}(\mathsf{pp}, \mathsf{com}_{t_0}, \mathsf{com}_{t_1}, \pi_E)$: This deterministic algorithm takes two datastore commitments and a proof (generated by ProveExt) and verifies it.

We require a RZKS to satisfy the following security properties:

Completeness We will say that an RZKS satisfies completeness if for all PPT adversaries $\mathcal{A}$, the probability that the game described in Figure 1 outputs 0 is negligible in λ.

Intuitively, all updates and queries should behave as expected by their descriptions in the definition. Furthermore, all proofs produced by various updating or querying algorithms should verify when properly queried to the corresponding verification algorithms. More formally, an adversary only breaks completeness if it is able to construct a sequence of queries such that one of the assertions in Figure 1 fails. For example, the assertion $\mathbf{D}(\mathsf{st}') = \mathbf{D}(\mathsf{st}) \cup \{(\mathsf{label}_i, \mathsf{val}_i, t+1)\}_{i \in [j]}$ in $\mathsf{Update}(S)$ will only fail if the elements added in S are not correctly added to the state of the datastore. Similarly, in $\mathsf{Query}(\mathsf{label}, u)$ we assert that $\mathsf{P}.\mathsf{Verify}(\mathsf{com}_u, \mathsf{label}, \mathsf{val}', t', \pi)$ succeeds, where (val', t', π) are those produced by the corresponding call to P.Query.

P-Completeness($\mathcal{A}$):

$\mathsf{pp}' \leftarrow \mathsf{P}.\mathsf{GenPP}(1^\lambda)$
$(\mathsf{com}', \mathsf{st}') \leftarrow \mathsf{P}.\mathsf{Init}(\mathsf{pp}')$
assert $\mathbf{com}(\mathsf{st}') = \mathsf{com}'$ and $\mathbf{t}(\mathsf{com}') = 0$ and $\mathbf{D}(\mathsf{st}') = \{\}$
$\mathsf{com}_0 \leftarrow \mathsf{com}', \mathsf{st} \leftarrow \mathsf{st}', t \leftarrow 0, \mathsf{pp} \leftarrow \mathsf{pp}'$
$\mathcal{A}^{\mathcal{O}\cdots}(\mathsf{pp}, \mathsf{com}_0)$
return 1

Oracles $\mathsf{Update}(S)$ and $\mathsf{PCSUpdate}(S)$:

parse S as $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_j, \mathsf{val}_j)$
require $\mathsf{label}_1, \ldots, \mathsf{label}_j$ are distinct and do not already appear in $\mathbf{D}(\mathsf{st})$
$(\mathsf{com}', \mathsf{st}', \pi) \leftarrow \mathsf{P}.\mathsf{Update}(\mathsf{st}, S)$ // resp. $\mathsf{P}.\mathsf{PCSUpdate}(\mathsf{st}, S)$
assert $\mathbf{com}(\mathsf{st}') = \mathsf{com}'$ and $\mathbf{t}(\mathsf{com}') = t+1$ and $\mathbf{D}(\mathsf{st}') = \mathbf{D}(\mathsf{st}) \cup \{(\mathsf{label}_i, \mathsf{val}_i, t+1)\}_{i \in [j]}$
assert $y \leftarrow \mathsf{P}.\mathsf{VerifyUpd}(\mathsf{com}_t, \mathsf{com}', \pi); \ y = 1$
$\mathsf{com}_{t+1} \leftarrow \mathsf{com}', \mathsf{st} \leftarrow \mathsf{st}', t \leftarrow t+1$

Oracle $\mathsf{Query}(\mathsf{label}, u)$:

require $0 \leq u \leq t$
$(\pi, \mathsf{val}', t') \leftarrow \mathsf{P}.\mathsf{Query}(\mathsf{st}, u, \mathsf{label})$
If $\mathsf{label} \in \mathbf{D}(\mathsf{st}), (\mathsf{val}_\mathsf{D}, u_\mathsf{D}) \leftarrow \mathbf{D}(\mathsf{st})[\mathsf{label}]$ and $u_\mathsf{D} \leq u$:
    assert $(\mathsf{val}', t') = (\mathsf{val}_\mathsf{D}, u_\mathsf{D})$
Else
    assert $(\mathsf{val}', t') = (\perp, \perp)$
assert $y \leftarrow \mathsf{P}.\mathsf{Verify}(\mathsf{com}_u, \mathsf{label}, \mathsf{val}', t', \pi); \ y = 1$

Oracle $\mathsf{ProveExt}(t_0, t_1)$: // RZKS only

require $0 \leq t_0 \leq t_1 \leq t$
$\pi_E \leftarrow \mathsf{P}.\mathsf{ProveExt}(\mathsf{st}, t_0, t_1)$
assert $y \leftarrow \mathsf{P}.\mathsf{VerExt}(\mathsf{com}_{t_0}, \mathsf{com}_{t_1}, \pi_E); \ y = 1$

Oracle $\mathsf{ProveAll}(t')$: // OA only

$\pi \leftarrow \mathsf{P}.\mathsf{ProveAll}(\mathsf{st}, t')$
assert $y \leftarrow \mathsf{P}.\mathsf{VerAll}(\mathsf{com}_{t'}, \mathbf{D}(\mathsf{st})_{\leq t'}, \pi); \ y = 1$

Fig. 1: Completeness for RZKS and OA (an Ordered Accumulator, defined in Section 5.1) primitives (denoted with P). Some of the oracles are only applicable to one primitive. In this experiment, the adversary can read all the game's state and the oracle's intermediate variables, such as $\mathsf{com}_i \forall i, \mathsf{st}, y$. The experiment returns 1 unless one of the assertions is triggered. These checks enforce that the data structure is updated consistently, that the outputs of query reflect the state of the data structure, and that honestly generated proofs pass verification as intended.

Soundness We will say that an RZKS satisfies soundness if there exists an extractor Extract such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the two experiments described in Figure 2 is negligible in λ. Note that all the algorithms executed in the experiment get implicit access to the Ideal oracle, as they might need to make, e.g., random oracle calls.

The extractor Extract is required to provide various functionalities based on its first input:

- $\mathsf{pp}', \mathsf{st} \leftarrow$ Extract(Init): Samples public parameters indistinguishable from honestly generated public parameters such that extraction will be possible. Also generates an initial state.
- $D_{\mathsf{com}} \leftarrow$ Extract(Extr, st, com): Takes in the internal state and a commitment to the datastore. Outputs the set of (label, val, i) committed to.
- $C_{\mathsf{com}} \leftarrow$ Extract(ExtrC, st, com): Takes in the internal state and a commitment to the datastore. Outputs the set of previous commitments, indexed by epoch.
- $out, \mathsf{st} \leftarrow$ Extract(Ideal, st, in): Simulates the behavior of some ideal functionality (for example a random oracle or generic group). Takes in any input and produces an output indistinguishable from the output the ideal functionality would have on that input.

One small subtlety of the definition here is that we do not allow the extractor to update its state outside of Ideal calls. The only advantage that the extractor gets over an honest party is its control over the ideal functionality. This allows for easier composition, since a larger primitive utilizing RZKS will not need to simulate extractor state.

An adversary breaks soundness if it either distinguishes answers to Ideal queries in the real game from those produced by the extractor, or if it causes some assertion to be false in the ideal game. Each assertion in the ideal game captures some way in which the extractor could be caught in an inconsistent state. For example, let us consider the assertion $\mathsf{D[com][label]} = (\mathsf{val}^*, i^*)$ in CheckVerD. This will be false if the adversary can provide a proof that (label, val^*, i^*) is in the datastore with digest com, but the extractor expects this datastore to either not contain label or to contain (label, val, i) for some different (val, i).

Our soundness definition strengthens the traditional one by providing extractability. aZKS soundness already guarantees that a (malicious) prover is unable to produce two verifying proofs for two different values for the same label with respect to an aZKS commitment it has already produced. However, that definition does not guarantee that the malicious prover knew the entire collection of (label, value) pairs at the time it produced the commitment. Extractability requires that by mandating that the entire datastore can be extracted from the commitment, except with negligible probability.

We also explicitly guarantee consistency among the RZKS commitments produced over epochs. Informally, consistency guarantees that each commitment to an epoch also binds the server to all previous commitments (i.e. these can be extracted from the former). In particular, when the client swaps a commitment com^a with a more recent one com^b by verifying an extension proof, and then checks with an auditor that com^b is legitimate, the client can be sure that any auditor who checked all consecutive audit proofs up to com^b must also have checked the same com^a for epoch a. This is modeled in the security game by the assertions in the ExtractC, CheckVerUpdC, and CheckVerExt oracles.

Zero Knowledge We will say that an RZKS is zero knowledge for leakage function $L = (L_{\mathsf{Update}}, L_{\mathsf{PCSUpdate}}, L_{\mathsf{Query}}, L_{\mathsf{ProveExt}}, L_{\mathsf{LeakState}})$ if there exists a simulator $\mathcal{S}$ such that every PPT malicious client algorithm $\mathcal{A}$ has negligible advantage in distinguishing the two experiments of Figure 3.

The stateful simulator $\mathcal{S}$ is required to provide various functionalities:

- $\mathsf{com}', \mathsf{pp}' \leftarrow \mathcal{S}(\mathtt{Init})$: Samples public parameters and an initial commitment indistinguishable from honest public parameters such that it will be possible to simulate proofs.
- $(\mathsf{com}', \pi) \leftarrow \mathcal{S}((\mathtt{PCS})\mathtt{Update}, l)$: Takes in some leakage l about an Update (or, analogously, PCSUpdate) query on input S, i.e. in the experiment $l \leftarrow L_{\mathsf{Update}}(S)$ (or $l \leftarrow L_{\mathsf{PCSUpdate}}(S)$). Outputs a commitment com' indistinguishable from a commitment to the previous datastore with the elements of S appended. Furthermore simulates a proof π that the update was done correctly.
- $(\pi, \mathsf{val}', t') \leftarrow \mathcal{S}(\mathtt{Query}, l)$: Takes in leakage $l \leftarrow L_{\mathsf{Query}}(u, \mathsf{label})$ about the entry indexed by (u, label) in the datastore. Outputs val', t' which would have been returned by an honest query. Also simulates a proof π that $\mathsf{D[label]} = (\mathsf{val}', t')$, or an absence proof if $\mathsf{label} \notin \mathsf{D}$.

```
P-Sound-IDEAL(𝒜):                                        Oracle CheckVerUpdD(comᵃ, comᵇ, π):

pp′, st ← Extract(Init)                                   require P.VerifyUpd(pp, comᵃ, comᵇ, π) = 1 and
D ← {}, C ← [ ], pp ← pp′                                     comᵃ, comᵇ ∈ D
b ← 𝒜^{Ideal(·),⋯}(pp)                                    assert D[comᵃ] ⊆ D[comᵇ], and
return b                                                     t(comᵇ) = t(comᵃ) + 1, and
                                                             ∀(label, val, t) ∈ D[comᵇ] \ D[comᵃ] :
                                                                 t = t(comᵇ), and
Oracle ExtractD(com):                                        (t(comᵃ) ≠ 0 or D[comᵃ] = {})

D_com ← Extract(Extr, st, com)
If com ∈ D assert D[com] = D_com                          Oracle CheckVerUpdC(comᵃ, comᵇ, π): // RZKS only
D[com] ← D_com
assert ∀ (label, val, i) ∈ D[com] : 0 < i ≤ t(com)       require P.VerifyUpd(pp, comᵃ, comᵇ, π) = 1 and
                                                             comᵃ, comᵇ ∈ C
                                                          assert t(comᵇ) = t(comᵃ) + 1, and
Oracle ExtractC(com): // RZKS only                           ∀ j ≤ t(comᵃ) : C[comᵃ][j] = C[comᵇ][j]

C_com ← Extract(ExtrC, st, com)
If com ∈ C assert C[com] = C_com                          Oracle CheckVerExt(comᵃ, comᵇ, π): // RZKS only
C[com] ← C_com
assert |C[com]| = t(com) and last(C[com]) = com          require P.VerExt(pp, comᵃ, comᵇ, π) = 1 and
                                                             comᵃ, comᵇ ∈ C
                                                          assert ∀ j ≤ t(comᵃ) : C[comᵃ][j] = C[comᵇ][j]

Oracle CheckVerD(com, label, val*, i*, π):
                                                          Oracle CheckVerAll(com, S, π): // OA only
require P.Verify(pp, com, label, val*, i*, π) = 1 and com ∈ D
If val* = ⊥ or i* = ⊥:                                    require P.VerAll(pp, com, S, π) = 1 and com ∈ D
    assert label ∉ D[com] ∧ val* = i* = ⊥                 assert D[com] = S
Else assert D[com][label] = (val*, i*)

                                                          Oracle Ideal(in):

                                                          out, st ← Extract(Ideal, st, in)
                                                          return out
```

Fig. 2: Soundness for RZKS and OA (both denoted by P). In the ideal world, the map D stores, for each commitment com, the datastore that the Extract algorithm output for that commitment. In addition the map C stores, for each commitment, the (ordered) list of commitments to previous epochs. When the adversary provides proofs, we require that the proofs are consistent with such data structures. In the real world (not pictured), the public parameters are generated as pp ← P.GenPP(1^λ), and all the oracles do nothing and return no output, except for the Ideal oracle, which implements the ideal objects (such as random oracles) that we abstract to prove security of the primitives (and that are controlled by the extractor in the ideal world). In both cases, P's algorithms implicitly get access to the Ideal oracle as needed.

- $\pi \leftarrow \mathcal{S}(\texttt{ProveExt}, l)$: Takes in partial information $l \leftarrow L_{\mathsf{ProveExt}}(t_0, t_1)$ from a ProveExt query the between epochs t_0 and t_1. Outputs an extension proof that the commitment provided at epoch t_1 binds to the one at epoch t_0.
- $\mathsf{st} \leftarrow \mathcal{S}(\texttt{Leak}, l)$: Takes in partial information $l \leftarrow L_{\mathsf{LeakState}}()$ about the datastore and outputs a simulated state consistent with the information given.
- $out \leftarrow \mathcal{S}(\texttt{Ideal}, in)$: Simulates the behavior of some ideal functionality. Takes in any input and produces an output indistinguishable from the output the ideal functionality would have on that input.

Note that the particular leakage given will be construction specific, but should be designed to be as minimal as possible. Our choice of leakage will be described in detail in Section 5.3. In the experiment, the only information the simulator has access to is the output of the leakage function, as well as the queries made to the Ideal oracle. The simulator's ability to control the ideal oracle is crucial for security proofs to go through.

Informally, zero knowledge here means that the proofs generated by any sequence of honest calls to RZKS algorithms can be simulated given access to minimal information about the queries made. The adversary breaks zero knowledge if it is able to generate a sequence of queries such that it can distinguish the output of the simulator from honestly generated outputs and proofs. For example, if the simulator is unable to simulate query proofs, then an adversary could succeed by calling the Update({label, val}) oracle for some (label, val), then the $(\pi, \mathsf{val}, 1) \leftarrow$ Query(label, 1) oracle, and running RZKS.Verify on π. Since the simulator can't simulate query proofs, π generated in the ideal world will not verify and so will be distinguished from π generated in the real world.

Post-compromise security is modelled by allowing for LeakState calls, which reveal the state in its entirety. When the adversary queries this oracle, the simulator is required to output a state that appears consistent with whatever proofs it has revealed before. Healing from compromise is modelled by having a dedicated leakage function for PCSUpdate (different from Update). Note that since all the leakage functions share state, calling LeakState or PCSUpdate might affect the leakage of other future queries.

RZKS-ZK-REAL($\mathcal{A}$):

$\mathsf{pp}' \leftarrow \mathcal{Z}.\mathsf{GenPP}(1^\lambda)$
$(\mathsf{com}', \mathsf{pp}', \mathsf{st}') \leftarrow \mathcal{Z}.\mathsf{Init}(\mathsf{pp}')$
$\mathsf{st} \leftarrow \mathsf{st}', t \leftarrow 0, \mathsf{pp} \leftarrow \mathsf{pp}'$
$b \leftarrow \mathcal{A}^{\mathsf{Update}(\cdot),\cdots}(\mathsf{com}', \mathsf{pp})$
return b

Update(S): // analogous for PCSUpdate

parse S as $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_j, \mathsf{val}_j)$
require $\mathsf{label}_1, \ldots, \mathsf{label}_j$ are distinct and do not already appear in $\mathbf{D}(\mathsf{st})$
$(\mathsf{com}', \mathsf{st}', \pi) \leftarrow \mathcal{Z}.\mathsf{Update}(\mathsf{st}, S)$
$\mathsf{st} \leftarrow \mathsf{st}', t \leftarrow t + 1$
return (com', π)

Query(label, u):

require $0 \leq u \leq t$
$(\pi, \mathsf{val}', t') \leftarrow \mathcal{Z}.\mathsf{Query}(\mathsf{pp}, \mathsf{st}, u, \mathsf{label})$
return (π, val', t')

ProveExt(t_0, t_1):

require $0 \leq t_0 \leq t_1 \leq t$
$\pi \leftarrow \mathcal{Z}.\mathsf{ProveExt}(\mathsf{pp}, \mathsf{st}, t_0, t_1)$
return π

LeakState():

return st

Ideal(in):

return $\mathsf{Ideal}(in)$

RZKS-ZK-IDEAL($\mathcal{A}$):

$\mathsf{com}', \mathsf{pp}' \leftarrow \mathcal{S}(\mathtt{Init})$
$t \leftarrow 0$
$b \leftarrow \mathcal{A}^{\mathsf{Update}(\cdot),\cdots}(\mathsf{com}', \mathsf{pp}')$
return b

Update(S): // analogous for PCSUpdate

parse S as $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_j, \mathsf{val}_j)$
require $\mathsf{label}_1, \ldots, \mathsf{label}_j$ are distinct and do not already appear in any of the $S_1, \ldots, S_t$
$(\mathsf{com}', \pi) \leftarrow \mathcal{S}(\mathtt{Update}, L_{\mathsf{Update}}(S))$
$t \leftarrow t + 1, S_t \leftarrow S$
return (com', π)

Query(label, u):

require $0 \leq u \leq t$
$(\pi, \mathsf{val}', t') \leftarrow \mathcal{S}(\mathtt{Query}, L_{\mathsf{Query}}(u, \mathsf{label}))$
return (π, val', t')

ProveExt(t_0, t_1):

require $0 \leq t_0 \leq t_1 \leq t$
$\pi \leftarrow \mathcal{S}(\mathtt{ProveExt}, L_{\mathsf{ProveExt}}(t_0, t_1))$
return π

LeakState():

return $\mathcal{S}(\mathtt{Leak}, L_{\mathsf{LeakState}}())$

Ideal(in):

return $\mathcal{S}(\mathtt{Ideal}, in)$

Fig. 3: Zero Knowledge (with leakage) security experiments for RZKS. $\mathcal{S}$ is a stateful algorithm (whose state we omit to simplify the notation). The leakage functions $L_{\mathsf{Update}}, L_{\mathsf{Query}}, \ldots$ also share state among each other.

3.1 Application to Key Transparency

Recall that in an aZKS, the value associated with each label cannot be updated: the prover can only add new (label, value) pairs to the directory. In SEEMless [8], the server uses aZKS to commit to its public key directory by setting the label to (userID || version number) and value to the public key of the user corresponding to that ID. Every update to the underlying public key directory becomes a new label addition to the aZKS. The server collects a batch of these additions and periodically updates the directory, creating a new epoch and publishing a new aZKS commitment. Clients must hold on to all previous commitments until they have double-checked them with the auditors (to ensure that the server is not violating the append-only property and that every client is seeing the same commitments). If clients want to retain the ability to hold the server accountable even if auditors are temporarily offline, or if they wish to do the audit themselves in the future, they need to hold on to all the commitments indefinitely, which is inefficient. To solve this problem, SEEMless suggests building a hashchain over all the aZKS commitments, so that the client only needs to remember the tail. This is an improvement, but to skip between two distant commitments, the client has to download all the epochs in between; moreover, the security guarantees deriving from this are not formalized. In contrast, we propose a more efficient solution and formalize its security: we add the ProveExt and VerExt algorithms, which allow the server to directly prove that any given datastore commitment stems from another.

Thus, our advantage over SEEMless lies both in the fact that we give the ability to heal from server state compromise and that we allow the client to only keep the very latest commitment, and to efficiently update to the next one without losing the ability to hold the server accountable later.

4 Rotatable Verifiable Random Functions

In this section, we introduce the notion of a Rotatable Verifiable Random Function, a key component of our RZKS construction. Verifiable Random Functions (VRFs), introduced in [36], are the asymmetric analogues of Pseudorandom Functions: the secret key is necessary to compute the (random-looking) function on any input, as well as a proof that the computation was performed correctly, which can be checked against the corresponding public key. We extend VRFs by adding "rotation" algorithms, which generate a new VRF key pair alongside zero-knowledge proofs that outputs of the new and old VRF on the same (hidden) input are associated. In addition, our rotatable VRFs also satisfy stricter soundness properties.

Definition 2 A Rotatable Verifiable Random Function is a tuple of algorithms VRF = (GenPP, KeyGen, Query, Verify, Rotate, VerRotate) defined as follows:

▷ $pp \leftarrow$ VRF.GenPP(1^λ): This algorithm takes the security parameter and produces public parameter pp for the scheme. All other algorithms take these pp as input, even when not explicitly specified.

▷ $(sk, pk) \leftarrow$ VRF.KeyGen(pp): The key generation algorithm takes in the global pp and outputs the public key pk and secret key sk.

▷ $(y, \pi) \leftarrow$ VRF.Query(pp, sk, x): The query algorithm takes in pp, the secret key sk and input x, and outputs the evaluation y of the VRF defined by sk on input x, as well as a proof π. We denote with VRF.Eval(sk, x) the first output y of the Query algorithm (i.e. Eval does not return a proof).

▷ $1/0 \leftarrow$ VRF.Verify(pp, pk, x, y, π): This deterministic function verifies the proof π that y is the output of the VRF defined by pk on input x.

▷ $sk', pk', \pi \leftarrow$ VRF.Rotate(pp, sk, X): Given a secret key[17] and a list of inputs X, this algorithm outputs an updated secret key, an updated public key, and a proof π that the set of VRF output pairs $P = \{(\text{VRF.Eval}(sk, x), \text{VRF.Eval}(sk', x))\}_{x \in X}$ satisfies the relationship that each pair corresponds to the same input x (without leaking information about X beyond its size).

▷ $0/1 \leftarrow$ VRF.VerRotate(pp, pk, pk', P, π): Given two public keys pk, pk' and list of P pairs (y, y'), this deterministic algorithm checks the proof π that each pair consists of the output of the VRFs identified by pk, pk' on the same input x.

For correctness, we require that for all $\lambda, n \in \mathbb{N}$, all sets of inputs $X_1, \ldots, X_n$, and all inputs x:

$$\Pr[pp \leftarrow \text{VRF.GenPP}(1^\lambda);\ sk_0, pk_0 \leftarrow \text{VRF.KeyGen}(pp);$$
$$sk_i, pk_i, \pi_i \leftarrow \text{VRF.Rotate}(sk_{i-1}, X_i) \text{ for } i = 1, \ldots, n;$$
$$y, \pi \leftarrow \text{VRF.Query}(sk_n, x) : \text{VRF.Verify}(pk_n, x, y, \pi) = 1] = 1.$$

Moreover, for all $\lambda, n > 0$ and all sets of inputs $X_1, \ldots, X_n$:

$$\Pr[pp \leftarrow \text{VRF.GenPP}(1^\lambda);\ sk_0, pk_0 \leftarrow \text{VRF.KeyGen}(pp);$$
$$sk_i, pk_i, \pi_i \leftarrow \text{VRF.Rotate}(sk_{i-1}, X_i) \text{ for } i = 1, \ldots, n : \text{VRF.VerRotate}(pk_n,$$
$$\{(\text{VRF.Eval}(sk_{n-1}, x), \text{VRF.Eval}(sk_n, x))\}_{x \in X_n}, \pi_n) = 1] = 1.$$

4.1 Rotatable VRF Security

Informally, VRFs satisfy two properties. *Uniqueness* mandates that for any public key and input x, there is only one y which can be proven to be output by the function on input x. *Pseudorandomness* guarantees that, for an honestly generated key pair sk, pk and given oracle access to the query oracle on arbitrary inputs, it is hard to distinguish the output of the function on any other (not yet queried) input from a uniformly random value.

We augment the uniqueness and pseudorandomness requirements into soundness and zero-knowledge respectively.

---

[17] Given that the old key sk and new key are independent from one another, we could have equivalently defined Rotate as taking any two secret keys as input.

Soundness (strengthened uniqueness) We will say that a VRF satisfies soundness if there exists an Extractor such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the experiments of Figure 4 is negligible.

The extractor Extract is required to provide three functionalities based on its first input:

- $\mathsf{pp}, \mathsf{st} \leftarrow \mathsf{Extract}(\texttt{Init})$: Samples public parameters indistinguishable from honestly generated public parameters such that extraction will be possible. Also generates an initial state.
- $x \leftarrow \mathsf{Extract}(\texttt{Extr}, \mathsf{st}, pk, y)$: Takes in an adversarially chosen public key pk and output y of the function. Outputs the only input x for which the adversary can produce an accepting proof.
- $out, \mathsf{st} \leftarrow \mathsf{Extract}(\texttt{Ideal}, \mathsf{st}, in)$: Simulates the behavior of some ideal functionality (for example a random oracle or generic group). Takes in any input and produces an output indistinguishable from the output the ideal functionality would have on that input.

As with RZKS, we do not allow the extractor to update its state outside $\texttt{Ideal}$ calls.

In the ideal experiment, the table T keeps track of the outputs of the extractor. An assertion is triggered (and the adversary can trivially win) if the extractor gives different answers to the same query over time, if the same answer is returned for multiple inputs under the same public key, or if the adversary produces an accepting proof for an input different than what the extractor had predicted (these requirements together capture uniqueness). Moreover, the game also enforces that proofs of rotation are consistent with the extractor's output and the equality condition is respected. In the real experiment, assertions are never triggered, so indistinguishability ensures that public parameters, as well as the answers to ideal queries, give the adversary the same view.

Zero Knowledge (strengthened pseudorandomness) We will say that a VRF satisfies zero-knowledge if there exists a simulator such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the experiments of Figure 5 is negligible.

The stateful simulator $\mathcal{S}$ is required to provide various functionalities:

- $\mathsf{pp}, pk_0 \leftarrow \mathcal{S}(\texttt{Init})$: Samples public parameters and an initial public key such that it will be possible to simulate proofs.
- $y \leftarrow \mathcal{S}(\texttt{Corrupted-Eval}, i, x)$: Takes in a corrupted generation i and input x, and outputs the evaluation of the VRF on i and x. If this is called, the adversary has already obtained the corresponding secret key for generation i, so the simulator is forced to output a value consistent with what the adversary could compute itself.
- $\pi \leftarrow \mathcal{S}(\texttt{Explain}, i, \mathsf{label}, y)$: Takes in a generation, input, and output. Outputs a simulated proof that the output of the oracle $\mathsf{Eval}(i, x) = y$.
- $pk_{i_{\mathsf{cur}}}, \pi_R \leftarrow \mathcal{S}(\texttt{Rotate}, P)$: Takes in a set P of pairs (y, y'). Samples a new public key $pk_{i_{\mathsf{cur}}}$ and outputs a simulated proof that for each $(y, y') \in P$ there exists an x such that $\mathsf{Eval}(i_{\mathsf{cur}-1}, x) = y$ and $\mathsf{Eval}(i_{\mathsf{cur}}, x) = y'$.
- $sk_{i_{\mathsf{crpt}}+1}, \ldots, sk_{i_{\mathsf{cur}}} \leftarrow \mathcal{S}(\texttt{Corrupt}, D)$: Takes in all queries made to Eval. Outputs a collection of secret keys consistent with output of all oracle queries made so far.
- $out \leftarrow \mathcal{S}(\texttt{Ideal}, in)$: Simulates the behavior of some ideal functionality. Takes in any input and produces an output indistinguishable from the output the ideal functionality would have on that input.

We combine pseudorandomness with a zero knowledge requirement by requiring that in each generation a simulator can sample public parameters such that it can simulate proofs that the VRF is consistent with a new truly random function. Furthermore, the simulator must be able to simulate rotation proofs that the outputs of two random functions stem from the same input. We model post compromise security by requiring that the simulator also be able to sample secret keys consistent with all previous queries. Since it is impossible to sample a secret key consistent with all future queries for a truly random function, after corruption we give the simulator the ability to control the function associated with that epoch. Note that the major difficulty in demonstrating zero knowledge is that the simulator must simulate queries to the ideal oracle without knowing what inputs are asked of the truly random function.

We remark that our definition of zero knowledge is heavily inspired by the notion of a simulatable VRF, introduced in [10]. Simulatable VRFs require that there exists a simulator that can sample simulated public parameters such that for any public key pk, input x in the domain, and y in the range of the VRF, it is possible to simulate a proof π that y is the output of the function on input x (i.e. $\mathsf{Verify}(\mathsf{pp}, pk, x, y, \pi) = 1$). The simulated parameters, outputs and proofs should be indistinguishable from honestly generated ones. Our definition of zero knowledge extends this notion by accounting for rotation proofs and corruptions. Our soundness notion is also stronger as we require extractability.

$\underline{\mathsf{VRF\text{-}Sound\text{-}IDEAL}(\mathcal{A})\text{:}}$

$T \leftarrow [\,]; \mathsf{pp}, \mathsf{st} \leftarrow \mathsf{Extract}(\mathtt{Init})$
$b \leftarrow \mathcal{A}^{\mathcal{O}\cdots}(pp)$
$\mathbf{return}\ b$

$\underline{\mathsf{Oracle\ Extract}(pk, y)\text{:}}$

$x \leftarrow \mathsf{Extract}(\mathtt{Extr}, \mathsf{st}, pk, y)$
If $(pk, y) \in T$ $\mathbf{assert}\ T[pk, y] = x$
$\mathbf{assert}\ x = \bot \lor \forall y' \neq y : T[pk, y'] \neq x$
$T[pk, y] \leftarrow x$

$\underline{\mathsf{Oracle\ CheckExtraction}(pk, y, x, \pi)\text{:}}$

$\mathbf{require}\ \mathsf{VRF.Verify}(pk, x, y, \pi) = 1 \land (pk, y) \in T$
$\mathbf{assert}\ T[pk, y] = x$

$\underline{\mathsf{Oracle\ CheckVerRotate}(pk_1, pk_2, P, \pi)\text{:}}$

$\mathbf{require}\ \mathsf{VRF.VerRotate}(pk_1, pk_2, P, \pi) = 1 \land$
$\quad \forall (u_1, u_2) \in P : (pk_1, u_1) \in T \land (pk_2, u_2) \in T$
$\mathbf{assert}\ \forall (u_1, u_2) \in P : T[pk_1, u_1] = T[pk_2, u_2]$

$\underline{\mathsf{Oracle\ Ideal}(in)\text{:}}$

$out, \mathsf{st} \leftarrow \mathsf{Extract}(\mathtt{Ideal}, \mathsf{st}, in)$
$\mathbf{return}\ out$

Fig. 4: Soundness for VRF. In the real world (not pictured), the public parameters are generated as $\mathsf{pp} \leftarrow \mathsf{VRF.GenPP}(1^\lambda)$, and the oracles do not do anything, except for the Ideal one which implements the necessary ideal objects according to their specification.

4.2 Rotatable VRF Construction

Our rotatable Verifiable Random Function $\mathsf{VRF} = (\mathsf{GenPP}, \mathsf{KeyGen}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Rotate}, \mathsf{VerRotate})$ is instantiated in Figure 6. In summary, let G be a group of (exponential) prime order p with generator g, and let $F(x)$ be a hash function that maps arbitrary-length bitstrings onto G. Then for a given input x, secret key $sk \in \mathbb{Z}_p^*$, and public key $pk = g^{sk}$, the VRF output is $y = F(x)^{sk}$. To prove this, Query simply produces a Fiat-Shamir zero-knowledge proof that $(g, F(x), pk = g^{sk}, y = F(x)^{sk})$ is a DDH tuple.

Given secret key $sk = \alpha_0 \cdot \cdots \cdot \alpha_i$ and public key $g^{\alpha_0 \cdots \alpha_i}$, Rotate samples α_{i+1} from $\mathbb{Z}_p^*$. It then sets $sk' = \alpha_0 \cdot \cdots \cdot \alpha_{i+1}$ and stores $pk' = pk^{\alpha_{i+1}} = g^{\alpha_0 \cdots \alpha_{i+1}} = g^{sk'}$. Then, it outputs a "batch" Fiat-Shamir zero-knowledge proof that (pk, y, pk', y') is a DDH tuple, where y and y' are random linear combinations of $\mathsf{VRF.Eval}(sk, x)$ and $\mathsf{VRF.Eval}(sk', x)$ for $x \in X$, respectively. In Figure 6, the coefficients for the random linear combination are derived as a_u.

4.3 Rotatable VRF Soundness Proof

Soundness of extraction stems directly from soundness of the underlying Fiat-Shamir proof that $(g, F(x), pk = g^{sk}, y = F(x)^{sk})$ is a DDH tuple. To show soundness of rotation, again we use the fact that the underlying Fiat-Shamir proof that (pk, y, pk', y') is a DDH tuple is sound. The only subtlety is to show that batching the rotation proofs in the manner we do works. That is, we need to show that if (y, y') are a random linear combination of $\{(\mathsf{VRF.Eval}(sk, x), \mathsf{VRF.Eval}(sk', x))\}_{x \in X}$, then if $y' = y^\alpha$, with all but negligible probability we also have $\mathsf{VRF.Eval}(sk', x) = \mathsf{VRF.Eval}(sk, x)^\alpha$ for all $x \in X$.

```
VRF-ZK-REAL(𝒜):

pp ← VRF.GenPP(1^λ)
sk_0, pk_0 ← VRF.KeyGen(pp)
i_cur ← 0, i_crpt ← −1
b ← 𝒜^{𝒪···}(pp, pk_0)
return b


Oracle Eval(i, x):

require 0 ≤ i ≤ i_cur
y ← VRF.Eval(sk_i, x)
return y


Oracle Prove(i, x):

require 0 ≤ i ≤ i_cur
y, π ← VRF.Query(sk_i, x)
return y, π


Oracle Rotate(X):

sk_{i_cur+1}, pk_{i_cur+1}, π_R ← VRF.Rotate(sk_{i_cur}, X)
i_cur ← i_cur + 1
return pk_{i_cur}, π_R


Oracle Corrupt():

return sk_{i_crpt+1}, ..., sk_{i_cur}
i_crpt ← i_cur


Oracle Ideal(in):

return Ideal(in)
```

```
VRF-ZK-IDEAL(𝒜):

pp, pk_0 ← 𝒮(Init)
i_cur ← 0, i_crpt ← −1, D = {0 : {}}
b ← 𝒜^{𝒪···}(pp, pk_0)
return b


Oracle Eval(i, x):

require 0 ≤ i ≤ i_cur
If i ≤ i_crpt:
    return 𝒮(Corrupted-Eval, i, x)
If x ∉ D[i]:
    y ←$ Y; D[i][x] ← y
return D[i][x]


Oracle Prove(i, x):

require 0 ≤ i ≤ i_cur
π ← 𝒮(Explain, i, x, Eval(i, x))
return Eval(i, x), π


Oracle Rotate(X):

i_cur ← i_cur + 1
P ← {(Eval(i_cur − 1, x), Eval(i_cur, x)) | x ∈ X}
pk_{i_cur}, π_R ← 𝒮(Rotate, P)
return pk_{i_cur}, π_R


Oracle Corrupt():

sk_{i_crpt+1}, ..., sk_{i_cur} ← 𝒮(Corrupt, D)
return sk_{i_crpt+1}, ..., sk_{i_cur}
i_crpt ← i_cur


Oracle Ideal(in):

return 𝒮(Ideal, in)
```
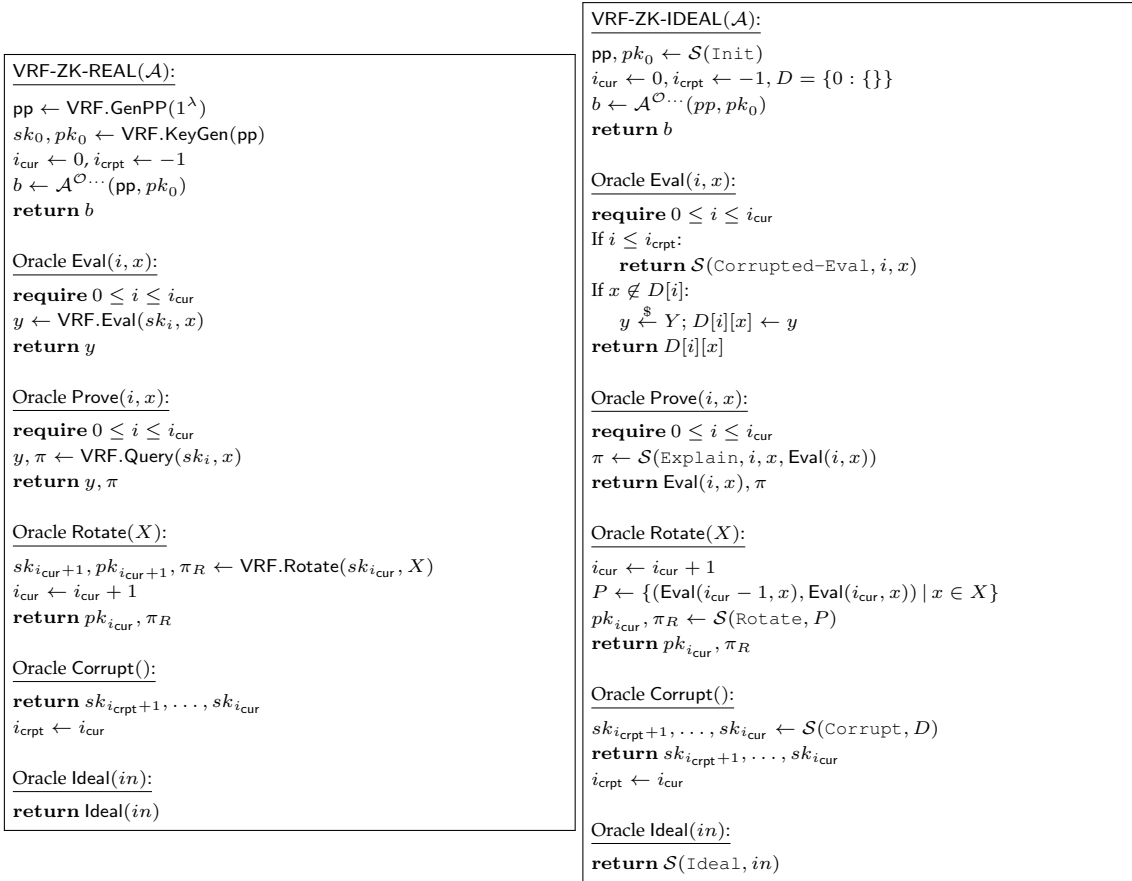
Fig. 5: Zero Knowledge experiments for the Rotatable VRF.

Taking the contrapositive, we just need to show that if there is any (y_0, y_0') in $\{(\mathsf{VRF.Eval}(sk, x), \mathsf{VRF.Eval}(sk', x))\}_{x \in X}$ such that $y_0' \neq y_0^\alpha$, then the probability that a random linear combination (y, y') satisfies $y' = y^\alpha$ must be negligible. Note that if there exists a pair (y_1, y_1') in $\{(\mathsf{VRF.Eval}(sk, x), \mathsf{VRF.Eval}(sk', x))\}_{x \in X}$ such that (y_0, y_0') and (y_1, y_1') are linearly independent as elements of $G \times G$, then (y, y') will be uniformly random and so will satisfy $y' = y^\alpha$ with only negligible probability. But if there is no such pair, then $(y, y') = (y_0^c, y_0'^c)$ for some c, so $y' = y^\alpha$ with probability 1.

Theorem 1 *If F and F' are modeled as random oracles, and if the DDH assumption holds, then there exists a simulator* Extract *such that for any efficient adversary $\mathcal{A}$,*

$$|\Pr[\mathsf{VRF\text{-}Sound\text{-}REAL}(\mathcal{A}) \to 1] - \Pr[\mathsf{VRF\text{-}Sound\text{-}IDEAL}(\mathcal{A}) \to 1]| \leq negl(\lambda).$$

Lemma 1. *If the DDH assumption holds on the group G, then the Fiat-Shamir proof used in our construction that (g, h, g', h') is a DDH tuple is sound.*

Proof of lemma Soundness of the interactive version of this proof is proven in [13]. Soundness of the final proof then comes from correctness of the Fiat-Shamir heuristic in the random oracle model [40].

Proof of theorem We need to define an extractor Extract and show that it makes the ideal experiment indistinguishable from the real one for any PPT adversary $\mathcal{A}$.

Extract's state consists of a table T which tracks the one maintained by the game, as well as a tables D and D' used to implement random oracles F and F' respectively. In particular, table $D' \subset \{0,1\}^* \times \mathbb{Z}_p$ is used to directly store the (uniformly sampled) answers to F' queries (whose range is $\mathbb{Z}_p$), while table $D \subset \{0,1\}^* \times \mathbb{Z}_p$ will store the discrete logarithms of the answers to F queries. Since our extractor won't ever store two tuples $(x, r), (x', r')$ in D such that $x = x'$ or $r = r'$ (i.e. we are simulating F as

```
▷ pp ← VRF.GenPP(1^λ):                           ▷ sk', pk', π ← VRF.Rotate(pp, sk, X):
  – p ← prime exponential in λ                     – parse pp as (p, G, g, F, F')
  – G ← group of order p                           – α ←$ Z_p^*
  – g ← generator of G                             – sk' ← sk · α
  – Sample hash function F : {0,1}* → G            – pk' ← g^{sk'}
  – Sample hash function F' : {0,1}* → Z_p         – P ← {(VRF.Eval(sk, x), VRF.Eval(sk', x))}_{x∈X}
  – return pp ← (p, G, g, F, F')                   – For each (u, u') ∈ P:
                                                      • a_u ← F'(u, u', pk, pk', P)
▷ (sk, pk) ← VRF.KeyGen(pp):                        – y ← ∏_{(u,u')∈P} u^{a_u}
  – parse pp as (p, G, g, F, F')                    – y' ← ∏_{(u,u')∈P} (u')^{a_u}
  – α_0 ←$ Z_p^*                                    – r ←$ Z_p
  – sk ← α_0                                        – c ← F'(pk, y, pk', y', pk^r, y^r)
  – pk ← g^{α_0}                                    – z ← r − cα
  – return (sk, pk)                                 – π ← (pk^r, y^r, z)
                                                    – return (sk', pk', π)
▷ (y, π) ← VRF.Query(pp, sk, x):
  – parse pp as (p, G, g, F, F')                  ▷ 0/1 ← VRF.VerRotate(pp, pk, pk', P, π):
  – y ← F(x)^{sk}                                   – parse pp as (p, G, g, F, F')
  – r ←$ Z_p                                        – ensure pk, pk' ≠ g^0
  – c ← F'(g, F(x), g^{sk}, F(x)^{sk}, g^r, F(x)^r) – For each (u, u') ∈ P:
  – z ← r − c · sk                                     • a_u ← F'(u, u', pk, pk', P)
  – π ← (g^r, F(x)^r, z)                            – y ← ∏_{(u,u')∈P} u^{a_u}
  – return (y, π)                                   – y' ← ∏_{(u,u')∈P} (u')^{a_u}
                                                    – parse π as (h_1, h_2, z)
▷ 1/0 ← VRF.Verify(pp, pk, x, y, π):               – c ← F'(pk, y, pk', y', h_1, h_2)
  – parse pp as (p, G, g, F, F')                    – ensure h_1 = pk^z · (pk')^c
  – ensure pk ≠ g^0                                 – ensure h_2 = y^z · (y')^c
  – parse π as (h_1, h_2, z)                        – return 1
  – c ← F'(g, F(x), pk, y, h_1, h_2)
  – ensure h_1 = g^z · pk^c
  – ensure h_2 = F(x)^z · y^c
  – return 1
```

Fig. 6: Our Rotatable VRF construction.

an injective function), then if $(x, r) \in D$, we denote $D_x := r$ and $D^r := x$. We also define $D_{(\cdot)}$ to be the projection of D onto $\{0, 1\}^*$, and we define $D^{(\cdot)}$ to be the projection of D onto $\mathbb{Z}_p$.

Extract works as follows:

– `Ideal` queries for $F'(x)$ are answered implementing an random oracle honestly, i.e. sampling outputs at random the first time they are queried and storing them in table D' so that repeated queries get the same answers.
– To answer `Ideal` queries for $F(x)$:
  • If $x \in D_{(\cdot)}$, $r \leftarrow D_x$, **return** g^r
  • $r \stackrel{\$}{\leftarrow} \mathbb{Z}_p$
  • **assert** $r \notin D^{(\cdot)}$, and for all $((pk, y), x) \in T : pk^r \neq y$
  • Store (x, r) in D
  • **return** g^r and the updated state
– To answer `Extr` queries for pk, y:
  • If there exists $(x, r) \in D$ such that $pk^r = y$, add $((pk, y), x)$ to T and **return** x
  • **return** $\perp$

Claim: For any PPT adversary $\mathcal{A}$, if at any point during an execution of VRF-Sound-IDEAL($\mathcal{A}$) with Extract (or any of the hybrids $Hyb0, \ldots, Hyb4$) there exist $((pk, y), x') \in T$ and $(x, r) \in D$ such that $pk^r = y$, then $x = x'$ (i.e. $T[pk, y] = x$).

Proof of claim: Let $((pk, y), x') \in T$ and $(x, r) \in D$ such that $pk^r = y$. Since $((pk, y), x') \in T$, then at some point Oracle Extract(pk, y) was called. Note that after this call, $((pk, y), x') \in T$. By the definition of F, for every call to F after this point, if F stores a new pair (x', r') in D, it must be the case that $pk^{r'} \neq y$. Thus, it must be the case that $(x, r) \in D$ before Oracle Extract(pk, y) was ever called. By construction, when Extract(`Extr`, st, pk, y) is called inside of Oracle Extract(pk, y), it must return x. Therefore, $T[pk, y] = x$.

We then proceed by a sequence of hybrids, starting with $Hyb0 = \mathsf{VRF\text{-}Sound\text{-}IDEAL}(\mathcal{A})$ and ending with $\mathsf{VRF\text{-}Sound\text{-}REAL}(\mathcal{A})$, and show that the advantage of any adversary in distinguishing a pair of consecutive hybrids is negligible.

- $Hyb0$. Defined as $\mathsf{VRF\text{-}Sound\text{-}IDEAL}(\mathcal{A})$ with Extract as defined above.
- $Hyb1$. Defined as $Hyb0$, but we add an extra **require** statement to Oracle CheckExtraction(pk, y, x, π), so that it behaves as follows:
    - **require** $\mathsf{VRF.Verify}(pk, x, y, \pi) = 1 \wedge (pk, y) \in T$
    - **require** $(g, F(x), pk, y)$ is a DDH-tuple.
    - **assert** $T[pk, y] = x$

  Lemma 2. $|\Pr[Hyb1 \to 1] - \Pr[Hyb0 \to 1]| \leq negl(\lambda)$.

  Proof: This follows directly from Lemma 1. Note that $\mathsf{VRF.Verify}$ checks that π is a proof claiming $(g, F(x), pk, y)$ is a DDH-tuple. Since π comes from an efficient adversary interacting with efficient oracles, if $\mathsf{VRF.Verify}(pk, x, y, \pi) = 1$, then with all but negligible probability $(g, F(x), pk, y)$ is a DDH-tuple. QED.
- $Hyb2$. Defined as $Hyb1$, but we replace Oracle CheckExtraction(pk, y, x, π) with the real version, which does nothing.

  Lemma 3. $\Pr[Hyb2 \to 1] = \Pr[Hyb1 \to 1]$.

  Proof: Let L be the claim "$(pk, y) \in T$ and $(g, F(x), pk, y)$ is a DDH-tuple". If we show that $L \Rightarrow T[pk, y] = x$, then we are done (as the assertion will never be triggered). Note that $(g, F(x), pk, y)$ is a DDH-tuple if and only if there exists some $r \in \mathbb{Z}_p$ such that $F(x) = g^r$ and $y = pk^r$. Thus, $L \Rightarrow$ there exists some $(x, r) \in D$ such that $pk^r = y$. Due to the claim, since $L \Rightarrow (pk, y) \in T$, we have $L \Rightarrow T[pk, y] = x$. QED.
- $Hyb3$. Defined as $Hyb2$, but we modify Oracle CheckVerRotate(pk_1, pk_2, P, π) by adding an extra **require** statement, so that it behaves as follows:
    - **require** $\mathsf{VRF.VerRotate}(pk_1, pk_2, P, \pi) = 1 \wedge \forall (u_1, u_2) \in P : (pk_1, u_1) \in T \wedge (pk_2, u_2) \in T$
    - **require** $\forall (u_1, u_2) \in P : (pk_1, u_1, pk_2, u_2)$ is a DDH-tuple.
    - **assert** $\forall (u_1, u_2) \in P : T[pk_1, u_1] = T[pk_2, u_2]$

  Lemma 4. $|\Pr[Hyb3 \to 1] - \Pr[Hyb2 \to 1]| \leq negl(\lambda)$.

  Proof:
  This is slightly more involved than the argument for CheckExtraction, since we also need to show that batching DDH tuple proofs in the way we want works. Note that any adversary distinguishing between $Hyb2$ and $Hyb3$ must by necessity find some pk_1, pk_2, P, π such that $\mathsf{VerRotate}(pk_1, pk_2, P) = 1$ but for some u_1, u_2, (pk_1, u_1, pk_2, u_2) is not a DDH tuple.
  Recall that VerRotate first checks $pk_1, pk_2 \neq g^0$. It then samples $a_{u_1} \leftarrow F'(u_1, u_2, pk_1, pk_2, P)$ for each $(u_1, u_2) \in P$ and sets y_1, y_2 to be the linear combinations

  $$y_1 \leftarrow \prod_{(u_1, u_2) \in P} u_1^{a_{u_1}}$$

  and

  $$y_2 \leftarrow \prod_{(u_1, u_2) \in P} u_2^{a_{u_1}}.$$

  It finally checks that π is a proof that (pk_1, y_1, pk_2, y_2) is a DDH tuple. Observe that when $pk_1, pk_2 \neq g^0$, we can write $pk_2 = pk_1^\alpha$ for some α.
  In order for an adversary to succeed with non-negligible probability, it must find $(pk_1, pk_1^\alpha, P, \pi)$ such that either there exists some $(u \neq g^0, u^\beta) \in P$ where $\beta \neq \alpha$, but still $y_1^\alpha = y_2$, or π is a false proof that $(pk_1, y_1, pk_1^\alpha, y_2)$ is a DDH-tuple. Due to Lemma 1, since the adversary is efficient and interacts with efficient oracles, the probability that it can succeed at the latter task is negligible in λ. Thus, it remains to be shown that the probability that any adversary can succeed at the first task is negligible.

Consider any given (pk_1, pk_1^α, P) with $(u \neq g^0, u^\beta) \in P$ for some $\beta \neq \alpha$. We will show that the probability over the choice of F' that $y_2 = y_1^\alpha$ is negligible. We will divide this into two cases. The first case will be where the exponents of P are all the same. Formally, for all $(u_1, u_2) \in P$, $u_2 = u_1^\beta$. The second case will thus be when there exists some $(u' \neq g^0, u'^{\beta'}) \in P$ such that $\beta' \neq \beta$.

In the first case,

$$y_1 = \prod_{(u, u^\beta) \in P} u^{a_u},$$

$$y_2 = \prod_{(u, u^\beta) \in P} u^{\beta a_u},$$

$$y_1^\alpha = \prod_{(u, u^\beta) \in P} u^{\alpha a_u}.$$

Thus, $y_1^\alpha = y_2$ if and only if $\sum_{(u, u^\beta) \in P} \alpha a_u = \sum_{(u, u^\beta) \in P} \beta a_u$. As $\alpha \neq \beta$, this happens if and only if $\sum_{(u, u^\beta) \in P} a_u = 0$. Since when F' is random so is each a_u, this occurs with negligible probability over the choice of F'.

In the second case, let us write $u = g^c$ and $u' = g^d$. Since $\beta \neq \beta'$ and $c, d \neq 0$, $\{(c, \beta c), (d, \beta' d)\}$ forms a basis for $\mathbb{Z}_p^2$. Thus, if $a_u, a_{u'}$ are uniformly random, then $(a_u c + a_{u'} d, a_u \beta c + a_{u'} \beta' d)$ is a uniformly random element of $\mathbb{Z}_p^2$. Similarly, $(u^{a_u} u'^{a_{u'}}, u^{\beta a_u} u'^{\beta' a_{u'}})$ is a pair of uniformly random group elements. But note that since these terms appear in the products defining y_1 and y_2, and since multiplication by any group element is a permutation, when a_u and $a_{u'}$ are uniformly random, (y_1, y_2) is a pair of uniformly random group elements. Thus, the probability over the choice of F' that $y_2 = y_1^\alpha$ is negligible.

Therefore, since each $a_{u_1} = F'(u_1, u_2, pk_1, pk_2, P)$, each a_u is uniformly random until the adversary queries F' on pk_1, pk_2, P. Therefore, since the adversary makes only polynomial number of queries to F', the probability that the attacker finds $(pk_1, pk_1^\alpha, P, \pi)$ with $(u \neq g^0, u^\beta) \in P$, $\alpha \neq \beta$ such that $y_1^\alpha = y_2$ is negligible. QED.

– $Hyb4$. Defined as $Hyb3$, but we replace Oracle CheckVerRotate(pk_1, pk_2, P, π) with the real version, which does nothing.

Lemma 5. $\Pr[Hyb4 \to 1] = \Pr[Hyb3 \to 1]$.

Proof: Let L_{y_1, y_2} be the claim "$(pk_1, y_1) \in T$, $(pk_2, y_2) \in T$, and (pk_1, y_1, pk_2, y_2) is a DDH-tuple". We will begin by showing $L_{y_1, y_2} \Rightarrow T[pk_1, y_1] = T[pk_2, y_2]$. First, note that if $T[pk_1, y_1] = x$ for some $x \neq \perp$, then by construction of Extract there must be some $r \in \mathbb{Z}_p$ such that $(x, r) \in D$ and $pk_1^r = y_1$. But then since (pk_1, y_1, pk_2, y_2) is a DDH-tuple, we have $pk_2^r = y_2$. But note that then we have $(x, r) \in D$, $pk_2^r = y_2$, and $(pk_2, y_2) \in T$, so by the claim $T[pk_2, y_2] = x$. Thus,

$$L_{y_1, y_2} \wedge T[pk_1, y_1] = x \Rightarrow T[pk_2, y_2] = x.$$

Similarly, we can show

$$L_{y_1, y_2} \wedge T[pk_2, y_2] = x \Rightarrow T[pk_1, y_1] = x.$$

Thus, if L_{y_1, y_2} holds and one of $T[pk_1, y_1]$ or $T[pk_2, y_2]$ is defined, they are equal. If $T[pk_1, y_1] = \perp$ and $T[pk_2, y_2] = \perp$ then they are also equal. Therefore, $L_{y_1, y_2} \Rightarrow T[pk_1, y_1] = T[pk_2, y_2]$.

Thus, for all $(y_1, y_2) \in P$, if the **require** statements hold, then $(pk_1, y_1) \in T \wedge (pk_2, y_2) \in T \Rightarrow T[pk_1, y_1] = T[pk_2, y_2]$. Therefore, in $Hyb3$ the **assert** statement will never fail, and so $Hyb4 = Hyb3$. QED.

– $Hyb5$. Defined as $Hyb4$, but we remove the assertions from Oracle Extract (while still executing the extractor).

Lemma 6. $\Pr[Hyb5 \to 1] = \Pr[Hyb4 \to 1]$.

Proof: First, note that by construction the extractor always returns the same answer to the same query, so the first assertion cannot be triggered. For the second one, consider the case where $x \neq \perp$. Note that $T[pk, y] = x$ only if for some r, $pk^r = y$ and $(x, r) \in D$. Thus, if $T[pk, y'] = x$ for some y', then $(x, r') \in D$ such that $pk^{r'} = y'$. But since D does not allow duplicates, $r = r'$, and so $pk^r = y' = y$. Therefore, there will never be a collision, and so the second **assert** statement will never be triggered. QED.

– $Hyb6$. Defined as $Hyb5$, but we remove the assertion from the random oracle F. That is, we replace F with the following:

  • If $x \in D_{(\cdot)}$, $r \leftarrow D_x$, **return** g^r.
  • $r \stackrel{\$}{\leftarrow} \mathbb{Z}_p$.
  • Store (x, r) in D.
  • **return** g^r and the updated state.

  Lemma 7. $|\Pr[Hyb6 \to 1] - \Pr[Hyb5 \to 1]| \leq negl(\lambda)$.

  Proof: To prove indistinguishability, it is enough to show that the assertion is only triggered with negligible probability. Since for every pair $(pk, y) \in T$ there is exactly one r' such that $pk^{r'} = y$, the number of values of r that would be rejected is at most $|D^{(\cdot)}| + |T|$. Thus, the probability that any given query to F fails is $\leq \frac{|D^{(\cdot)}| + |T|}{p} = negl(\lambda)$ since the numerator is polynomially bounded. QED.

– $Hyb6$. Defined as $Hyb5$, but we replace F with a standard random oracle, and replace <u>Oracle Extract</u> with the real version, which does nothing. Note that at this point we are fully in the real world, so $Hyb6 = \mathsf{VRF\text{-}Sound\text{-}REAL}(\mathcal{A})$.

  Lemma 8. $\Pr[Hyb6 \to 1] = \Pr[Hyb5 \to 1]$.

  Proof: Since g is a generator for G, the function $f : \mathbb{Z}_p \to G$ defined by $f(r) = g^r$ is a permutation. Thus, sampling $r \stackrel{\$}{\leftarrow} \mathbb{Z}_p$ and returning g^r is identically distributed to sampling $y \stackrel{\$}{\leftarrow} G$. Furthermore, since the database D is never used, no longer storing this discrete log will have no impact on the game. Calling the extractor on Extract queries also has no longer any impact on the adversary's view. Thus, $Hyb5$ and $Hyb6$ are identically distributed. QED.

Since $Hyb0 = \mathsf{VRF\text{-}Sound\text{-}IDEAL}(\mathcal{A})$, $Hyb6 = \mathsf{VRF\text{-}Sound\text{-}REAL}(\mathcal{A})$, we conclude that

$$|\Pr[\mathsf{VRF\text{-}Sound\text{-}REAL}(\mathcal{A}) \to 1] - \Pr[\mathsf{VRF\text{-}Sound\text{-}IDEAL}(\mathcal{A}) \to 1]| \leq negl(\lambda).$$

QED.

4.4 Rotatable VRF Zero Knowledge Proof

Since our construction generates zero-knowledge proofs in Prove and Rotate, one would hope that simulating these proofs would be enough to prove zero-knowledge of the construction. In fact, if there were no Corrupt oracle, then simply programming the random oracle F' would be enough to simulate these proofs and achieve zero-knowledge. However, once an adversary has called Corrupt and obtained some secret key sk_i, it can then easily distinguish previously outputted $\mathsf{Eval}(i, x)$ from the true VRF output $F(x)^{sk_i}$ by simply calculating $F(x)^{sk_i}$ itself and comparing the two.

This intuition extends to arbitrary simulation strategies. Consider for example an adversary who asks for $F(x)$ and $F(x')$, $y \leftarrow \mathsf{Eval}(i, x)$, $y' \leftarrow \mathsf{Eval}(i, x')$ in this order, for two distinct x, x' and some i. At the time of the F queries, the uniformly random outputs of the VRF have not yet been sampled, and so the simulator's output cannot depend on them. Once the adversary calls the Corrupt oracle, the simulator can produce a value for sk_i only if $\log_{F(x)}(y) = \log_{F(x')}(y')$, which only happens with negligible probability. While this specific problem could be solved by adding an additional hash at the end of the VRF computation, i.e. defining $\mathsf{VRF.Eval}(sk, x) = H(F(x)^{sk})$ as in [18] and treating H as a programmable random oracle, similar issues arise when considering our efficient rotation proofs, which would force the game to reveal preimages for the hash before the corruption happens (and so before the simulator knows what algebraic relations should exists between the outputs of F and the group elements revealed in rotation proofs).

To solve this problem, we need to treat G as a generic group. This allows the simulator to hold off on sampling sk_i until Corrupt is called. Until this point, the simulator will treat pk_i as an arbitrary group element, but it will keep track of all algebraic relationships between unknown arbitrary group elements. Then, when Corrupt is called, the simulator will have access to a list of all group elements h such that the adversary expects $h = g^{f(sk_i)}$ for some function f of sk_i. At this point, the simulator will choose sk_i uniformly at random, and can program the generic group such that $g^{f(sk_i)} = h$ for all such f.

Theorem 2 *If the group G is modeled as a generic group, and F, F' are modeled as random oracles, then there exists a simulator $\mathcal{S}$ such that for any efficient $\mathcal{A}$,*

$$|\Pr[\mathsf{VRF\text{-}ZK\text{-}REAL}(\mathcal{A}) \to 1] - \Pr[\mathsf{VRF\text{-}ZK\text{-}IDEAL}(\mathcal{A}) \to 1]| \leq negl(\lambda).$$

Proof. We want to prove that, for the VRF construction we defined, there exists a simulator which makes the real VRF ZK game indistinguishable from the ideal one for any poly-time adversary. We will be operating under the model where G is a generic group and F, F' are random oracles. We will do this by showing a sequence of hybrids, where we will start with VRF-ZK-REAL($\mathcal{A}$). In the last hybrid, we will explicitly define the simulator $\mathcal{S}$.

Let $\mathcal{A}$ be any PPT adversary. We will define $q_{\mathcal{O}}$ to be a bound on the queries made by $\mathcal{A}$ to the oracle $\mathcal{O}$. For example, q_{Rotate} is a bound on the queries made by $\mathcal{A}$ to the Eval oracle. We will write the generic group $G = (S, \tau, \star, EXP, g = \tau(1))$, where S is the set of strings representing group elements, τ is the permutation mapping $\mathbb{Z}_p$ to S, $\star$ and EXP are the group operation and exponentiation oracles (accessible to the adversary through Ideal oracle queries), and g is the group generator (see Section A.3 for details).

- *Hyb0*. This is defined as VRF-ZK-REAL($\mathcal{A}$).
- *Hyb1*. Defined as the previous hybrid, but with the random oracles F, F' acting lazily. That is, each random oracle internally stores a table of queried values. When a fresh query x is made to a random oracle, it samples $y \xleftarrow{\$} S$ and stores (x, y) in its table. When x is queried again, the random oracle returns y from its table. We also define q_F and $q_{F'}$ to be upper bounds on the sizes of the tables used to implement F and F' in all the hybrids of this proof. Note that we can bound these quantities based on the number of queries the adversary makes (not just to the Ideal oracle for F and F', but also the other ones), which is polynomial in the security parameter (as $\mathcal{A}$ is PPT).

  Lemma 9. $\Pr[Hyb1 \to 1] = \Pr[Hyb0 \to 1]$.

  Proof: Since the output of this random oracle is identically distributed to the output of a true random oracle, $Hyb1 \equiv Hyb0$. QED.

- *Hyb2*. Defined as the previous hybrid, but with the generic group acting lazily, and F avoiding collisions as detailed below. Instead of being represented by a random permutation τ, G will internally store a table $T \subset \mathbb{Z}_p[[q_T]] \times S$. That is, T stores a table mapping polynomials over at most q_T variables to group elements $s \in S$. Here q_T is an upper bound on the number or rows of table T in any of the hybrids (as for q_F in $Hyb1$, this is polynomially bounded). We will define how T is generated so that it has no collisions in either index (i.e. it models an injective function). If $(P, h) \in T$, we denote $T_P := h$ and $T^h := P$, and due to the no-collision requirement these are uniquely defined. We will use $P \in T_{(\cdot)}$ to denote $\exists\, h$ such that $(P, h) \in T$, and we will use $h \in T^{(\cdot)}$ to denote $\exists\, P$ such that $(P, h) \in T$. At initialization, we sample $g \xleftarrow{\$} S$ and insert $(1, g)$ into T.
  In this hybrid, we implement $\star(g_1, g_2, b) \to h$ as follows:
  - If $g_1 \notin T^{(\cdot)}$, create a new variable B_1 and add (B_1, g_1) to T.
  - If $g_2 \notin T^{(\cdot)}$, create a new variable B_2 and add (B_2, g_2) to T.
  - $P_1 \leftarrow T^{g_1}$.
  - $P_2 \leftarrow T^{g_2}$.
  - If $P_1 + b \cdot P_2 \in T_{(\cdot)}$, **return** $T_{P_1 + b \cdot P_2}$.
  - $h \xleftarrow{\$} S \setminus T^{(\cdot)}$.
  - Set $T_{P_1 + b \cdot P_2} = h$ (i.e. add $(P_1 + b \cdot P_2, h)$ to T).
  - **return** h.
  We further implement $EXP(h, r) \to h'$ as follows:
  - If $h \notin T^{(\cdot)}$, create a new variable B and add (B, h) to T.
  - $P \leftarrow T^h$.
  - If $r \cdot P \in T_{(\cdot)}$, **return** $T_{r \cdot P}$.
  - $h' \xleftarrow{\$} S \setminus T^{(\cdot)}$.
  - Set $T_{r \cdot P} = h'$.
  - **return** h'.
  Moreover, when answering oracle queries for F, we avoid collisions with itself and with T. That is, when evaluating F on a fresh value x, instead of sampling $y \xleftarrow{\$} S$, F will sample $y \xleftarrow{\$} S \setminus (T^{(\cdot)} \cup TF^{(\cdot)})$ where TF is the table of F. It will then add (x, y) to TF, and (B_y, y) to T using a fresh formal variable B_y.

Lemma 10. $|\Pr[Hyb2 \to 1] - \Pr[Hyb1 \to 1]| \leq \frac{q_T^2}{p} + q_F \frac{q_T + q_F}{p}$.

Proof: Consider the situation where at the end of the game, all random variables B_i in T are sampled uniformly at random from $\mathbb{Z}_p$, and all polynomials in T are replaced with their evaluation on the B_i's. Let BAD_T be the event where this situation induces a collision in T. That is, BAD_T is the event where for some $(p_1, g_1), (p_2, g_2) \in T$, $p_1(B_1, \ldots, B_q) = p_2(B_1, \ldots, B_q)$. Also, let BAD_F be the event that in $Hyb1$, when answering a new query for F with an element which is already part of $TF^{(\cdot)}$ or $T^{(\cdot)}$. Note that if BAD_T does not occur, then the table of T at the end of the game is identically distributed to the list of queried input-output pairs to τ in $Hyb1$. Thus, if neither BAD_T not BAD_F occur, $\mathcal{A}$'s view is identical in $Hyb2$ and $Hyb1$. We can bound the probability that either event occurs with the sum of the probabilities that each occur independently.

First, note that the probability that F ever collides with T or TF in $Hyb2$ is $\Pr[BAD_F] \leq q_F \frac{q_T + q_F}{p}$ by the union bound. Then, we show that $\Pr[BAD_T] \leq \frac{q_T^2}{p}$. Note that the manipulations of polynomials in $\star$ and EXP will never cause the degree to exceed 1. $\star$ may create a new polynomial of degree 1, and then adds two polynomials together. EXP may create a new polynomial of degree 1, and then multiplies a polynomial by a scalar. Hence, by the Schwartz-Zippel lemma, the probability that a given pair of polynomials in T collide when evaluated is $\leq \frac{1}{p}$. Thus, by the union bound, $\Pr[BAD_T] \leq \frac{q_T^2}{p}$. QED.

– $Hyb3$. Defined as the previous hybrid, but we replace the honestly computed Fiat-Shamir DH proof in Prove with a simulated one. We also replace calls to VRF.Eval (for example, those executed inside VRF.Query or VRF.Rotate as part of Prove or Rotate queries) with calls to the Eval oracle (which itself still calls VRF.Eval). For example, we replace $(y, \pi) \leftarrow$ VRF.Query(pp, sk_i, x) as part of Prove oracle queries with the following:
  - **parse** pp as (p, G, g, F, F')
  - $y \leftarrow$ Eval(i, x)
  - $z \overset{\$}{\leftarrow} \mathbb{Z}_p$
  - $c \overset{\$}{\leftarrow} \mathbb{Z}_p$
  - Program $c = F'(g, F(x), pk_i, y, g^z \cdot pk_i^c, F(x)^z \cdot y^c)$
    // i.e., add $((g, \ldots, F(x)^z \cdot y^c), c)$ to TF' and fail if it creates a conflict.
  - $\pi \leftarrow (g^z \cdot pk_i^c, F(x)^z \cdot y^c, z)$
  - **return** (y, π)

Lemma 11. $|\Pr[Hyb3 \to 1] - \Pr[Hyb2 \to 1]| \leq \frac{q_{F'} \cdot q_{\mathsf{Prove}}}{p}$.

Proof: Replacing VRF.Eval calls with the Eval oracle is just a syntactic difference and doesn't change $\mathcal{A}$'s view, as at this point Eval itself calls VRF.Eval (and returns $F(x)_i^{sk}$), but this will make the exposition easier. Other than that, this hybrid is is just replacing the Fiat-Shamir DH proof with a standard simulation, which doesn't affect $\mathcal{A}$'s view either.

We sketch the proof here for completeness. Both in $Hyb2$ and $Hyb3$, (z, r, c) have a uniformly random distribution subject to the constraint that $r = z + c \cdot sk_i$ (where r is the discrete log of the first proof element). In $Hyb2$, we sample r and c and set z accordingly, while in $Hyb3$ we sample z and c and implicitly set the distribution on r. Indeed,

$$g^z \cdot pk_i^c = g^{r - c \cdot sk_i} \cdot g^{c \cdot sk_i} = g^r$$

and

$$F(x)^z \cdot y^c = F(x)^{r - c \cdot sk_i} \cdot F(x)^{c \cdot sk_i} = F(x)^r.$$

Thus, as long as we do not fail in programming c, π is distributed identically under $Hyb2$ and $Hyb3$. As long as the query to F' is always novel in $Hyb3$, programming succeeds and the output of Prove is identically distributed to $Hyb2$. Since z is uniformly random and multiplication by any group element is a permutation, $g^z \cdot pk_i^c$ is uniformly random. The probability that programming fails in any one particular iteration is bounded by the probability that $g^z \cdot pk_i^c$ was queried to F' before, which is $\leq \frac{q_{F'}}{p}$. Therefore, by the union bound, the advantage of any attacker in distinguishing $Hyb2$ and $Hyb3$ is $\leq \frac{q_{F'} \cdot q_{\mathsf{Prove}}}{p}$. QED.

– *Hyb*4. Defined as the previous hybrid, but where we simulate the Fiat-Shamir DH proof in Rotate, similarly to what we did in the previous hybrid for VRF.Query. In more detail, we replace $(pk_{i_{\text{cur}}}, \pi_R) \leftarrow$ Rotate(X) oracle queries with the following (expanding out VRF.Rotate inside):

- $i_{\text{cur}} \leftarrow i_{\text{cur}} + 1$
- **parse** pp as (p, G, g, F, F')
- $\alpha \xleftarrow{\$} \mathbb{Z}_p^*$
- $sk_{i_{\text{cur}}} \leftarrow sk_{i_{\text{cur}}-1} \cdot \alpha$
- $pk_{i_{\text{cur}}} \leftarrow g^{sk_{i_{\text{cur}}}}$
- $P \leftarrow \{\mathsf{Eval}(i_{\text{cur}} - 1, x), \mathsf{Eval}(i_{\text{cur}}, x)\}_{x \in X}$
- For each $(u, u') \in P$:
  * $a_u \leftarrow F'(u, u', pk, pk', P)$
- $y \leftarrow \prod_{(u,u') \in P} u^{a_u}$
- $y' \leftarrow \prod_{(u,u') \in P} (u')^{a_u}$
- $z \xleftarrow{\$} \mathbb{Z}_p$
- $c \xleftarrow{\$} \mathbb{Z}_p$
- Program $c = F'(pk_{i_{\text{cur}}-1}, y, pk_{i_{\text{cur}}}, y', pk_{i_{\text{cur}}-1}^z \cdot pk_{i_{\text{cur}}}^c, y^z \cdot y'^c)$
- $\pi \leftarrow (pk_{i_{\text{cur}}-1}^z \cdot pk_{i_{\text{cur}}}^c, y^z \cdot y'^c, z)$
- **return** $(pk_{i_{\text{cur}}}, \pi_X)$

Lemma 12. $|\Pr[Hyb4 \to 1] - \Pr[Hyb3 \to 1]| \leq \frac{q_{F'} \cdot q_{\text{Rotate}}}{p}$.

Proof: The argument here is the same as for the previous hybrid, since we are again replacing the Fiat-Shamir proof with its simulator. We only need to check that the algebra works out. Set $r = z + c \cdot \alpha$. Then we have

$$pk_{i_{\text{cur}}-1}^z \cdot (pk_{i_{\text{cur}}})^c = pk_{i_{\text{cur}}-1}^{r-c\cdot\alpha} \cdot (pk_{i_{\text{cur}}-1}^\alpha)^c = pk_{i_{\text{cur}}-1}^r$$

and

$$y^z \cdot y'^c = y^{r - c\cdot\alpha} \cdot (y^\alpha)^c = y^r,$$

which is the same as in the construction. Note again that $pk_{i_{\text{cur}}-1}^z \cdot (pk_{i_{\text{cur}}})^c$ is uniformly random, and so programming fails only with probability $\leq \frac{q_{F'} \cdot q_{\text{Rotate}}}{p}$. QED.

– *Hyb*5. Defined as the previous hybrid, but we now replace all α's in T with formal variables. We do this by adding a new oracle to the generic group, the random exponent oracle $h' \leftarrow RE(h)$:

- If $h \notin T$, then create a new variable B_h and store (B_h, h) in T.
- $p_h \leftarrow T^h$.
- Sample $h' \xleftarrow{\$} S \setminus T^{(\cdot)}$.
- Create a new variable A, and store $(p_h \cdot A, h')$ in T.
- **return** h'

We then replace all places where α appears with the random exponent oracle. In particular, we replace VRF.KeyGen with the following:

- **parse** pp as (p, G, g, F, F')
- $pk \leftarrow RE(g)$
- **return** (pk)

Similarly, when handling Rotate oracle queries, we replace the instructions $\alpha \xleftarrow{\$} \mathbb{Z}_p^*$, $sk_{i_{\text{cur}}} \leftarrow sk_{i_{\text{cur}}-1} \cdot \alpha$, $pk_{i_{\text{cur}}} \leftarrow g^{sk_{i_{\text{cur}}}}$ with $pk_{i_{\text{cur}}} \leftarrow RE(pk_{i_{\text{cur}}-1})$.

We also replace our reference to sk_i in $\mathsf{Eval}(i, x)$ with the polynomial indexing pk_i:

- $B_x \leftarrow T^{F(x)}$ // If x is not in TF yet, make an oracle query for $F(x)$ and update TF and T as defined in *Hyb*2.
- $p_i \leftarrow T^{pk_i}$.
- If $p_i \cdot B_x \in T_{(\cdot)}$, **return** $T_{p_i \cdot B_x}$
- Sample $v_{i,x} \xleftarrow{\$} S \setminus T^{(\cdot)}$. Store $(p_i \cdot B_x, v_{i,x})$ in T.
- **return** $v_{i,x}$

Note that now the only place sk is used is inside of Corrupt. Thus, we replace Corrupt with the following:

- For $k = i_{\mathsf{crpt}} + 1, \ldots, i_{\mathsf{cur}}$:
    * $\alpha_k \xleftarrow{\$} \mathbb{Z}_p^*$.
    * $sk_k \xleftarrow{\$} \prod_{i=1}^k \alpha_i$.
    * $A_k \leftarrow$ the new formal variable introduced when calling RE to obtain pk_k.
    * Replace all instances of A_k in T with α_k, and simplify all polynomials. If this causes collisions, fail.
- **return** $sk_{i_{\mathsf{crpt}}}, \ldots, sk_{i_{\mathsf{cur}}}$, and set $i_{\mathsf{crpt}} \leftarrow i_{\mathsf{cur}}$.

Lemma 13. $|\Pr[Hyb5 \to 1] - \Pr[Hyb4 \to 1]| \leq \frac{2q_T^2 \cdot (q_{\mathsf{Rotate}}+1)}{p}$.

Proof:
Note that the only times RE is called is when creating a new public key. Thus, this process adds in a new formal variable A_i for each public key pk_i and no other new formal variables. Consider the situation where at the end of the game, each A_i in T (which hasn't yet been replaced with an α_i value during a corrupt query) is sampled uniformly at random from $\mathbb{Z}_p^*$, and all polynomials in T are replaced with their partial evaluation on the A_i's. Note that if no collisions in T are induced by Corrupt or by this final sampling, at the end of the game T is identically distributed to T in $Hyb4$. Furthermore, each oracle in this game returns the corresponding value in T it would have returned in $Hyb4$. Thus, let BAD be the event where a collision is induced in T by some call to Corrupt or by this end-of-game replacement. It is clear that the advantage of any adversary distinguishing $Hyb4$ and $Hyb5$ is bounded by $\Pr(BAD)$.

Define BAD_i to be the event that BAD occurs during the i-th query to Corrupt. We will let the last BAD_i correspond to the end-of-game replacement. Given a polynomial P, define $\deg_A(P)$ to be the total degree of A_i variables in P (e.g. $\deg_A(5B_1A_2A_3 + 6A_4) = 2$). Let rot_i be the number of times RE was called between the i-th call to Corrupt (or the end of the game for the last i) and the previous one (or the beginning of the game for $i = 1$). Note that the only way the A-degree of any polynomial can increase is through a call to RE, and RE only increases the degree by 1. Therefore at the i-th time Corrupt is called, $\max_{P \in T_{(\cdot)}} \deg_A(P) \leq rot_i$. By Schwartz-Zippel and a union bound, the probability that any two polynomials in T collide upon a random evaluation of A_i variables is $\leq q_T^2 \frac{rot_i}{p-1}$.
Therefore, by the union bound,

$$\Pr(BAD) \leq \sum_i \frac{q_T^2 \cdot rot_i}{p-1} = \frac{q_T^2 \cdot (q_{\mathsf{Rotate}}+1)}{p-1} \leq \frac{2q_T^2 \cdot (q_{\mathsf{Rotate}}+1)}{p}.$$

QED.

– $Hyb6$. Defined as the previous hybrid, but instead of letting Eval look at the index of pk in T to use as the secret key, we have Eval use a dummy variable instead. We add a database D to Eval to track repeat queries, and for queries to corrupted secret keys, we answer honestly. In this hybrid, $\mathsf{Eval}(i, x)$ oracle queries are answered as follows:
  - If $i \leq i_{\mathsf{crpt}}$, **return** $F(x)^{sk_i}$ (by making a random oracle query to F and a call to EXP, updating T and TF, and returning the result).
  - If $(i, x) \in D$, **return** $D[i][x]$.
  - $B_x \leftarrow T^{F(x)}$ // If x is not in TF yet, make an oracle query for $F(x)$ and update TF and T as defined in $Hyb2$.
  - Let $y_{i,x}$ be a new variable.
  - Sample $v_{i,x} \xleftarrow{\$} S \setminus T^{(\cdot)}$. Store $(y_{i,x}, v_{i,x})$ in T.
  - Set $D[i][x] = y_{i,x}$.
  - **return** $v_{i,x}$.

At the beginning of Corrupt, we now replace the dummy variables from Eval with the polynomials indexing the corresponding public keys:
  - For each $(i, x) \in D$:
    * $v_{i,x} \leftarrow D[i][x]$

* $y_{i,x} \leftarrow T^{v_{i,x}}$
* $B_x \leftarrow T^{F(x)}$ // If x is not in TF yet, make an oracle query for $F(x)$ and update TF and T as defined in $Hyb2$.
* $p_i \leftarrow T^{pk_i}$.
* $p_{i,x} \leftarrow p_i \cdot B_x$
* Replace all instances of $y_{i,x}$ in T with $p_{i,x}$ and simplify.
* Continue executing Corrupt as defined in $Hyb5$.

Lemma 14. $\Pr[Hyb6 \rightarrow 1] = \Pr[Hyb5 \rightarrow 1]$.

Proof: Note that $p_{i,x}$ is identical to the polynomial representing $\mathsf{Eval}(i,x)$ in T in the previous hybrid. Thus, this hybrid simply involves replacing $p_{i,x}$ with a formal variable until Corrupt is called. Let BAD be the event that this new addition to Corrupt ever induces a collision in T. Note that if BAD does not occur, after the replacement in the beginning of Corrupt, T is identically distributed in $Hyb5$ and $Hyb6$, as are the previous outputs of Eval queries. Thus, we just need to show that $\Pr(BAD) = negl(\lambda)$. In fact, we will show that $\Pr(BAD) = 0$.

As before, we denote the new formal variable introduced when calling RE to obtain pk_i as A_i. All other formal variables (besides the variables of the form $y_{i,x}$) are associated with some group element m, and we will denote such a variable as B_m. Note that the only multiplications of polynomials in the experiments happen either in RE (which is only called on public keys, where a single monomial which is product of some A_j is multiplied by another A_j variable), or in the computation of $p_{i,x}$ in a Corrupt query, where a monomial product of A_j's is multiplied by a single formal variable B_x ($B_x \leftarrow T^F(x)$ is a single formal variable by construction, as it is generated fresh without collisions when we query F for x). Furthermore, any term in any polynomial of T not containing an A_i will have degree at most 1, since the only way to increase degree beyond 1 is by using RE.

We will have $p_i = sk_{i_{\mathsf{crpt}}} \cdot A_{j+1} \cdot \cdots \cdot A_i$ for each $i > i_{\mathsf{crpt}}$. Note that since we do not create formal variables $y_{i,x}$ for $i \leq i_{\mathsf{crpt}}$, and since all such formal variables have been removed by the end of the previous corrupt query, all replacements $y_{i,x} \rightarrow p_{i,x}$ will involve $i > i_{\mathsf{crpt}}$. But note that for $i > i_{\mathsf{crpt}}$, any two polynomials $p_{i,x}$ and $p_{i',x'}$ for $(i,x) \neq (i',x')$ will be distinct, as they would have a different $B_x, B_{x'}$ factors if $x \neq x'$, and at least a different A_j variable if $x = x'$ but $i \neq i'$.

Let P be a polynomial in T before corruption. Due to the arguments in the previous description, we can denote $P = P_A + P_B + P_y$ where P_A is the terms of P containing only A_i formal variables, P_B is the terms of P containing only B_x formal variables, and P_y is the terms of P containing only $y_{i,x}$ formal variables. Denote $\widetilde{P}$ to be P after the replacement $y_{i,x} \rightarrow p_{i,x}$ and simplification. Clearly $\widetilde{P} = P_A + P_B + \widetilde{P_y}$. Since each term in $\widetilde{P_y}$ comes from a single replacement $y_{i,x} \rightarrow p_{i,x}$, and since each $p_{i,x}$ is a distinct product of formal variables, there is a one-to-one correspondence between terms of $\widetilde{P_y}$ and P_y. But the terms in $\widetilde{P_y}$ contain both A_is and a B_x, and thus will not collide in terms of formal variables with any terms in P_A or P_B. Thus, there is a one-to-one correspondence between terms of $\widetilde{P}$ and terms of P. Therefore, if Q is another polynomial in T before collision, $\widetilde{P} = \widetilde{Q}$ implies that $P = Q$. Thus, replacement of $y_{i,x} \rightarrow p_{i,x}$ will not result in any collisions, and so $\Pr(BAD) = 0$. QED.

- $Hyb7$. Defined as in the previous hybrid, but we set Eval to be the Eval of the ideal game, with $\mathcal{S}(\texttt{Corrupted-Eval}, i, x)$ implemented as returning $F(x)^{sk_i}$. In particular $\mathsf{Eval}(i,x)$ might sample a value $v_{i,x}$ that already appears in T, and does not insert $v_{i,x}$ in T immediately, which means the experiment could sample the same $v_{i,x}$ again when handling other G queries. To handle that, if during a Corrupt query we have that for some $v_{i,x} \leftarrow D[i][x]$, it holds that $v_{i,x} \in T^{(\cdot)}$ and $T^{v_{i,x}}$ is not a single formal variable, the experiment fails.

Lemma 15. $|\Pr[Hyb7 \rightarrow 1] - \Pr[Hyb6 \rightarrow 1]| \leq \frac{q_T^2}{p}$.

Proof: The only difference between the two hybrids is that in $Hyb6$, for $i > i_{\mathsf{crpt}}$, $\mathsf{Eval}(i,x)$ immediately samples $B_x, y_{i,x}, v_{i,x}, v_x$ and adds (x, v_x) to TF (i.e. $F(x) = v_x$), and (B_x, v_x) and $(y_{i,x}, v_{i,x})$ to T (if they do not exist). Instead, in $Hyb7$, the creation of all these values is delayed until either one of v_x or $v_{i,x}$ is passed as input to the generic group oracles, or Corrupt is called. Note that in either case (as long as the $v_{i,x}$ sampled by $\mathsf{Eval}(i,x)$ does not collide with an element of T), the new entries added for these values will be indexed by a pure formal variable, just as in the previous hybrid, so this delayed sampling has no effect on the adversary's view.

The only thing left to quantify is the probability that the $v_{i,x}$ sampled by Eval collides with an element of T, either one already in T when $v_{i,x}$ is sampled, or one sampled after as a result of a $\star$, EXP or F query. The probability that these collisions occur at any point during $Hyb7$ is $\leq \frac{q_T^2}{p}$ (note that q_T upper bounds $|T|$ in $Hyb6$, which also takes into account all the $v_{i,x}$ sampled during Eval queries which in $Hyb7$ do not immediately result in new entries to T). QED.

– $Hyb8$. The ideal experiment, with $\mathcal{S}$ defined as in Figure 7.

Lemma 16. $\Pr[Hyb8 \to 1] = \Pr[Hyb7 \to 1]$.

Upon inspection, we observe that $Hyb8$ is functionally identical to $Hyb7$.

Putting together all the hybrids, we see that

$$|\Pr[Hyb8 \to 1] - \Pr[Hyb0 \to 1]| \leq \frac{q_T^2 + q_F^2 + q_F q_T + q_{F'}(q_{\mathsf{Prove}} + q_{\mathsf{Rotate}}) + 2q_T^2(q_{\mathsf{Rotate}} + 1) + q_T^2}{p}.$$

But note that p is exponential in λ and any PPT adversary can make at most a polynomial number of queries, which implies q_{Prove} and q_{Rotate} are polynomial in λ. Similarly, we can bound $q_{F'}$ with $q_{\mathsf{Ideal}} + q_{\mathsf{Prove}} + q_{\mathsf{Rotate}}$, as TF' grows either as a result of direct random oracle queries for F', or due to the programming in Prove and Rotate. An analogous polynomial bound can be derived for q_T. Thus,

$$|\Pr[\mathsf{VRF\text{-}ZK\text{-}REAL}(\mathcal{A}) \to 1] - \Pr[\mathsf{VRF\text{-}ZK\text{-}IDEAL}(\mathcal{A}) \to 1]| \leq negl(\lambda).$$

QED.

5 RZKS-Construction

5.1 Relevant Primitives

In order to construct RZKS, we rely on a number of building blocks aside from Rotatable VRFs. Security definitions and constructions are included in Appendices C to E, but we include the syntax and a short description here for ease of reference.

Simulatable Commitments. A commitment is a scheme which allows a prover to publish a commitment to any given value such that the prover may later publish a proof that the commitment was indeed generated from the initial value. Furthermore, the simulatability requirement states that the commitment reveals no information about the committed value. A full definition and construction is included in Appendix E.

Definition 3 (Simulatable Commitments) A Simulatable Commitment Scheme C consists of 3 algorithms (C.Init, C.Commit, C.Verify) defined as follows:

▷ $pp \leftarrow \mathsf{C.GenPP}(1^\lambda)$: On input the security parameter, GenPP outputs public parameters pp.
▷ $\mathsf{com}, \mathsf{aux} \leftarrow \mathsf{C.Commit}(pp, m)$: Using the global parameters pp, the (randomized) commit algorithm produces commitment com to message m, and decommitment information aux.
▷ $1/0 \leftarrow \mathsf{C.Verify}(pp, \mathsf{com}, m, \mathsf{aux})$: This deterministic algorithm checks whether com is a valid commitment to message m, given the decommitment aux.

Ordered Accumulator (OA). An ordered accumulator is a scheme which allows a prover to commit to a sequence of label/value pairs. Furthermore, an ordered accumulator allows the prover to verifiably append label/value pairs to a previously committed sequence to generate a new commitment. The prover can later provide proofs that a given label/value pair is in the committed sequence or that a given label is not included in the committed sequence. A construction is given in Appendix C. Completeness and soundness are defined analogously to RZKS in Figures 1 and 2 respectively.

Definition 4 An Ordered Accumulator is a tuple of algorithms OA = (GenPP, Init, Update, VerifyUpd, Query, Verify, ProveAll, VerAll) defined as follows:

▷ pp, $pk_0 \leftarrow \mathcal{S}(\texttt{Init})$:
 – $TF, TF', T \leftarrow \{\}, \{\}, \{\}$
 – $i_{\mathsf{crpt}} \leftarrow -1, i_{\mathsf{cur}} \leftarrow 0$
 – $p \leftarrow$ prime exponential in λ
 – $S \leftarrow$ set of (at least) p strings. // Exponential size, but can have small description.
 – Sample $g \xleftarrow{\$} S$, add $(1, g)$ to T
 – $pk_0 \leftarrow RE(g)$
 – pp $\leftarrow (p, g, S)$ // We don't explicitly give out G, F, F' as they are replaced by oracles.
 – **return** pp, pk_0

▷ $\pi \leftarrow \mathcal{S}(\texttt{Explain}, i, \mathsf{label}, y)$:
 – $z \xleftarrow{\$} \mathbb{Z}_p$
 – $c \xleftarrow{\$} \mathbb{Z}_p$
 – Program $c = F'(g, F(x), pk_i, y, g^z \cdot pk_i^c, F(x)^z \cdot y^c)$
 – $\pi \leftarrow (g^z \cdot pk_i^c, F(x)^z \cdot y^c, z)$
 – **return** π

▷ $pk, \pi \leftarrow \mathcal{S}(\texttt{Rotate}, P)$:
 – $i_{\mathsf{cur}} \leftarrow i_{\mathsf{cur}} + 1$
 – $pk_{i_{\mathsf{cur}}} = RE(pk_{i_{\mathsf{cur}}-1})$
 – For each $(u, u') \in P$:
   • $a_u \leftarrow F'(u, pk_{i_{\mathsf{cur}}-1}, pk_{i_{\mathsf{cur}}}, P)$
 – $y \leftarrow \prod_{(u,u') \in P} u^{a_u}$
 – $y' \leftarrow \prod_{(u,u') \in P} u'^{a_u}$
 – $z \xleftarrow{\$} \mathbb{Z}_p$
 – $c \xleftarrow{\$} \mathbb{Z}_p$
 – Program $c = F'(pk_{i_{\mathsf{cur}}-1}, y, pk_{i_{\mathsf{cur}}}, y', pk_{i_{\mathsf{cur}}-1}^z \cdot pk_{i_{\mathsf{cur}}}^c, y^z \cdot (y')^c)$
 – $\pi \leftarrow (pk_{i_{\mathsf{cur}}-1}^z \cdot pk_{i_{\mathsf{cur}}}^c, y^z \cdot (y')^c, z)$
 – **return** $(pk_{i_{\mathsf{cur}}}, \pi)$

▷ $sk_{i_{\mathsf{crpt}}+1}, \ldots, sk_{i_{\mathsf{cur}}} \leftarrow \mathcal{S}(\texttt{Corrupt}, D)$:
 – For each $(i, x) \in D$:
   • $v_{i,x} \leftarrow D[i][x]$
   • $y_{i,x} \leftarrow$ index of $v_{i,x}$ in T. If not present, create a new formal variable and set it as the index. If not a single formal variable, fail.
   • $B_x \leftarrow T^{F(x)}$. // i.e. $q \leftarrow \mathcal{S}(F, x)$; $B_x \leftarrow T^q$.
   • $p_i \leftarrow T^{pk_i}$.
   • $p_{i,x} \leftarrow p_i \cdot B_x$
   • Replace all instances of $y_{i,x}$ in T with $p_{i,x}$ and simplify.
 – For $k = i_{\mathsf{crpt}} + 1, \ldots, i_{\mathsf{cur}}$:
   • $\alpha_k \xleftarrow{\$} \mathbb{Z}_p^*$.
   • $sk_k \xleftarrow{\$} \prod_{i=1}^k \alpha_i$.
   • $A_k \leftarrow$ the variable in the polynomial indexing pk_k in T.
   • Replace all instances of A_k in T with α_k, and simplify all polynomials. If this causes collisions, fail.
 – **return** $sk_0, \ldots, sk_{i_{\mathsf{cur}}}$, and set $i_{\mathsf{crpt}} \leftarrow i_{\mathsf{cur}}$.

▷ $y \leftarrow \mathcal{S}(\texttt{Corrupted-Eval}, i, x)$:
 – **return** $F(x)^{sk_i}$ // i.e., $\mathcal{S}(EXP, \mathcal{S}(F, x), sk_i)$

▷ $h \leftarrow \mathcal{S}(\star, g_1, g_2, b)$:
 – If $g_1 \notin T^{(\cdot)}$, create a new variable B_1 and add (B_1, g_1) to T.
 – If $g_2 \notin T^{(\cdot)}$, create a new variable B_2 and add (B_2, g_2) to T.
 – $P_1 \leftarrow T^{g_1}$.
 – $P_2 \leftarrow T^{g_2}$.
 – If $P_1 + b \cdot P_2 \in T_{(\cdot)}$, **return** $T_{P_1 + b \cdot P_2}$.
 – $h \xleftarrow{\$} S \setminus T^{(\cdot)}$.
 – Add $(P_1 + b \cdot P_2, h)$ to T.
 – **return** h.

▷ $h' \leftarrow \mathcal{S}(EXP, h, r)$:
 – If $h \notin T^{(\cdot)}$, create a new variable B and add (B, h) to T.
 – $P \leftarrow T^h$.
 – If $r \cdot P \in T_{(\cdot)}$, **return** $T_{r \cdot P}$.
 – $h' \xleftarrow{\$} S \setminus T^{(\cdot)}$.
 – Add $(r \cdot P, h')$ to T.
 – **return** h'.

▷ $y \leftarrow \mathcal{S}(F, x)$:
 – If $x \in TF$, **return** TF_x
 – $y \xleftarrow{\$} S \setminus (T^{(\cdot)} \cup TF^{(\cdot)})$
 – Add (x, y) to TF
 – Add (B_x, y) to T, where B_x is a new variable.
 – **return** y

▷ $y \leftarrow \mathcal{S}(F', x)$:
 – If $x \in TF'$, **return** TF'_x
 – $y \xleftarrow{\$} \mathbb{Z}_p$
 – Add (x, y) to TF'
 – **return** y

▷ $h' \leftarrow RE(h)$: // This is just a subroutine, not exposed as an oracle
 – If $h \notin T$, then create a new variable B_h and store (B_h, h) in T.
 – $p_h \leftarrow T^h$.
 – Sample $h' \xleftarrow{\$} S \setminus T^{(\cdot)}$.
 – Create a new variable A, and store $(p_h \cdot A, h')$ in T.
 – **return** h'

Fig. 7: The full VRF simulator for Theorem 2. Note that $\mathcal{S}$ has to implement two random oracles F, F' and two different oracles $\star, EXP$ for the generic group when answering $\texttt{Ideal}$ queries: for simplicity, we write $\mathcal{S}(F, x)$ instead of $\mathcal{S}(\texttt{Ideal}, (F, x))$ and analogously for F', EXP and $\star$. Moreover, we abuse notation and we write $a \cdot b$ (with $a, b \in S$) instead of $\mathcal{S}(\star, a, b, 1)$ and a^z (with $z \in Z_p$) to mean $\mathcal{S}(EXP, a, x)$. The state of the simulator consists of tables TF, TF', T which are used to implement the 3 ideal objects, as well as vectors of all the α_i, sk_i and pk_i values, and counters i_{crpt} and i_{cur} which track those of the experiment.

▷ GenPP, Init, Update, VerifyUpd, Query, Verify are defined analogously to the RZKS in Definition 1.

▷ $\pi \leftarrow$ OA.ProveAll(pp, st, u): This algorithm outputs π which can be verified against the commitment com_u output by the u-th call to Update. It proves the set of label value pairs included in the datastore up to epoch u.

▷ $1/0 \leftarrow$ OA.VerAll(pp, com_u, P, π): This deterministic algorithm takes a digest com_u, a set P of (label, val, t) pairs, and a proof. It checks that P is the set of all pairs that com_u commits to.

Append-Only Vector Commitments (AVC). An append-only vector commitment can be used to commit to a list of values, extend the list without recomputing the commitment from scratch, prove what the value is at a specific position in the list, and prove that two commitments have been obtained by extending the same list.

We briefly discuss the syntax of this primitive here, and defer the security definitions and construction to Appendix D.1.

Definition 5 An Append-only Vector Commitment is a tuple of algorithms AVC = (GenPP, Init, Update, ProveExt, VerExt, Query, Verify) defined as follows:

▷ pp $\leftarrow$ AVC.GenPP(1^λ): This algorithm takes the security parameter and produces public parameter pp for the scheme. All other algorithms take these pp as input, even when not explicitly specified.

▷ (com, st) $\leftarrow$ AVC.Init(pp): This algorithm produces an initial commitment com to an empty list $\mathbf{D}_0 = \{\}$, and an initial server/prover state st. Each server state st will contain a list and a digest, which we will refer to as $\mathbf{D}(\text{st})$ and $\mathbf{com}(\text{st})$. Similarly, each commitment will include an integer $\mathbf{t}(\text{com})$ (also called an epoch for consistency with other primitives) representing the size of the list it commits to. (Alternatively, these can be thought of as deterministic functions which are part of the scheme.)

▷ (com$'$, st$'$, π_S) $\leftarrow$ AVC.Update(pp, st, val): This algorithm takes in the current state of the prover st, and a value val. The algorithm outputs an updated commitment to the datastore, an updated internal state st$'$, and proof π (to be verified with VerExt) that the update has been done correctly. Intuitively, com$'$ is a commitment to the list $\mathbf{D}(\text{st}') = \mathbf{D}(\text{st})\|\text{val}$ of size $\mathbf{t}(\text{com}') = \mathbf{t}(\mathbf{com}(\text{st})) + 1$.

▷ $\pi \leftarrow$ AVC.ProveExt(pp, st, t', t): Given the prover's state st and two integers, the algorithm produces a proof that the list committed to by com_t (output at the t-th invocation of Update) extends the one committed to by $\text{com}_{t'}$.

▷ $0/1 \leftarrow$ AVC.VerExt(pp, com$'$, com, π): This deterministic algorithm takes in two digests and proves that the list committed to by com extends the one committed to by com$'$. The proofs can be produced by either Update or ProveExt.

▷ (π, val) $\leftarrow$ AVC.Query(pp, st, u, t'): This algorithm takes as input a state st and epochs u and t' such that $u \leq t' \leq \mathbf{t}(\text{st})$. It returns val $= \mathbf{D}(\text{st})[u]$ and a membership proof π to be verified against the commitment $\text{com}_{t'}$ output by Update during the t'-th update.

▷ $0/1 \leftarrow$ AVC.Verify(pp, com, u, val, π): This deterministic algorithm checks the proof π (produced by Query) that val is the is the u-th element of the list committed by com.

5.2 RZKS Construction

We describe our RZKS construction in Figure 8. The RZKS commits to a set of (label, val) pairs by storing in an ordered accumulator (tlbl, tval) pairs, where a given tlbl is the VRF output[18] for a given label, and a given tval is the commitment to a given val. Elements are added to the OA in batches, where the i-th update to the OA produces a digest at the i-th epoch. At each epoch, the OA digest and VRF public key are stored in the corresponding index of the AVC. The resulting AVC digest is returned as the RZKS digest.

Updating the RZKS produces an append-only proof, which contains the append-only proofs for the underlying OA and AVC. To verify the presence of a (label, val), inclusion/exclusion proofs include the VRF proof, commitment opening, OA digest, a proof that the label/value pair is consistent with that digest, and a proof that the digest is at the expected index of the vector that the AVC digest commits to.

The AVC data structure allows the RZKS to support the ProveExt and VerExt algorithms, in which the server proves that a recent RZKS digest commits to an older one that the verifier currently holds

---

[18] The Rotatable VRF presented in this work outputs group elements, while the ordered accumulator takes as input bit-strings, so we implicitly assume that these group elements have a unique bit-string representation.

(therefore, the client can forget the old digest without losing the ability to hold the server accountable later).

The RZKS construction is similar to the append-only ZKS described in SEEMless [8], but i) each leaf also contains the epoch number at which such leaf was inserted, and ii) it uses a rotatable VRF instead of a standard one. To perform a rotation, the prover rotates the VRF key and builds a brand new ordered accumulator using the same commitments as the old one, but uses the new VRF outputs as labels. The audit proof for such a rotation involves the VRF rotation proof for all the pre-existing labels, plus an append-only proof for any new labels that were added.

Finally, we summarize the state that the RZKS maintains (note that some values in the state are redundant for the sake of readability). It maintains D, a map of all the (label, val) pairs in the RZKS, and epno, the latest epoch number. It also contains st_{OA} and st_{AVC}, the underlying state of the OA and AVC, respectively. It stores com_{epno}, which is the latest value stored at the epno-th position in the AVC (recall that it contains the latest OA digest and VRF public key). And, the RZKS state stores K_{VRF}, a map of the VRF keypair for each VRF keypair generation; G, a map of the corresponding VRF generation for each epoch number; and g, the latest VRF keypair generation number.

5.3 RZKS Protocol Security

Theorem 3 *The scheme described in Figure 8 satisfies completeness according to Definition 1.*

This is easy to see by inspection.

Theorem 4 *Let* OA *be an Ordered Accumulator,* C *be a Commitment scheme,* VRF *be a VRF, and* AVC *be an Append-only Vector Commitment, all satisfying their respective definitions of soundness w.r.t. their own idealized objects. Then the* RZKS *construction of Figure 8 satisfies soundness, w.r.t. the set of all such idealized objects.*

Proof Sketch: To prove soundness, we define an RZKS extractor that trivially combines those for the underlying building blocks. It extracts a dictionary from an RZKS digest by feeding the output of each extractor as input to the next, and answers Ideal oracle queries for a primitive's ideal object by running the appropriate extractor. Given this extractor, we make a hybrid argument: we first need to add extra assertions to the ideal RZKS game enforcing that the individual components of an RZKS proof match the output of the corresponding extractors (indistinguishability can be proven based on the soundness of those primitives). This prevents an adversary from submitting proofs for the same tuples that the combined extractor outputs, but that disagree with the internal extractors. After that, we can start removing the individual extractors and honestly implementing the corresponding ideal objects (relying a second time on the same soundness properties of the underlying primitives) to get to the real game. The full proof is in Appendix F.

Last, we prove that the RZKS construction satisfies zero-knowledge with leakage. The leakage function provides the simulator, for Update queries, with the number of elements that are being added to the data structure, as well as the labels (but not values) of any added pair that the adversary has queried since the last PCSUpdate (and was given absence proofs for). PCSUpdate queries only include the number of added pairs. When the adversary calls the Query oracle, the simulator is given the queried label, as well as the epoch it was added at (if the label is in the RZKS) and value (if it was added no later than the queried epoch). On LeakState queries, the simulator is given the full contents of the data structure, and subsequent Update queries until the next PCSUpdate also reveal all the added labels (but not the values). Finally, ProveExt queries just reveal the queried epochs. A formal definition follows:

Leakage L.

- The shared state consists of a set of labels X, a datastore D, a counter t for the current epoch (initialized to 0), a counter g for the current generation (i.e. the number of PCSUpdate operations performed, also starting at 0), a map G that matches each epoch to the respective generation, and a boolean $leaked$ (initially false).
- $L_{\textbf{Query}}$(label, u): If $\exists$(label, val, t') $\in$ D such that $t' \leq u$, the function returns (label, val, t', u). If $\exists$(label, val, t') $\in$ D such that $G[t'] = G[u]$, the function returns (label, $\bot$, t', u). Otherwise, it returns (label, $\bot$, $\bot$, u) and, if $G[u] = g$, adds label to X.

▷ $\mathsf{pp} \leftarrow \mathsf{RZKS.GenPP}(1^\lambda)$:
- $\mathsf{pp}_{\mathsf{VRF}} \leftarrow \mathsf{VRF.GenPP}(1^\lambda)$
- $\mathsf{pp}_{\mathsf{OA}} \leftarrow \mathsf{OA.GenPP}(1^\lambda)$
- $\mathsf{pp}_{\mathsf{C}} \leftarrow \mathsf{C.GenPP}(1^\lambda)$
- $\mathsf{pp}_{\mathsf{AVC}} \leftarrow \mathsf{AVC.GenPP}(1^\lambda)$
- **return** $\mathsf{pp} \leftarrow (\mathsf{pp}_{\mathsf{VRF}}, \mathsf{pp}_{\mathsf{OA}}, \mathsf{pp}_{\mathsf{C}}, \mathsf{pp}_{\mathsf{AVC}})$

▷ $(\mathsf{com}, \mathsf{st}) \leftarrow \mathsf{RZKS.Init}(\mathsf{pp})$:
- **parse** pp as $(\mathsf{pp}_{\mathsf{VRF}}, \mathsf{pp}_{\mathsf{OA}}, \mathsf{pp}_{\mathsf{C}}, \mathsf{pp}_{\mathsf{AVC}})$
- $\mathsf{epno} \leftarrow 0, \mathsf{g} \leftarrow 0, \mathsf{K}_{\mathsf{VRF}} \leftarrow \{\}, \mathsf{D} \leftarrow \{\}, \mathsf{st}_{\mathsf{OA}} \leftarrow \{\}, \mathsf{G} \leftarrow \{\}$
- $sk_0, pk_0 \leftarrow \mathsf{VRF.KeyGen}(\mathsf{pp}_{\mathsf{VRF}})$; $\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}] \leftarrow (sk_0, pk_0), \mathsf{G}[\mathsf{epno}] \leftarrow \mathsf{g}$
- $(\mathsf{st}', \mathsf{com}_{\mathsf{OA}}^0) \leftarrow \mathsf{OA.Init}(\mathsf{pp}_{\mathsf{OA}})$; $\mathsf{com}_{\mathsf{INT}}^0 \leftarrow (\mathsf{com}_{\mathsf{OA}}^0, pk_0); \mathsf{st}_{\mathsf{OA}}[\mathsf{g}] \leftarrow \mathsf{st}'$
- $(\mathsf{st}_{\mathsf{AVC}}, \_) \leftarrow \mathsf{AVC.Init}(\mathsf{pp}_{\mathsf{AVC}})$; $\mathsf{com}^1, \mathsf{st}_{\mathsf{AVC}}, \pi^0 \leftarrow \mathsf{AVC.Update}(\mathsf{st}_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}}^0)$
- $\mathsf{st} \leftarrow (\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}^1, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- **return** $\mathsf{com}^1, \mathsf{st}$

▷ $(\mathsf{com}, \mathsf{st}', \pi) \leftarrow \mathsf{RZKS.Update}(\mathsf{st}, S_{\mathsf{update}})$:
  $(\mathsf{com}, \mathsf{st}', \pi) \leftarrow \mathsf{RZKS.PCSUpdate}(\mathsf{st}, S_{\mathsf{update}})$:
  // bullets with □ only apply to PCSUpdate
- **parse** st as $(\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$; set $\mathsf{epno} \leftarrow \mathsf{epno} + 1$
- **parse** S_{update} as $(\mathsf{label}_1, \mathsf{val}_1), \ldots, (\mathsf{label}_n, \mathsf{val}_n)$
- **ensure** $\mathsf{label}_1, \ldots, \mathsf{label}_n$ are distinct and $\notin \mathsf{D}$
□ $L \leftarrow \{\mathsf{label} \mid (\mathsf{label}, (\ldots)) \in \mathsf{D}\}$
□ $sk_{\mathsf{g}+1}, pk_{\mathsf{g}+1}, \pi_{\mathsf{VRF}} \leftarrow \mathsf{VRF.Rotate}(\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}], L)$
□ $\mathsf{K}_{\mathsf{VRF}}[\mathsf{g} + 1] \leftarrow (sk_{\mathsf{g}+1}, pk_{\mathsf{g}+1})$
□ $\mathsf{g} \leftarrow \mathsf{g} + 1$
□ For $\mathsf{g}' \in \{\mathsf{g}, \mathsf{g} - 1\}$:
  • $\{\mathsf{tlbl}_j^{\mathsf{g}'}\}_{j \in L} \leftarrow \{\mathsf{VRF.Eval}(\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}'].sk, j)\}_{j \in L}$
□ $\pi_{\mathsf{OA}}^{\mathsf{g}-1} \leftarrow \mathsf{OA.ProveAll}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g} - 1], \mathsf{epno} - 1)$
□ $\mathsf{st}', \_ \leftarrow \mathsf{OA.Init}(\mathsf{pp}_{\mathsf{OA}})$; For $i \in [\mathsf{epno} - 1]$:
  • $S_{\mathsf{OA}} \leftarrow \{(\mathsf{tlbl}_j^{\mathsf{g}}, \mathsf{tval}_j) \mid (j, (\cdot, i, \mathsf{tval}_j, \cdot)) \in \mathsf{D}\}$
  • $\mathsf{st}', \mathsf{com}'_{\mathsf{OA}}, \_ \leftarrow \mathsf{OA.Update}(\mathsf{st}', S_{\mathsf{OA}})$
□ $\mathsf{st}_{\mathsf{OA}}[\mathsf{g}] \leftarrow \mathsf{st}', \mathsf{com}_{\mathsf{OA}}^{\mathsf{epno}-1} \leftarrow \mathsf{com}'_{\mathsf{OA}}$
□ $\pi_{\mathsf{OA}}^{\mathsf{g}} \leftarrow \mathsf{OA.ProveAll}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], \mathsf{epno} - 1)$
- $S_{\mathsf{OA}} \leftarrow \{\}$; For each $(\mathsf{label}_i, \mathsf{val}_i) \in S_{\mathsf{update}}$:
  • $\mathsf{tlbl}_i \leftarrow \mathsf{VRF.Eval}(\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}].sk, \mathsf{label}_i)$
  • $\mathsf{tval}_i, \mathsf{aux}_i \leftarrow \mathsf{C.Commit}(\mathsf{val}_i)$
  • $S_{\mathsf{OA}} \leftarrow S_{\mathsf{OA}} \cup \{(\mathsf{tlbl}_i, \mathsf{tval}_i)\}$
  • $\mathsf{D} \leftarrow \mathsf{D} \cup \{(\mathsf{label}_i, (\mathsf{val}, \mathsf{epno}, \mathsf{tval}_i, \mathsf{aux}_i))\}$
- $\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], \mathsf{com}_{\mathsf{OA}}^{\mathsf{epno}}, \pi_{\mathsf{OA}} \leftarrow \mathsf{OA.Update}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], S_{\mathsf{OA}})$; $\mathsf{com}_{\mathsf{INT}}^{\mathsf{epno}} \leftarrow (\mathsf{com}_{\mathsf{OA}}^{\mathsf{epno}}, \mathsf{K}_{\mathsf{VRF}}[\mathsf{g}].pk)$; $\mathsf{G}[\mathsf{epno}] \leftarrow \mathsf{g}; \pi' \leftarrow \pi_{\mathsf{OA}}$
- $\mathsf{com}, \mathsf{st}_{\mathsf{AVC}}, \pi_{\mathsf{AVC}} \leftarrow \mathsf{AVC.Update}(\mathsf{st}_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}}^{\mathsf{epno}})$
- $\_, \pi_{\mathsf{AVC}}^{\mathsf{epno}} \leftarrow \mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, \mathbf{t}(\mathsf{com}), \mathbf{t}(\mathsf{com}))$
- $\mathsf{com}_{\mathsf{INT}}^{\mathsf{epno}-1}, \pi_{\mathsf{AVC}}^{\mathsf{epno}-1} \leftarrow$ $\mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, \mathbf{t}(\mathsf{com}) - 1, \mathbf{t}(\mathsf{com}) - 1)$
□ $\pi' \leftarrow (\pi_{\mathsf{OA}}, \pi_{\mathsf{OA}}^{\mathsf{g}-1}, \pi_{\mathsf{OA}}^{\mathsf{g}}, \pi_{\mathsf{VRF}}, \mathsf{com}_{\mathsf{OA}}^{\mathsf{epno}-1}, \{(\mathsf{tlbl}_j^{g-1}, \mathsf{tlbl}_j^{g}, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L})$
- $\pi \leftarrow (\pi', \pi_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}}^{\mathsf{epno}}, \mathsf{com}_{\mathsf{INT}}^{\mathsf{epno}-1}, \pi_{\mathsf{AVC}}^{\mathsf{epno}}, \pi_{\mathsf{AVC}}^{\mathsf{epno}-1})$
- $\mathsf{st} \leftarrow (\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- **return** $(\mathsf{com}, \mathsf{st}, \pi)$

▷ $0/1 \leftarrow \mathsf{RZKS.VerifyUpd}(\mathsf{com}^{t_0}, \mathsf{com}^{t_1}, \pi)$:
- **parse** π as $(\pi', \pi_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}}^{t_1}, \mathsf{com}_{\mathsf{INT}}^{t_0}, \pi_{\mathsf{AVC}}^{t_1}, \pi_{\mathsf{AVC}}^{t_0})$
- **parse** $\mathsf{com}_{\mathsf{INT}}^{t_0}$ as $(\mathsf{com}_{\mathsf{OA}}^{t_0}, pk_{t_0})$
- **parse** $\mathsf{com}_{\mathsf{INT}}^{t_1}$ as $(\mathsf{com}_{\mathsf{OA}}^{t_1}, pk_{t_1})$
- **ensure** $\mathsf{OA.t}(\mathsf{com}_{\mathsf{OA}}^{t_0}) + 2 = \mathsf{AVC.t}(\mathsf{com}^{t_0}) + 1 = \mathsf{AVC.t}(\mathsf{com}^{t_1}) = \mathsf{OA.t}(\mathsf{com}_{\mathsf{OA}}^{t_1}) + 1$
- **ensure** $\mathsf{AVC.VerExt}(\mathsf{com}^{t_0}, \mathsf{com}^{t_1}, \pi_{\mathsf{AVC}}) = 1$
- For $t \in \{t_0, t_1\}$:
  • **ensure** $\mathsf{AVC.Verify}(\mathsf{com}^t, \mathsf{AVC.t}(\mathsf{com}^t), \mathsf{com}_{\mathsf{INT}}^t, \pi_{\mathsf{AVC}}^t) = 1$
- If $pk_{t_0} = pk_{t_1}$:
  • **parse** π' as π_{OA}; set $\mathsf{com}'_{\mathsf{OA}} \leftarrow \mathsf{com}_{\mathsf{OA}}^{t_0}$
  Else:
  • **parse** π' as $(\pi_{\mathsf{OA}}, \pi_{\mathsf{OA}}^{\mathsf{g}-1}, \pi_{\mathsf{OA}}^{\mathsf{g}}, \pi_{\mathsf{VRF}}, \mathsf{com}'_{\mathsf{OA}}, \{(\mathsf{tlbl}_j^{\mathsf{g}-1}, \mathsf{tlbl}_j^{\mathsf{g}}, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L})$
  • **ensure** $\mathsf{VRF.VerRotate}(pk_{t_0}, pk_{t_1}, \{(\mathsf{tlbl}_j^{\mathsf{g}-1}, \mathsf{tlbl}_j^{\mathsf{g}})\}_{j \in L}, \pi_{\mathsf{VRF}}) = 1$
  • **ensure** $\mathsf{OA.VerAll}(\mathsf{com}_{\mathsf{OA}}^{t_0}, \{(\mathsf{tlbl}_j^{\mathsf{g}-1}, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L}, \pi_{\mathsf{OA}}^{\mathsf{g}-1}) = 1$
  • **ensure** $\mathsf{OA.VerAll}(\mathsf{com}'_{\mathsf{OA}}, \{(\mathsf{tlbl}_j^{\mathsf{g}}, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L}, \pi_{\mathsf{OA}}^{\mathsf{g}}) = 1$
- **ensure** $\mathsf{OA.VerifyUpd}(\mathsf{com}'_{\mathsf{OA}}, \mathsf{com}_{\mathsf{OA}}^{t_1}, \pi_{\mathsf{OA}}) = 1$
- **return** 1

▷ $(\pi, \mathsf{val}, t) \leftarrow \mathsf{RZKS.Query}(\mathsf{st}, u, \mathsf{label})$:
- **parse** st as $(\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- **ensure** $u \leq \mathsf{epno}$
- $(\mathsf{tlbl}, \pi_{\mathsf{VRF}}) \leftarrow \mathsf{VRF.Query}(\mathsf{K}_{\mathsf{VRF}}[\mathsf{G}[u]].sk, \mathsf{label})$
- If $\mathsf{label} \in \mathsf{D}$ and $\mathsf{D}[\mathsf{label}].\mathsf{epno}_{\mathsf{label}} \leq u$:
  • $(\mathsf{val}, \mathsf{epno}_{\mathsf{label}}, \mathsf{tval}, \mathsf{aux}) \leftarrow \mathsf{D}[\mathsf{label}]$
  Else:
  • $(\mathsf{val}, \mathsf{epno}_{\mathsf{label}}, \mathsf{tval}, \mathsf{aux}) \leftarrow (\bot, \bot, \bot, \bot)$
- $\pi_{\mathsf{OA}}, \_ \leftarrow \mathsf{OA.Query}(\mathsf{st}_{\mathsf{OA}}[\mathsf{G}[u]], u, \mathsf{tlbl})$
- $\pi_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}} \leftarrow \mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, u, u)$
- $\pi \leftarrow (\pi_{\mathsf{AVC}}, \pi_{\mathsf{OA}}, \pi_{\mathsf{VRF}}, \mathsf{tlbl}, \mathsf{tval}, \mathsf{aux}, \mathsf{com}_{\mathsf{INT}})$
- **return** $\pi, \mathsf{val}, \mathsf{epno}_{\mathsf{label}}$

▷ $0/1 \leftarrow \mathsf{RZKS.Verify}(\mathsf{com}, \mathsf{label}, \mathsf{val}, t, \pi)$:
- **parse** π as $(\pi_{\mathsf{AVC}}, \pi_{\mathsf{OA}}, \pi_{\mathsf{VRF}}, \mathsf{tlbl}, \mathsf{tval}, \mathsf{aux}, \mathsf{com}_{\mathsf{INT}})$
- **parse** $\mathsf{com}_{\mathsf{INT}}$ as $(\mathsf{com}_{\mathsf{OA}}, pk)$
- **ensure** $\mathsf{VRF.Verify}(pk, \mathsf{label}, \mathsf{tlbl}, \pi_{\mathsf{VRF}}) = 1$
- **ensure** $\mathsf{AVC.t}(\mathsf{com}) = \mathsf{OA.t}(\mathsf{com}_{\mathsf{OA}}) + 1$
- If $t = \bot \vee \mathsf{val} = \bot \vee \mathsf{tval} = \bot$
  Then **ensure** $\mathsf{val} = \mathsf{tval} = t = \bot$
  Else **ensure** $\mathsf{C.Verify}(\mathsf{val}, \mathsf{tval}, \mathsf{aux}) = 1$
- **ensure** $\mathsf{OA.Verify}(\mathsf{com}_{\mathsf{OA}}, \mathsf{tlbl}, \mathsf{tval}, t, \pi_{\mathsf{OA}}) = 1$
- **ensure** $\mathsf{AVC.Verify}(\mathsf{com}, \mathsf{AVC.t}(\mathsf{com}), \mathsf{com}_{\mathsf{INT}}, \pi_{\mathsf{AVC}}) = 1$
- **return** 1

▷ $(\pi, \mathsf{val}, t) \leftarrow \mathsf{RZKS.ProveExt}(\mathsf{st}, t_0, t_1)$:
- **parse** st as $(\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- **return** $\mathsf{AVC.ProveExt}(\mathsf{st}_{\mathsf{AVC}}, t_0, t_1)$

▷ $0/1 \leftarrow \mathsf{RZKS.VerExt}(\mathsf{com}^{t_0}, \mathsf{com}^{t_1}, \pi)$:
- **return** $\mathsf{AVC.VerExt}(\mathsf{com}^{t_0}, \mathsf{com}^{t_1}, \pi)$

Fig. 8: Our RZKS construction. We implicitly assume that the public parameters output by GenPP are input to all other algorithms, parsed into their components and input to the VRF, OA and C, AVC algorithms as appropriate (as shown in Init). Moreover, since the OA commitment to the empty datastore ends up as the first element of the AVC, in this construction we define $\mathsf{RZKS.t}(\mathsf{com})$ as $\mathsf{AVC.t}(\mathsf{com}) - 1$.

- $L_{\mathsf{Update}}(S)$: Parse $S = \{(\mathsf{label}_i, \mathsf{val}_i)\}$. If S contains any duplicate label, or any label which appears in D, this function returns $\perp$. Else, it increments t, sets $G[t] \leftarrow g$, and adds the pairs from S to the datastore D at epoch t. If *leaked*, it returns the labels in S. Else, it returns $|S|$ and the set of labels from S which are also in X.
- $L_{\mathsf{PCSUpdate}}(S)$: Parse $S = \{(\mathsf{label}_i, \mathsf{val}_i)\}$. If S contains any duplicate label, or any label which appears in D, this function returns $\perp$. Else, it increments t, adds the pairs from S to the datastore D at epoch t, and updates $X \leftarrow \{\}$, *leaked* $\leftarrow$ *false*, and $g \leftarrow g+1$, $G[t] \leftarrow g$. It returns $|S|$.
- $L_{\mathsf{LeakState}}(S)$: Set *leaked* $\leftarrow$ *true*. **return** D.
- $L_{\mathsf{ProveExt}}(t_0, t_1)$: **return** (t_0, t_1).

Theorem 5 *Let* VRF *be a rotatable VRF in some idealized model,* C *be a simulatable commitments scheme in some idealized model, and* AVC *be any Append-only Vector Commitment. Then, our $\mathcal{Z}$ construction satisfies zero-knowledge with leakage L as above in the idealized models used by the underlying protocols.*

Proof Sketch: The proof is structured as a hybrid argument. Starting from the real game, one can first substitute Commitments and (Rotatable) VRF outputs and proofs with random strings or those produced by the respective simulators, and then notice that, at this point, the information provided by the leakage function L is sufficient to produce these simulated values without relying on the full input to the oracle calls. For example, when an Update oracle query happens (for a non compromised key), the simulator receives the number of pairs that the adversary wants to add to the RZKS, and can itself generate enough random strings (to use as VRF outputs) and simulated commitments to add to the OA, and then adds the new OA commitment to the AVC. Upon corruption or queries, the simulator learns the actual values corresponding to these queries, and can simulate commitment openings and VRF proofs accordingly, and provide honestly generated OA and AVC proofs. A full proof is deferred to Appendix G.

5.4 Instantiation and Complexity

If we allow each building block to be instantiated as constructed in the appendices, we can define an instantiation for the entire scheme. We then calculate the efficiency of each building block, which then gives us the efficiency of the entire scheme.

We calculate an upper bound on the number of hash computations and group exponentiations under the constructions of our building blocks and RZKS as follows: we define n to be the size of the stored datastore, and s to be the size of the update query (when relevant). Let ℓ be the number of bits needed to represent a group element. We assume without loss of generality that ℓ is also the number of bits needed to represent a group exponent. Let ℓ' be the number of bits to represent a label. Note that when some algorithm ignores the proof output from another, we skip the proof calculation. The full list of complexities is displayed in Figure 9.

Acknowledgements

References

1. Masayuki Abe, Jens Groth, Kristiyan Haralambiev, and Miyako Ohkubo. Optimal structure-preserving signatures in asymmetric bilinear groups. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 649–666. Springer, Heidelberg, August 2011.
2. apple.com. Apple privacy. https://www.apple.com/privacy/features. Accessed: 2022-08-03.
3. Hala Assal, Stephanie Hurtado, Ahsan Imran, and Sonia Chiasson. What's the deal with privacy apps? a comprehensive exploration of user perception and usability. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, MUM '15, page 25–36, New York, NY, USA, 2015. Association for Computing Machinery.
4. Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73. ACM Press, November 1993.

	Hashes	Exponentiations	Proof length
VRF.Query	2	4	3ℓ
VRF.Verify	2	4	-
VRF.Rotate	$3s+1$	$4s+3$	3ℓ
VRF.VerRotate	$s+1$	$2s+4$	-
C.Commit	1	-	2λ
C.Verify	1	-	-
OA.Update	$2(s+\lceil \log n \rceil)$	-	$s\ell' + 2(s+1)\lceil \log(n+s) \rceil \lambda$
OA.VerifyUpd	$4(s+1)\lceil \log(n+s) \rceil + 2s$	-	-
OA.Query	$2n-1$	-	$4\lceil \log(n+1) \rceil \lambda$
OA.Verify	$4\lceil \log(n+1) \rceil + 2$	-	-
OA.ProveAll	-	-	-
OA.VerAll	$2n$	-	-
AVC.Update	$2(1+\lceil \log n \rceil)$	-	$4\lambda \lceil \log n \rceil$
AVC.ProveExt	$2\lceil \log n \rceil$	-	$4\lambda \lceil \log n \rceil$
AVC.VerExt	$6\lceil \log n \rceil$	-	-
AVC.Query	$2\lceil \log n \rceil$	-	$4\lambda \lceil \log n \rceil$
AVC.Verify	$4\lceil \log n \rceil + 1$	-	-
RZKS.Update	$4s+6\lceil \log n \rceil + 2$	s	$s\ell' + (2(s+1)\lceil \log(n+s) \rceil + 12\lceil \log n \rceil)\lambda$
RZKS.PCSUpdate	$6s+8\lceil \log n \rceil + 3n + 3$	$s+4n+3$	$3\ell + s\ell' + (2(s+1)\lceil \log(n+s) \rceil + 12\lceil \log n \rceil)\lambda$
RZKS.VerifyUpd	$6\lceil \log(n+s) \rceil + 8\lceil \log n \rceil + 4n + 2s + 3$	$2s+4$	-
RZKS.Query	$2\lceil \log n \rceil + 2n + 1$	4	$3\ell + 8\lambda \lceil \log(n+1) \rceil$
RZKS.Verify	$8\lceil \log(n+1) \rceil + 6$	4	-
RZKS.ProveExt	$2\lceil \log n \rceil$	-	-
RZKS.VerExt	$6\lceil \log n \rceil$	-	-

Fig. 9: The complexity of our various constructions.

5. John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from PGV. Cryptology ePrint Archive, Report 2002/066, 2002. https://eprint.iacr.org/2002/066.

6. John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography, 9th Annual International Workshop, SAC 2002, St. John's, Newfoundland, Canada, August 15-16, 2002. Revised Papers*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2002.

7. Josh Blum, Simon Booth, Brian Chen, Oded Gal, Maxwell Krohn, Julia Len, Karan Lyons, Antonio Marcedone, Mike Maxim, Merry Ember Mou, Jack O'Connor, Surya Rien, Miles Steele, Matthew Green, Lea Kissner, and Alex Stamos. E2e encryption for zoom meetings. White paper, 2021. https://github.com/zoom/zoom-e2e-whitepaper/blob/master/zoom_e2e.pdf.

8. Melissa Chase, Apoorvaa Deshpande, Esha Ghosh, and Harjasleen Malvai. SEEMless: Secure end-to-end encrypted messaging with less trust. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1639–1656. ACM Press, November 2019.

9. Melissa Chase, Alexander Healy, Anna Lysyanskaya, Tal Malkin, and Leonid Reyzin. Mercurial commitments with applications to zero-knowledge sets. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 422–439. Springer, 2005.

10. Melissa Chase and Anna Lysyanskaya. Simulatable VRFs with applications to multi-theorem NIZK. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 303–322. Springer, Heidelberg, August 2007.

11. Melissa Chase and Sarah Meiklejohn. Transparency overlays and applications. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 168–179. ACM Press, October 2016.

12. Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In *ACM CCS 20*, pages 1445–1459. ACM Press, 2020.

13. David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 89–105. Springer, Heidelberg, August 1993.

14. Scott A Crosby and Dan S Wallach. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium*, pages 317–334, 2009.

15. Novi Financial. Auditable key directory. https://github.com/novifinancial/akd/, 2021. Accessed: 2022-05-26.

16. Oliver Gasser, Benjamin Hof, Max Helm, Maciej Korczynski, Ralph Holz, and Georg Carle. In log we trust: Revealing poor security practices with certificate transparency logs and internet measurements. In *International Conference on Passive and Active Network Measurement*, pages 173–185. Springer, 2018.

17. Ashrujit Ghoshal and Stefano Tessaro. Tight state-restoration soundness in the algebraic group model. LNCS, pages 64–93. Springer, Heidelberg, 2021.

18. Sharon Goldberg, Leonid Reyzin, Dimitrios Papadopoulos, and Jan Včelák. Verifiable Random Functions (VRFs). Internet-Draft draft-irtf-cfrg-vrf-12, Internet Engineering Task Force, May 2022. Work in Progress.

19. Google. Key transparency overview. https://github.com/google/keytransparency/blob/master/docs/overview.md. Accessed: 2022-08-31.

20. Amir Herzberg and Hemi Leibowitz. Can johnny finally encrypt? evaluating e2e-encryption in popular im applications. In *Proceedings of the 6th Workshop on Socio-Technical Aspects in Security and Trust*, STAST '16, page 17–28, New York, NY, USA, 2016. Association for Computing Machinery.

21. Amir Herzberg, Hemi Leibowitz, Kent Seamons, Elham Vaziripour, Justin Wu, and Daniel Zappala. Secure messaging authentication ceremonies are broken. *IEEE Security Privacy*, 19(2):29–37, 2021.

22. Yuncong Hu, Kian Hooshmand, Harika Kalidhindi, Seung Jin Yang, and Raluca A. Popa. Merkle2: A low-latency transparency log system. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 285–303, 2021.

23. Keybase.io. Keybase chat. https://book.keybase.io/docs/chat. Accessed: 2022-08-03.

24. keybase.io. Keybase is now writing to the stellar blockchain. https://book.keybase.io/docs/server/stellar. Accessed: 2022-07-29.

25. Keybase.io. Meet your sigchain (and everyone else's). https://book.keybase.io/docs/server#meet-your-sigchain-and-everyone-elses. Accessed: 2022-07-29.

26. keybase.io. Keybase first commitment. https://keybase.io/_/api/1.0/merkle/root.json?seqno=1, 2014. Accessed: 2022-05-26.

27. Keybase.io. Keybase is not softer than tofu. https://keybase.io/blog/chat-apps-softer-than-tofu, 2019. Accessed: 2019-05-05.

28. Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. RFC 9162, December 2021.

29. Ada Lerner, Eric Zeng, and Franziska Roesner. Confidante: Usable encrypted email: A case study with lawyers and journalists. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 385–400. IEEE, 2017.

30. Ueli M. Maurer. Abstract models of computation in cryptography (invited paper). In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding*, volume 3796 of *LNCS*, pages 1–12. Springer, Heidelberg, December 2005.

31. Sarah Meiklejohn, Pavel Kalinnikov, Cindy S. Lin, Martin Hutchinson, Gary Belvin, Mariana Raykova, and Al Cutter. Think global, act local: Gossip and client audits in verifiable data structures, 2020.

32. Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *Usenix Security*, pages 383–398, 2015.

33. Silvio Micali, Michael Rabin, and Joe Kilian. Zero-knowledge sets. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '03, page 80, USA, 2003. IEEE Computer Society.

34. Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th FOCS*, pages 120–130. IEEE Computer Society Press, October 1999.

35. microsoft.com. Teams end-to-end encryption. https://docs.microsoft.com/en-us/microsoftteams/teams-end-to-end-encryption, 2022. Accessed: 2022-05-26.

36. S. Muthukrishnan, Rajmohan Rajaraman, Anthony Shaheen, and Johannes Gehrke. Online scheduling to minimize average stretch. In *40th FOCS*, pages 433–442. IEEE Computer Society Press, October 1999.

37. Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 111–126. Springer, Heidelberg, August 2002.

38. Elaine Barker (NIST). Nist sp 800-57 part 1 rev. 5 recommendation for key management: Part 1 – general. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf, 2022. Accessed: 2022-08-10.

39. LLC. PCI Security Standards Council. Payment card industry data security standard: Requirements and testing procedures, v4.0. https://listings.pcisecuritystandards.org/documents/PCI-DSS-v4_0.pdf, 2022. Accessed: 2022-08-10.

40. David Pointcheval and Jacques Stern. Security proofs for signature schemes. In Ueli M. Maurer, editor, *EURO-CRYPT'96*, volume 1070 of *LNCS*, pages 387–398. Springer, Heidelberg, May 1996.

41. Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 256–266, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

42. signal.org. Technical information. https://www.signal.org/docs, 2016. Accessed: 2022-08-03.

43. signal.org. Technology preview: Signal private group system. https://signal.org/blog/signal-private-group-system/, 2019. Accessed: 2022-08-22.

44. Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1299–1316. ACM Press, November 2019.

45. Nirvan Tyagi, Ben Fisch, Joseph Bonneau, and Stefano Tessaro. Client-auditable verifiable registries. Cryptology ePrint Archive, Paper 2021/627, 2021. https://eprint.iacr.org/2021/627.

46. Ioanna Tzialla, Abhiram Kothapalli, Bryan Parno, and Srinath Setty. Transparency dictionaries with succinct proofs of correct operation. Cryptology ePrint Archive, Paper 2021/1263, 2021. https://eprint.iacr.org/2021/1263.

47. Elham Vaziripour, Justin Wu, Mark O'Neill, Jordan Whitehead, Scott Heidbrink, Kent Seamons, and Daniel Zappala. Is that you, alice? a usability study of the authentication ceremony of secure messaging applications. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 29–47, Santa Clara, CA, July 2017. USENIX Association.

48. webex.com. Webex end-to-end encryption. https://help.webex.com/en-us/article/WBX44739/What-Does-End-to-End-Encryption-Do?, 2022. Accessed: 2022-05-26.

49. whatsapp.com. Whatsapp encryption overview. White paper, 2021. Accessed: 2022-08-03.

50. Mark Zhandry. To label, or not to label (in generic groups). In *Advances in Cryptology – CRYPTO*, 2022.

A Additional Preliminaries

A.1 Decisional Diffie-Hellman Assumption

The *decisional Diffie-Hellman* (DDH) assumption on a group G of order p states that given a generator g and g^a, g^b for $a, b \xleftarrow{\$} \mathbb{Z}_p$, g^{ab} is indistinguishable from uniform. Formally,

Assumption 1 *There exists a family of groups G_λ of order $p(\lambda)$ with generator g such that for any efficient adversary $\mathcal{A}$,*

$$\left| \Pr_{a,b \xleftarrow{\$} \mathbb{Z}_p} [\mathcal{A}(g^a, g^b, g^{ab}) \to 1] - \Pr_{a,b,c \xleftarrow{\$} \mathbb{Z}_p} [\mathcal{A}(g^a, g^b, g^c) \to 1] \right| \leq negl(\lambda).$$

We also define DDH tuples, which correspond to the tuple (g, g^a, g^b, g^{ab}). Formally,

Definition 6 *A DDH tuple is a 4-tuple (g, h, g', h') such that there exists $\alpha \in \mathbb{Z}_p$ such that $g' = g^\alpha$ and $h' = h^\alpha$.*

A.2 Random Oracle Model

In the Random Oracle Model (ROM), introduced by Bellare and Rogaway [4], we treat some hash function $H : \{0,1\}^* \to S$ as a truly random function. We also allow "programming" the random oracle. That is, in our proofs for soundness and zero-knowledge for our cryptographic primitives, we model the hash function as being wrapped in an oracle. In the real game, the oracle will pass inputs directly to H, but in the ideal game, we will send oracle queries to H through the simulator, which can answer as it wishes. Note that by indistinguishability, we will require that the simulator's outputs to random oracle queries be computationally indistinguishable from a true random oracle.

A.3 Generic Group Model

In Shoup's Generic Group Model (GGM) [41], we treat some group G of order p as having group elements uniformly distributed along some set of bit-strings S. Formally, G is modelled as some truly-random bijective function $\tau : \mathbb{Z}_p \to S$, where the element of G associated to $x \in \mathbb{Z}_p$ is $\tau(x)$. Intuitively, for a generator g of G, $\tau(x)$ represents g^x. G also provides oracle access to an addition oracle, $\star : S \times S \times \{-1, 1\} \to S$ defined by $\star(g_1, g_2, b) = \tau(\tau^{-1}(g_1) + b \cdot \tau^{-1}(g_2))$ (denoted $g_1 g_2^b$).

In addition to the standard GGM, we add an additional exponentiation oracle $EXP : S \times \mathbb{Z}_p \to S$ to G, defined by $EXP(g, r) = \tau(\tau^{-1}(g) \cdot r)$ (denoted g^r). Note that this can efficiently be implemented by repeated squaring, but we add this oracle to model the fact that in some group implementations exponentiation is implemented in a more efficient manner. Assuming that τ is a bijection allows us to handle $\star$ queries on elements of S not coming from τ.

As a simplification, instead of G providing τ as an oracle, we have G simply provide $g = \tau(1)$. Any algorithm interacting with G can then calculate $\tau(x) = EXP(g, x)$ by using the EXP oracle.

We also allow "programming" the generic group similar to programming the random oracle. In the real games, queries to $\star$ and EXP will be forwarded to an honestly implemented generic group, but in the ideal game they will instead be forwarded to the simulator.

We consider Shoup's generic group model over Maurer's [30] for two reasons. First, Maurer's model does not allow us to treat random bit-strings as elements of the generic group, which is required for our definition. Even if this problem was solved through clever modeling, a recent result by Zhandry [50] shows that when both models apply, security in Shoup's model implies security in Maurer's for single-stage games (as all of our security games are).

B Merkle Trees and Covering Sets

Our Ordered Accumulator and Append-only Vector Commitment constructions are both implemented using Merkle trees. Merkle trees can be used to store both vectors (as with AVCs) as well as dictionaries of label/value pairs (as with OAs). Merkle trees can be represented by a compact commitment, and

they admit logarithmic-size proofs of inclusion and exclusion of a label/value pair with respect to the commitment.

For our constructions, a Merkle Tree is a binary tree where each node is associated with a unique bitstring label and a hash value. All leaf labels in the same tree are bitstrings with the same length l. Leaf nodes (which have no descendants) each store either an element of the vector (for AVCs, the ith element is stored in a leaf with label the l-bit representation of i) or a label-value pair (where the leaf label corresponds to the pair label). Given a hash function H (which we model as a random oracle), the hash value of a leaf node is computed as $H(\texttt{Leaf}, \mathsf{label}, \mathsf{val})$, while the hash value for an inner node is computed recursively as $H(\texttt{Internal}, \mathsf{label}, \mathsf{val}_{\mathsf{left}}, \mathsf{val}_{\mathsf{right}})$. In addition, we require that the parent of each leaf node have two children, and that for each node with label x, its left child's label begins with $x||0$ and its right child's label begins with $x||1$. In the OA, nodes that are not parents of leaf nodes may have only one child; in this case, $\mathsf{val}_{\mathsf{left}}$ or $\mathsf{val}_{\mathsf{right}}$ will be $\bot$.

We will use the root node's hash as a commitment to the whole data structure. Proofs consist of a list of node labels and their hashes which allows us to recompute the root hash while proving that specific leaves are included or excluded in the data structure. An adversary that can produce two different trees with the same commitment would be able to break the collision resistance of the hash function, and in the random oracle model, we extract these commitments to reconstruct such a data structure.

Covering Sets In order to represent consistency and inclusion proofs for the OA and AVC, we introduce the notion of a *covering set*. This definition is inspired by that in [31], but we generalize beyond history trees so it applies to both our OA and AVC constructions.

Given a Merkle Tree, a covering set S is a subset of the tree nodes such that no node in S is a descendant of another node in S. We define $\mathsf{cover}(S)$ as the set of leaf nodes in the tree that are descendants of some node in S, and define $\mathsf{range}(S)$ as the set of all l-bit-strings ℓ such that there exists a node $s \in S$ whose label is a prefix of ℓ. A set of covering sets is *mergeable* if the union of the subsets is also a covering set.

Given a Merkle Tree and a mergeable set of covering sets that collectively cover all the leaves in the tree, we can reconstruct the root hash from the hashes of the nodes in the covering set as follows. Algorithm MergeToRootHash takes as input the label and hash of each node in each covering set, and is able to compute the label for each node's parent due to the constraints imposed on the Merkle Tree structure, as discussed in Appendices C and D.1 regarding the OA and AVC constructions, respectively. The algorithm returns the hash of the root node.

▷ $h \leftarrow$ MergeToRootHash(covers) :
  – Let $U = \bigcup \mathsf{covers}$
  – If $U = \{\}$, **return** "''
  – While U contains more than one element:
    • Select a node $A = (l_A, v_A)$ in U for which l_A has maximum length in U.
    • Consider A's parent P and sibling B. Let l_A, v_A, l_B, v_B be the labels and values of A and B. Note that B may not exist in the Merkle Tree and thus it might be that $v_B = \bot$.
    • Compute $v_P = H(\texttt{Internal}, l_A, v_A, l_B, v_B)$.
    • Remove A and B (if it exists) from U and add (P, v_P).
  – **parse** U as $\{(l_R, v_R)\}$ and **return** v_R

Note that in the while loop, if P has two children, the child other than A must also be in U. If it weren't, some descendant of that child must be in U for all the leaves of the subtree of that child to be covered, but this contradicts our selection of A such that l_A has maximum length.

Additionally, each iteration of the while loop maintains the invariant that U is a covering set that covers all of the leaves of the tree. Finally, note that the algorithm always terminates because the sum of the lengths of the labels of all nodes in U is a non-negative quantity that decreases at each loop iteration. The number of nodes in the tree is an upper bound on the number of steps the algorithm takes.

To construct our OA and AVC (and their security proofs), we make frequent use of an algorithm GetCover(n, I) that, given a node n and an interval I, computes a covering set S such that

1. S contains only n or descendants of n
2. $\mathsf{range}(S) \subseteq I$

3. $\text{cover}(S)$ is exactly the set of descendants of n whose labels are in I.

Given two values $L, R \in \{0,1\}^l \cup \{-\infty, \infty\}$ (the set of all l-bit-strings extended with a minimum and maximum element), we define the open interval (L, R) to be the set of l-bit-strings s such that $L < s < R$ in lexicographic ordering, and define the closed interval $[L, R]$ to be the set of l-bit-strings s such that $L \le s \le R$.

$\triangleright\, h \leftarrow \mathsf{GetCover}(n, I)$:
   – If $\text{range}(n) \subseteq I$, **return** $\{n\}$
   – If $\text{range}(n) \cap I = \{\}$, **return** $\{\}$
   – If n has a left child n_l, set $C_l = \mathsf{GetCover}(n_l, I)$, else $C_l = \{\}$
   – If n has a right child n_r, set $C_r = \mathsf{GetCover}(n_r, I)$, else $C_r = \{\}$
   – **return** $C_l \cup C_r$

For efficiency reasons, we are interested in bounding the size of each cover.

Lemma 17. $\mathsf{GetCover}(n, I)$ *returns at most $2d$ nodes, where d is the height of the subtree rooted at n.*

Proof: To prove the lemma, it is enough to show that $\mathsf{GetCover}(n, I)$ returns at most 2 nodes at each depth. Assume that the interval is of the form $[L, R]$ (the case for open intervals is analogous). Given a bitstring ℓ, denote by ℓ_i the i-bit prefix of ℓ, defined as 0^i if $\ell = -\infty$ and 1^i if $\ell = \infty$. Assume by contradiction $\mathsf{GetCover}(n, [L, R])$ returns more than two nodes at the same depth. Let d be the smallest depth for which this occurs, and let $A_1, \dots, A_n$ be the $n \ge 3$ nodes at depth d, ordered lexicographically by label.

Note that the algorithm provides an in-order traversal of the tree, deciding whether to visit a node's children depending on the parent's label. In particular, no two nodes A_i, A_{i+1} returned by the algorithm can be siblings, as otherwise the algorithm would have returned the parent instead of recursing into the children, as the range of the parent is equal to the union of the range of its children. Additionally, $\forall i, A_i \ne n$, since n is the only node at that depth. Consider A_1, A_2, A_3's parents P_1, P_2, P_3, which therefore must be in the subtree rooted at n. Since $\text{range}(A_1), \text{range}(A_2), \text{range}(A_3) \subseteq [L, R]$, it must be that $L_d \le A_1 < A_2 < A_3 \le R_d$, which implies that $L_{d-1} \le P_1 < P_2 < P_3 \le R_{d-1}$. Thus, $\text{range}(P_2) \subseteq [L, R]$, and when the algorithm executed $\mathsf{GetCover}(P_2, [L, R])$, it would have returned P_2 instead of recursing into its children, contradicting that A_2 is part of the output of the algorithm, and proving the lemma. QED.

C Ordered Accumulator (OA)

We construct the Ordered Accumulator as a Merkle Tree, where each leaf's parent has exactly two children, and each non-leaf node N (except the root) has a parent whose label is equal to N's label without the last bit. Moreover, each label/value pair will be stored in the leaf of the tree with the same label (recall we require all labels to be strings of the same length), and with the leaf value being a tuple of the value and the epoch when the pair was added to the tree. This construction is inspired by the compressed Patricia trie described in SEEMless [8], but we instead keep (do not compress) internal nodes whose only child is not a leaf, in order to support a simpler and more database-friendly implementation.

Note that, given a set of label/value/epoch tuples, there exists only one Merkle tree that is consistent with the above constraints and contains exactly these tuples: starting from a complete binary tree, one can first iteratively remove all leaf nodes which do not correspond to one of the tuples above, and then iteratively remove all leaf parents that only have one child, connecting the leaf to the removed parent's parent. Similarly, given a cover set S which covers all the leaves in a given tree, all the ancestors of the nodes in S are fully determined. Therefore, the $\mathsf{MergeToRootHash}$ algorithm does not need to be explicitly given the full tree, but just the labels and hash values for the nodes that are in the cover, as the rest of the tree can be inferred according to the above rules.

We define the commitment to an Ordered Accumulator to be a pair (t, h) where t is the epoch number (the number of times the accumulator has been updated), and h is the current root hash value.

We give the construction for the Ordered Accumulator in Figure 10. The state of an ordered accumulator st consists of a map $\mathbf{D}(\mathsf{st})$ of $(\text{label}, \text{val}, t)$ tuples that are part of the data structure, as well as

a representation of the nodes in the Merkle Tree (which could alternatively be recomputed on the fly based on the datastore). For simplicity, in the pseudocode of Figure 10, we only store the root hash explicitly (which also includes the current epoch). Note that OA.ProveAll does not have any output in our construction, as OA.VerAll can deterministically recompute the Merkle Tree root from its other inputs.

C.1 Ordered Accumulator Soundness

Theorem 6 *If H is modeled as a random oracle, the ordered accumulator described in Figure 10 satisfies soundness.*

We need to show that there exists an extractor such that no adversary can distinguish between the ideal and real worlds with better than negligible probability. The extractor works as follows:

- The extractor's state consists of a set of pairs H (used to record answers to random oracle queries) and a set C that stores values that we do not want the random oracle to ever sample.
- Extract($\texttt{Ideal}, \mathsf{st}, in$):
  - If $(in, out) \in H$ for some out, **return** (out, st)
  - If in is of the form $(\texttt{Internal}, \mathsf{label}, \mathsf{val}_{\mathsf{left}}, \mathsf{val}_{\mathsf{right}})$, then add $\mathsf{val}_{\mathsf{left}}, \mathsf{val}_{\mathsf{right}}$ to C in st
  - Sample out from $\{0,1\}^\lambda \setminus C$
  - Add out to C and (in, out) to H in st.
  - **return** (out, st)
- Extract($\texttt{Extr}, \mathsf{st}, \mathsf{com}$):
  - **parse** com as (t, h). If $t = 0$, **return** $\{\}$
  - **return** ExtractFromHash($\mathsf{st}, \mathsf{h}, \text{``''}, t$)
- ExtractFromHash($\mathsf{st}, \mathsf{h}, s, t$):
  - Let (in, out) be the only pair in H (part of st) such that $out = \mathsf{h}$ (by construction, there is at most one). If no such pair exists, or $\mathsf{h} = \bot$, **return** $\{\}$
  - If in can be parsed as $(\texttt{Leaf}, \mathsf{label}, (e, v))$, $e \leq t$, and s is a prefix of label, **return** $\{(\mathsf{label}, (v, e))\}$
  - If in can be parsed as $(\texttt{Internal}, \mathsf{label}, \mathsf{val}_{\mathsf{left}}, \mathsf{val}_{\mathsf{right}})$ and s is a prefix of label, **return** ExtractFromHash($\mathsf{st}, \mathsf{val}_{\mathsf{left}}, \mathsf{label}\|0, t$) $\cup$ ExtractFromHash($\mathsf{st}, \mathsf{val}_{\mathsf{right}}, \mathsf{label}\|1, t$)
  - **return** $\{\}$

Note that the random oracle map H is, by construction, injective, since for any two distinct pairs $(in, out), (in', out')$ in H, we have $in \neq in'$ and $out \neq out'$. Moreover, every tuple returned by ExtractFromHash($\mathsf{st}, \mathsf{h}, s, t$) has a distinct label that is in the leaf range of the node with label s.

To argue that the ideal and real games are indistinguishable, we use a hybrid argument. Consider the following sequence of games.

- $Hyb0$: Ideal
- $Hyb1$: Like previous, but every time the adversary makes an ExtractD(com) oracle query where $\mathsf{com} = (t, \mathsf{h})$, the extractor adds h to C in st.
- $Hyb2$: Like previous, but we skip all the assertions on the various oracle calls which are not part of the real game. Importantly, $\texttt{Ideal}$ queries for the random oracle are still answered at random with a λ-bit-string not in C.
- $Hyb3$: Real

An adversary distinguishing real and ideal worlds would also be able to distinguish between two consecutive games with better than negligible probability.

First, note that $Hyb0 \approx Hyb1$ and $Hyb2 \approx Hyb3$. The only difference between the games in each pair is that answers to $\texttt{Ideal}$ queries are sampled uniformly at random from $\{0,1\}^\lambda \setminus C$, where the set C is slightly different in each case (and empty in $Hyb3$). The only way the adversary can notice this difference is if a value that would belong to C in $Hyb1$ (respectively, $Hyb2$) is actually sampled in $Hyb0$ (respectively, $Hyb3$). Given that for each $\texttt{Ideal}$ query we add at most 3 elements to C and for each $\texttt{Extract}$ query at most 1, the size of C is linear in the number of queries of the adversary, and thus the probability of sampling one of the elements of C from a space of exponential size is negligible in the security parameter.

Second, $Hyb1 \approx Hyb2$. To prove this, it is enough to show that each assertion is triggered in $Hyb2$ with at most negligible probability. We do so after introducing some notation and two auxiliary lemmas. Given a cover set S (each node represented as pair $(\mathsf{label}, \mathsf{h})$ containing its label and hash value), denote by ExtractAll(st, S, t) $= \bigcup_{(\mathsf{label}, \mathsf{h}) \in S}$ ExtractFromHash($\mathsf{st}, \mathsf{h}, \mathsf{label}, t$) for any st, t.

▷ pp ← OA.GenPP(1^λ):
  – Sample hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$
  – **return** pp ← (H, λ)

▷ (com, st) ← OA.Init(pp):
  – D ← {}, $t \leftarrow 0$
  – com ← $(t, \text{""})$
  – st ← (D, com)
  – **return** (com, st)

▷ (com', st', π_s) ← OA.Update(pp, st, $S = \{(\text{label}_1, \text{val}_1), \dots\}$):
  – **parse** st as (D, com), com as (t, h)
  – For each (label, val) $\in S$:
    • **ensure** label $\notin$ D
    • D[label] ← (val, $t + 1$)
    • Add the corresponding leaf to the Merkle Tree
  – Recompute the hash rootHash of the root $root$ of the tree (including the new leaves)
  – com' ← $(t + 1, \text{rootHash})$
  – Sort S by label as $\text{label}_1, \dots, \text{label}_n$
  – C ← GetCover($root, (-\infty, \text{label}_1)$) ∪ GetCover($root, (\text{label}_1, \text{label}_2)$) $\dots$
    GetCover($root, (\text{label}_{n-1}, \text{label}_n)$) ∪ GetCover($root, (\text{label}_n, +\infty)$) [that is, C is a cover of all nodes not in S]
  – $\pi \leftarrow (S, C)$
  – st ← (D, com')
  – **return** (com', st, π)

▷ 0/1 ← OA.VerifyUpd(pp, $\text{com}_0, \text{com}_1, \pi_S$):
  – **parse** com_0 as (t_0, h_0), com_1 as (t_1, h_1) and π_S as (S, C)
  – **ensure** $t_0 + 1 = t_1$
  – If $t_0 = 0$, **ensure** $h_0 = \text{""}$ and $C = \{\}$
  – **ensure** MergeToRootHash(C) = h_0
  – If $S = \{(\text{label}_1, \text{val}_1), \dots\}$, let S' be the set of leaf nodes, each having label label_i and hash $H(\texttt{Leaf}, \text{label}_i, (\text{val}_i, t_1))$
  – **ensure** MergeToRootHash($S' \cup C$) = h_1
  – **return** 1

▷ (π, val, t) ← OA.Query(pp, st, u, label):
  – **parse** st as (D, com), com as (t, h)
  – **ensure** $u \leq t$
  – If label $\in$ D, let (val, epno) ← D[label] if epno $\leq u$. Else, let (val, epno) ← $(\bot, \bot)$.
  – Compute the root node $root_u$ of the Merkle Tree including only the labels with associated epochs epno $\leq u$
  – π ← (GetCover($root_u, (-\infty, \text{label})$) ∪ GetCover($root_u, (\text{label}, +\infty)$)
  – **return** (π, val, epno)

▷ 0/1 ← OA.Verify(pp, com_u, label, val, i, π):
  – **parse** com_u as $(u, hash_u)$ and π as C
  – If val = $\bot$ or $i = \bot$:
    • **ensure** val = $i = \bot$
    • **ensure** label $\notin$ range(C)
    • **ensure** MergeToRootHash(C) = $hash_u$
    • **return** 1
  – Else:
    • Let L be the node with label label and hash value $H(\texttt{Leaf}, \text{label}, (\text{val}, i))$
    • **ensure** $i \leq u$
    • **ensure** MergeToRootHash($C \cup \{L\}$) = $hash_u$
    • **return** 1

▷ π ← OA.ProveAll(pp, st, u):
  – **return** {}

▷ 0/1 ← OA.VerAll(pp, com_u, i, P, π):
  – **parse** com_u as $(u, hash_u)$ and P as a collection $\{(\text{label}, \text{val}, i)\}_j$
  – Construct a set of leaves S' containing a leaf with label label and a hash $H(\texttt{Leaf}, \text{label}, (\text{val}, i))$ for each tuple (label, val, i) $\in P$
  – **ensure** each label is P is unique
  – **ensure** all tuples in P have epoch $i \leq u$
  – **ensure** MergeToRootHash(S') = $hash_u$
  – **return** 1

Fig. 10: OA construction.

Lemma 18. *For any adversary $\mathcal{A}$, let $D \leftarrow$ ExtractFromHash(st, h, label, t) and $D' \leftarrow$ ExtractFromHash(st', h, label, t') be the outputs of two calls with the same (label, h) input performed by the game during an execution of Hyb2 in response to queries by $\mathcal{A}$. We have that if $t = t'$ then $D = D'$, and if $t < t'$, then $D \subseteq D'$ and $D' \setminus D$ contains (label, (val, t'')) with $t < t'' \le t'$.*

Proof: First, let's consider the case $t = t'$. The only way the extractor would return a different output to the same query is if, during the first `Extract` query, one of the lookups into table H for some value was not found, and it was later added to H before the second query, or if a different pair for the same *out* was added. This cannot happen because every time a new tuple is added to H, we sample *out* such that it does not collide with any of the values ExtractFromHash would potentially seek (i.e., the value $h \in$ com, the value *out* itself, and the values $\text{val}_{\text{left}}, \text{val}_{\text{right}}$ for all previous random oracle queries performed by the adversary). Similarly, in the case $t < t'$, the extractor performs exactly the same lookup calls and obtains the same results from table H, but might choose to discard some leaf tuples with epoch t'' such that $t < t'' \le t'$ in the first but not in the second call. QED.

Lemma 19. *For any execution of Hyb2, and any cover set S, if $h = $ MergeToRootHash(S), then ExtractFromHash(st, h, "", t) = ExtractFromAll(st, S, t).*

Proof: To prove this, it is sufficient to notice that MergeToRootHash computes the hash of the root of the Merkle Tree by maintaining a set U and replacing each node (or pair of nodes) with their parent, computing the parent's hash through the function H. On the other hand, ExtractFromHash uses the same (injective) function H to visit the tree starting from the root node and then considering its children. Given that the two algorithms both consider the same root node and use the same hash function H to define the edges of the tree (and that H has no collisions and does not introduce new edges for existing nodes with a given hash across the execution of the experiment), it follows that ExtractFromHash will eventually consider all nodes in the cover S, and thus return the union of the outputs obtained by visiting them, which proves our result. QED.

Using these helper lemmas, we will now prove that $Hyb1 \approx Hyb2$.

– Query ExtractD(com). The assertion that $D[\text{com}] = D_{\text{com}}$ is always true due to Lemma 18. By construction, the assertion on the epoch is never triggered by our extractor (tuples whose epochs are too large are dropped).

– Query CheckVerD(com, label, val^*, i^*, π). In the case where $\text{val}^* = \bot$, since OA.Verify returns 1, it must be the case that $i^* = \bot$. In addition, MergeToRootHash(C) must return h_{root}, where com $= (t, h_{\text{root}})$, and C is a cover set consisting of the nodes in the set π. By Lemma 19, this implies that the output of Extract(`Extr`, com) equals ExtractAll(st, C, t). But since none of the nodes in C has a label label' that is the prefix of label, the set won't contain a tuple for label either, and thus label $\notin D[\text{com}]$ and the assertion isn't triggered. Similarly, if label $\ne \bot$, then the set C would contain a leaf node whose hash is $h = H(\texttt{Leaf}, \text{label}, (\text{val}^*, i^*))$ and thus ExtractFromHash(st, h, label', t') would return (label, (val*, i^*)) as a tuple, and thus $D[\text{com}][\text{label}] = (\text{val}^*, i^*)$, and the assertion won't be triggered.

– Query CheckVerUpdD(com$_0$, com$_1$, π). Since OA.VerifyUpd(pp, com$_0$, com$_1$, π) = 1, it must be that $t(\text{com}_0) = t(\text{com}_1) + 1$, and by construction if $t(\text{com}_0) = 0$, then $D[\text{com}_0] = \{\}$. Let com$_0 = (t_0, h_0)$, com$_1 = (t_1, h_1)$, $\pi = (R, S)$. Since MergeToRootHash(R) = h_0 and MergeToRootHash($R \cup S$) = h_1, we have that by Lemma 19, Extract(`Extr`, com$_0$) = ExtractAll(st, R, t_0) and Extract(`Extr`, com$_1$) = ExtractAll(st, $R \cup S, t_0 + 1$) = ExtractAll(st, $R, t_0 + 1$) $\cup$ ExtractAll(st, $S, t_0 + 1$). Therefore, by Lemma 18, we have that ExtractAll(st, R, t_0) $\subset$ ExtractAll(st, $R, t_0 + 1$) and therefore $D[\text{com}_0] \subseteq D[\text{com}_1]$ and that their difference only contains labels with $t + 1$ as the associated epochs. Therefore, no assertions are triggered for these queries.

– Query CheckVerAll(com, S, π). Let S' be the list of leaf nodes each containing one of the tuples in S. Again, since VerifyAll returns 1 and by Lemma 19, Extract(`Extr`, com) = ExtractAll(st, $S', t(\text{com})$). But S' only contains leaf nodes whose epochs are smaller than $t[\text{com}]$, therefore for each leaf node N,
ExtractFromHash(st, $N, t(\text{com})$) returns exactly the corresponding tuple, and thus $D[\text{com}]$ equals exactly the set of leaves in S.

D Append-only Vector Commitment - Definition

Definition 7 An Append-only Vector Commitment is a tuple of algorithms AVC = (GenPP, Init, Update, ProveExt, VerExt, Query, Verify).

```
AVC-Completeness(A):

pp' ← AVC.GenPP(1^λ)
(com', st') ← AVC.Init(pp)
assert com(st') = com' and t(com') = 0 and D(st_0) = {}
com_0 ← com', st ← st', t ← 0
A^{O...}(com_0)
return 1


Oracle Update(val):

(com', st', π) ← AVC.Update(st, val)
assert com(st') = com', t(com') = t + 1 and D(st') = D(st)||val
assert y ← AVC.VerExt(com_t, com', π); y = 1
com_{t+1} ← com', st ← st', t ← t + 1


Oracle Query(u, t'):

require 0 ≤ u ≤ t' ≤ t
(π, val') ← AVC.Query(st, u, t')
assert D(st)[u] = val'
assert y ← AVC.Verify(com_{t'}, u, val, π); y = 1


Oracle ProveExt(t_0, t_1):

require 0 ≤ t_0 ≤ t_1 ≤ t
π_E ← AVC.ProveExt(st, t_0, t_1)
assert y ← AVC.VerExt(com_{t_0}, com_{t_1}, π_E); y = 1
```

Fig. 11: Completeness for AVC. The scheme satisfies completeness if for any adversary $\mathcal{A}$, the output of the experiment is 1 with all but negligible probability.

Completeness We will say that an AVC satisfies completeness if for all PPT adversaries $\mathcal{A}$, the probability that the game described in Figure 11 doesn't return 1 is negligible in λ.

Soundness We will say that an AVC satisfies soundness if there exists an extractor Extract such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the two experiments described in Figure 12 is negligible in λ. Note that all the algorithms executed in the experiment get implicit access to the Ideal oracle, as they might need to make, e.g., random oracle calls.

D.1 Append-only Vector Commitment (AVC)

We construct the AVC using a History Tree, also known as a Merkle Mountain Range (as defined in [14,31]), as the underlying data structure, which we recall here for completeness. A History Tree with $N > 1$ leaves (which can be used to store a vector of N values) is a Merkle Tree in which the left subtree (the one rooted at the left child of the root) is a perfect tree with 2^i leaves for $i = \lceil \log(N) \rceil - 1$, and the right subtree is a history tree with $N - 2^i$ leaves. A History Tree with 1 leaf is exactly 1 leaf node. When adding a new leaf, if the right subtree becomes "full" at size 2^i, then it is incorporated into the left subtree, such that the root of the tree is the root of the new left subtree, and the new right subtree only contains the new leaf. Let M be the upper bound on the length of a given AVC, then we define $l = \lceil \log(M) \rceil$ to be the length of bitstring labels.

As in the OA case, the number of leaves in a History Tree fully determines its shape, i.e. the label (and hash) of the parent of each node. We define the i-th leaf's label in the tree (0-indexed) to be the l-bit

```
AVC-Sound-IDEAL(𝒜):                              Oracle CheckVerExt(com₀, com₁, π):

pp′, st ← Extract(Init)                          require 𝒵.VerExt(pp, com₀, com₁, π) = 1
L ← [ ], C ← [ ], st ← ⊥, pp ← pp′                  and com₀, com₁ ∈ D
b ← 𝒜^{Ideal(·),···}(pp)                          assert ∀ j ≤ t(com₀) :
return b                                             C[com₀][j] = C[com₁][j] ∧ L[com₀][j] = L[com₁][j]


Oracle Extract(com):                             Oracle Ideal(in):

L_com, C_com ← Extract(Extr, st, com)            out, st ← Extract(Ideal, st, in)
If com ∈ C:                                       return out
    assert C[com] = C_com ∧ L[com] = L_com
C[com] ← C_com, L[com] ← L_com
assert |C[com]| = |L[com]| = t(com)
    and last(C[com]) = com


Oracle CheckVer(com, u, val*, π):

require AVC.Verify(pp, com, u, val*, π) = 1 and com ∈ L
assert L[com][u] = val*
```

Fig. 12: Soundness for AVC. The extractor, on input com, returns a list L_{com} of size $t(com)$ as well as a list C_{com} which contains at position i the AVC digest corresponding to the sublist containing the first i elements of L_{com}. In the real experiment (not pictured), the public parameters are generated as $pp \leftarrow \mathcal{Z}.\mathsf{GenPP}(1^\lambda)$, and all oracles do nothing except for the Ideal one, which implements the ideal objects according to the specification. The scheme satisfies soundness if all PPT adversaries have negligible advantage in distinguishing these two games.

big endian representation of i (and use it to store the $i + 1$-th element of the vector, as for convenience, the vector is 1-indexed). All non-leaf nodes have two children, and we define the label of an inner node to be the longest common prefix of its children's labels. Note that this means that, unlike in the OA case, the root of a History Tree might not have the empty string as the label, and instead have a sequence of 0 bits.

We give a formal construction of our AVC in Figure 13. The state of an AVC st consists of a vector $\mathbf{D}(\mathsf{st})$ of values that are part of the data structure (the i-th value is indicated with $\mathbf{D}(\mathsf{st})[i]$, with i being 1-indexed), and a representation of the nodes in the Merkle Tree (which could alternatively be recomputed on the fly based on the list $\mathbf{D}(\mathsf{st})$). For simplicity, in the pseudocode of Figure 13, we only store the root hash explicitly.

D.2 Append-only Vector Commitment Soundness

Theorem 7 *If H is modelled as a random oracle, the append-only vector commitment described in Figure 13 satisfies Soundness.*

Proof. The proof is similar to the one of the OA theorem. We need to show that there exists an extractor Extract such that no adversary can distinguish between the games in Figure 12 with better than negligible probability. The extractor works as follows:

– The extractor's state consists of a set of pairs H (used to record answers to random oracle queries), and a set C that stores values that we do not want the random oracle to ever sample.
– To handle Extract(Ideal, st, com):
  • If (in, out) in H for some out, **return** (out, st).
  • If in is of the form (Internal, label, leftHash, rightHash), then add leftHash, rightHash to C in st
  • Sample out from $\{0, 1\}^\lambda \setminus C$.
  • Add out to C and (in, out) to H in st. **return** (out, st)
– To handle Extract(Extr, st, com):
  • **parse** com as $(t, hash)$. If $t = 0$, **return** $\{\}$. Let rootLabel be the label of the root node of a history tree with λ-bit leaf labels and t leaves.
  • $D, S \leftarrow$ ExtractFromHash$(st, hash, \mathsf{rootLabel}, t)$.
  • Let P be a vector of size t with all entries initialized to $\bot$.
  • For $i = 1, \ldots, t$:

$\triangleright$ pp $\leftarrow$ AVC.GenPP(1^λ):
 – Sample hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$
 – **return** pp $\leftarrow (H, \lambda)$

$\triangleright$ (com, st) $\leftarrow$ AVC.Init(pp):
 – D $\leftarrow$ []
 – com $\leftarrow$ (0, "")
 – st $\leftarrow$ (D, com)
 – **return** (com, st)

$\triangleright$ (com$'$, st$'$, π_s) $\leftarrow$ AVC.Update(pp, st, val):
 – **parse** st as (D, com)
 – D $\leftarrow$ D$||val$
 – Compute the root node $root$ of the Merkle Tree who has elements of D in its leaf nodes, and its hash value rootHash
 – com$'$ $\leftarrow$ (|D|, rootHash)
 – $\pi \leftarrow$ (GetCover($root, [1, t]$), GetCover($root, [t+1, t+1]$))
 – st $\leftarrow$ (D, com$'$)
 – **return** (com$'$, st, π)

$\triangleright$ $\pi \leftarrow$ AVC.ProveExt(pp, st, t', t):
 – **parse** st as (D, com)
 – **ensure** $t' < t \leq$ |D|
 – Let $root_t$ be the root of the Merkle Tree built from the first t elements of D
 – $\pi \leftarrow$ (GetCover($root_t, [1, t']$), GetCover($root_t, [t'+1, t]$))
 – **return** π

$\triangleright$ 0/1 $\leftarrow$ AVC.VerExt(pp, com$'$, com, π):
 – **parse** π as (P', P), com$'$ as $(t', hash')$ and com as $(t, hash)$
 – **ensure** range(P') $= [1, t']$ and range($P' \cup P$) $= [1, t]$
 – **ensure** MergeToRootHash(P') $= hash'$
 – **ensure** MergeToRootHash($P' \cup P$) $= hash$
 – **return** 1

$\triangleright$ (π, val) $\leftarrow$ AVC.Query(pp, st, u, t'):
 – **parse** st as (D, com)
 – **ensure** $u \leq t' \leq$ |D|
 – $val \leftarrow$ D[u]
 – Let $root_{t'}$ be the root node of the Merkle Tree built from the first t' elements of D
 – $\pi \leftarrow$ (GetCover($root_{t'}, [1, u-1]$) $\cup$ GetCover($root_{t'}, [u+1, t']$))
 – **return** (π, val)

$\triangleright$ 0/1 $\leftarrow$ AVC.Verify(pp, com$_{t'}$, u, val, π):
 – **parse** com$_{t'}$ as $(t', hash_{t'})$
 – **parse** π as C
 – Let L be a leaf node with label u and a hash $H(\texttt{Leaf}, u, val)$
 – **ensure** range($C \cup \{L\}$) $= [1, t']$
 – **ensure** MergeToRootHash($C \cup \{L\}$) $= hash_{t'}$
 – **return** 1

Fig. 13: AVC construction.

∗ If the set of nodes in S contains a cover set C such that $\mathsf{range}(C) = [1, i]$, then compute $hash_i \leftarrow \mathsf{MergeToRootHash}(C)$ and set $P[i] \leftarrow (i, hash_i)$
- **return** D, P

We define the helper function:

▷ $D, S \leftarrow \mathsf{ExtractFromHash}(\mathsf{st}, hash, s, t)$:
  - If $hash = \bot$, **return** $\{\}, \{\}$
  - Let (in, out) be the only pair in H (part of st) such that $out = hash$ (by construction, there is at most one). If no such pair exists, **return** $\{\}, \{(s, hash)\}$.
  - If in can be parsed as $(\texttt{Leaf}, \mathsf{label}, \mathsf{val})$, $\mathsf{label} = s$, and s is the label of a leaf node in a history tree with λ-bit labels and t leaves, then **return** $\{(s, \mathsf{val})\}, \{(s, hash)\}$
  - If in can be parsed as $(\texttt{Internal}, \mathsf{label}, \mathsf{leftHash}, \mathsf{rightHash})$, and $\mathsf{label} = s$ then:
    • Let $\mathsf{leftLabel}, \mathsf{rightLabel}$ be the labels of the left and right children of the node with label label in a history tree with λ-bit labels and t leaves
    • $D_L, S_L \leftarrow \mathsf{ExtractFromHash}(\mathsf{st}, \mathsf{leftHash}, \mathsf{leftLabel}, t)$
    • $D_R, S_R \leftarrow \mathsf{ExtractFromHash}(\mathsf{st}, \mathsf{rightHash}, \mathsf{rightLabel}, t)$
    • **return** $D_L \cup D_R, S_L \cup S_R \cup \{(s, hash)\}$
  - **return** $\{\}, \{(s, hash)\}$

To prove that the real game is indistinguishable from the ideal one with this extractor, one can use a hybrid argument similar to the one of Theorem 6. QED.

E   Random Oracle Commitments

We require that our commitments satisfy a simulatability property in order to achieve zero-knowledge in the final protocol. This definition is a simplification of the one in [9].

Definition 8 (Simulatable Commitments) A Simulatable Commitment Scheme C consists of 3 algorithms $(\mathsf{C.Init}, \mathsf{C.Commit}, \mathsf{C.Verify})$.
The algorithms must satisfy the following properties:
Correctness: For all security parameters $\lambda \in \mathbb{N}$ and for all m,

$$\Pr[\mathsf{pp} \leftarrow \mathsf{C.Init}(1^\lambda); \mathsf{com}, \mathsf{aux} \leftarrow \mathsf{C.Commit}(\mathsf{pp}, m) :$$
$$\mathsf{C.Verify}(\mathsf{pp}, \mathsf{com}, m, \mathsf{aux}) = 1] = 1$$

Simulatability: For all security parameters $\lambda \in \mathbb{N}$ there exists a simulator $\mathcal{S}$ such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the games of Figure 14 is negligible.
Extractability: For all security parameters $\lambda \in \mathbb{N}$ there exists an extractor Extract such that for all PPT adversaries $\mathcal{A}$, the advantage of $\mathcal{A}$ in distinguishing the games of Figure 15 is negligible.

In Figure 16, we recall the folklore construction of Random Oracle based commitments. In the following, we prove that it satisfies our security definitions.

Theorem 8 *If H is modelled as a random oracle, the commitment scheme* RoComm *described in Figure 16 satisfies Simulatability.*

Proof. We need to show that there exists a (stateful) simulator $\mathcal{S}$ such that no adversary can distinguish between C-Sim-IDEAL and C-Sim-REAL with better than negligible probability.
  In the real world, the Ideal oracle implements H by sampling a new uniformly random λ-bit output for each new input, and storing it in a table which we still call H (with a slight abuse of notation) so that the same query is always given the same output.
  We prove the theorem through a hybrid argument. Consider the following sequence of games:

- *Hyb*0. Defined as C-Sim-REAL($\mathcal{A}$).

```
C-Sim-REAL(A):                                     C-Sim-IDEAL(A):
─────────────────                                  ──────────────────
pp ← C.Init(1^λ), D ← {}                           pp ← S(Init), D ← {}
b ← A^···(pp)                                       b ← A^···(pp)
return b                                            return b


Oracle Commit(m):                                  Oracle Commit(m):
─────────────────                                  ─────────────────
com, aux ← C.Commit^{Ideal(·)}(pp, m)              com ← S(Commit)
D[com] ← (m, aux)                                  D[com] ← m
return com                                         return com


Oracle Open(com):                                  Oracle Open(com):
─────────────────                                  ─────────────────
require com ∈ D                                    require com ∈ D
return D[com]                                      return S(Open, com, D[com])


Oracle Ideal(in):                                  Oracle Ideal(in):
─────────────────                                  ─────────────────
return Ideal(in)                                   return S(Ideal, in)
```

Fig. 14: Games for Commitment Simulatability. S is a stateful algorithm.

```
C-Extr-IDEAL(A):
──────────────────
pp, st ← Extract(Init), D ← {}
b ← A^{Ideal(),···}(pp)
return b


Oracle Extract(com):
──────────────────
m ← Extract(Extract, st, com)
If com ∈ D, assert D[com] = m
D[com] ← m


Oracle CheckVer(com, m, aux):
──────────────────
require com ∈ D
require C.Verify^{Ideal(·)}(pp, com, m, aux) = 1
assert D[com] = m


Oracle Ideal(in):
──────────────────
out, st ← Extract(Ideal, st, in)
return out
```

Fig. 15: Commitment Extractability Game. In the real world (not pictured), the public parameters are generated as $pp \leftarrow C.Init(1^\lambda)$ and the oracles do not do anything, except for Ideal, which implements the ideal objects (such as random oracles) the scheme requires.

```
▷ pp ← C.Init(1^λ):
  – Sample hash function H : {0,1}* → {0,1}^λ
  – return (λ, H)

▷ c, aux ← C.Commit(pp, m):
  – aux ←$ {0,1}^λ; c ← H(m, aux)
  – return c, aux

▷ 0/1 ← C.Verify(pp, c, m, aux):
  – return H(m, aux) =? c
```

Fig. 16: Random Oracle Commitment Scheme construction.

– $Hyb1$. Defined as the previous hybrid, except that the random function H is replaced with a random injective function: to answer `Ideal` queries, the game repeats the sampling whenever it obtains a value which had already been returned before for a different input.
– $Hyb2$. Defined as the previous hybrid, but the game samples aux values (as part of executing C.Commit for a Commit query) uniformly at random, but excluding values aux′ such that there exists m, com where $H((m, \mathsf{aux}')) = \mathsf{com}$.
– $Hyb3$. Defined as the previous hybrid, but the game returns ERROR if the adversary makes an Ideal query for a pair (m, aux) such that aux was sampled by the Commit oracle during a query for m, but the resulting commitment was never input to the Open oracle.
– $Hyb4$. Defined as the previous hybrid, but instead of sampling the values aux during Commit queries, they are sampled during Open queries.
– $Hyb5$. Defined as C-Sim-IDEAL($\mathcal{A}$), with the following simulator $\mathcal{S}$:
  • It keeps a table H of tuples (in, out), intended to map arbitrary-length in, to λ-length out. We refer to the set of the first elements in of each tuple in H as $H.in$, and the second elements as $H.out$.
  • $\mathcal{S}(\mathtt{Ideal}, in)$:
    ∗ If there exists out: $(in, out) \in H$, **return** out.
    ∗ $out \xleftarrow{\$} \{0,1\}^\lambda \setminus H.out$. Add (in, out) to H
    ∗ **return** out
  • $\mathcal{S}(\mathtt{Commit})$:
    ∗ $\mathsf{com} \xleftarrow{\$} \{0,1\}^\lambda \setminus H.out$
    ∗ **return** com
  • $\mathcal{S}(\mathtt{Open}, \mathsf{com}, m)$:
    ∗ $\mathsf{aux} \xleftarrow{\$} \{0,1\}^\lambda \setminus AUX$, where $AUX = \{\mathsf{aux}' : ((m, \mathsf{aux}'), out) \in H\}$
    ∗ Add $((m, \mathsf{aux}), \mathsf{com})$ to H
    ∗ **return** (m, aux)

$Hyb0 \approx Hyb1$. The adversary has the same view unless during an execution of $Hyb0$ there is a collision in the random oracle. This happens with at most negligible probability.

$Hyb1 \approx Hyb2$. The two games are identical unless during an execution of $Hyb1$ the game samples a value aux such that $((m, \mathsf{aux}), out) \in H$ for some m, out. Since there are polynomially many such values, and the sampling space is exponential, a collision happens with at most negligible probability.

$Hyb2 \approx Hyb3$. The games are identical unless the adversary makes a query that returns ERROR. Note that there are only polynomially many values aux that would cause such a query, but they are information theoretically hidden from the adversary, who can only guess them with negligible probability.

$Hyb3 \approx Hyb4$. Given that the values aux are information theoretically hidden from the adversary, the view of the adversary has the same distribution in the two games conditioned on the fact that there are no queries that return ERROR in $Hyb3$. But since $Hyb3 \approx Hyb2$, this happens with at most negligible probability.

$Hyb4 \approx Hyb5$. The output of the two games is identical, as they execute the same instructions to answer all queries, but in $Hyb5$ the code is organized differently by grouping some instructions into the simulator.

QED.

Theorem 9 *If H is modelled as a random oracle, the commitment scheme* RoComm *described in Figure 16 satisfies Extractability.*

Proof. We need to show that there exists an extractor such that for all PPT adversaries the distinguishing advantage is negligible.

In the real world, the Ideal oracle implements H by sampling a new uniformly random λ-bit output for each new input, and storing it in a table H (part of its state) so that the same query is always given the same output.

The extractor works as follows:

- It keeps a table H as part of its state, and uses it to handle Ideal queries as in the real world.
- To answer `Extract` queries on input com, the extractor checks if there exists (m, aux) such that $((m, \text{aux}), \text{com}) \in H$. If so, it returns m (or the lexicographically first such m if more than one tuple exists), otherwise it returns a special symbol $\bot$ (not part of the message space).

First of all, we note that the distribution of the answers to ideal queries has the same distribution in both games. Therefore, to prove the theorem it is enough to argue that the assertions that are part of Extract and Open queries are triggered with at most negligible probability.

First consider the assertion that is part of Extract oracle queries. This is triggered if the extractor returns two different outputs when queried on the same input. For this to happen, there must either be a collision in the random oracle (two inputs mapping to the same output), or the adversary needs to make an Extract query for a value com, and such value needs to be sampled as the output of a subsequent Ideal query. Since the space of random oracle output is exponential and the adversary makes at most polynomially many queries, either event can happen with at most negligible probability.

A similar argument can be made for the assertion during CheckVer queries: if the assertion is triggered, it has to be that C.Verify(pp, com, aux, m') = 1 (and therefore $((m', \text{aux}), \text{com}) \in H$) where $m' \neq D[\text{com}]$. There are two possibilities:

- $D[\text{com}] = \bot$. This occurs when, at the time the Extract(com) query was made, com was not part of H. The probability of one such com being sampled at random as part of a subsequent query is negligible.
- $D[\text{com}] \neq \bot$. In this case, there are two tuples in H mapping to the same output com, which again can happen with at most negligible probability as we argued above.

QED.

F   Proof of Theorem 4

Proof. Let C.Extract, OA.Extract, VRF.Extract, AVC.Extract be extractors from the soundness definitions of the various primitives. We can construct an extractor RZKS.Extract for the RZKS soundness game as follows:

RZKS.Extract :

- Internally run an instance of each of the above extractors, each with their own independent state. Set RZKS.st = (OA.st, VRF.st, C.st, AVC.st).
- To handle Extract(`Ideal`, st, in):
  - Determine to which primitive P the input in is referring. Each primitive implements their own independent ideal objects (such as domain separated random oracles).
  - out, P.st$'$ ← P.Extract(`Ideal`, P.st, in) where P.st is parsed from st.
  - Update st with P.st$'$; **return** (out, st)
- To handle Extract(`ExtrC`, st, com):
  - **parse** RZKS.st as (OA.st, VRF.st, C.st, AVC.st)
  - L, C ← AVC.Extract(`Extr`, AVC.st, com)
  - **return** C
- To handle Extract(`ExtrD`, st, com):
  - **parse** RZKS.st as (OA.st, VRF.st, C.st, AVC.st)
  - L, C ← AVC.Extract(`Extr`, AVC.st, com)
  - **parse** $last(L)$ as $(\text{com}_{\text{OA}}, pk)$
  - D_{OA} ← OA.Extract(`ExtrD`, OA.st, com_{OA})
  - $D_{\text{com}} = \{\}$; For each $(\text{tlbl}, \text{tval}, t) \in D_{\text{OA}}$:

* label ← VRF.Extract(VRF.st, pk, tlbl)
* val ← C.Extract(C.st, tval)
* If label $\neq \perp$, val $\neq \perp$, and label $\notin D_{\text{com}}$ and $1 \leq t \leq$ RZKS.$\mathbf{t}$(com), then $D_{\text{com}}[\text{label}] \leftarrow (\text{val}, t)$
  * **return** D_{com}

We want to prove that for all adversaries, the two RZKS-Sound games instantiated with RZKS.Extract are computationally indistinguishable for any adversary $\mathcal{A}$. To do so, let's consider a sequence of hybrids:

– $Hyb0$. This is defined as RZKS-Sound-IDEAL($\mathcal{A}$).
– $Hyb1$. Defined as $Hyb0$, but the extractor keeps additional maps $D_{\text{OA}}, D_{\text{C}}, D_{\text{VRF}}, L_{\text{AVC}}, C_{\text{AVC}}$ to track the outputs of each sub-extractor call to make sure it is consistent across multiple calls (we stress that in $Hyb0$ the extractor is not allowed to update its state during ExtrD queries). More in detail, every time the extractor (while handling an ExtrD or ExtrC query) makes a call $L_{\text{com}}, C_{\text{com}} \leftarrow$ AVC.Extract($\texttt{ExtrC}$, AVC.st, com), it checks if com $\in C_{\text{AVC}}$ and, if so, it asserts $C_{\text{AVC}}[\text{com}] = C_{\text{com}}$ and $L_{\text{AVC}}[\text{com}] = L_{\text{com}}$; otherwise it sets $C_{\text{AVC}}[\text{com}] \leftarrow C_{\text{com}}$ and $L_{\text{AVC}}[\text{com}] \leftarrow L_{\text{com}}$. Then, while handling an ExtrD query, every time the extractor makes a call $E \leftarrow$ OA.Extract($\texttt{ExtrD}$, OA.st, com$_{\text{OA}}$), it checks if com$_{\text{OA}} \in D_{\text{OA}}$ and, if so, it asserts $D_{\text{OA}}[\text{com}_{\text{OA}}] = E$; otherwise it sets $D_{\text{OA}}[\text{com}_{\text{OA}}] \leftarrow E$. Analogously, outputs of C.Extract(C.st, tval) are stored in $D_{\text{C}}[\text{tval}]$ and outputs of VRF.Extract(VRF.st, pk, tlbl) in $D_{\text{VRF}}[pk][\text{tlbl}]$ (with assertions being triggered if different values are already in those maps).
– $Hyb2$. Defined as the previous hybrid, except:
    * During ExtractD(com) queries, the game also enforces that every time a new tuple (tlbl, label) is added to $D_{\text{VRF}}[pk]$ by the extractor (for some pk), that there is no other tlbl$' \neq$ tlbl such that $D_{\text{VRF}}[pk][\text{tlbl}'] = \text{label}$.
    * During CheckVerD(com, label, val*, i^*, π) queries, the game also parses π as ($\pi_{\text{AVC}}, \pi_{\text{OA}}, \pi_{VRF}$, tlbl*, tval*, aux, com$_{\text{INT}}$) and com$_{\text{INT}}$ as (com$_{\text{OA}}$, pk). Then, it checks if tlbl$^* \in D_{\text{VRF}}[pk]$ and, if not, it computes $D_{\text{VRF}}[pk][\text{tlbl}^*] \leftarrow$ VRF.Extract($\texttt{Extr}$, st$_{\text{VRF}}$, pk, tlbl*) (and asserts that there are no other tlbl$'$ mapping to the same label for the same pk as above). Then it asserts that $D_{\text{VRF}}[pk][\text{tlbl}^*] = \text{label}$.
    * During CheckVerUpdD(com^a, com^b, π), the game also parses π as (π', π_{AVC}, com$_{\text{INT}}^{t_1}$, com$_{\text{INT}}^{t_0}$, $\pi_{\text{AVC}}^{t_1}$, $\pi_{\text{AVC}}^{t_0}$), com$_{\text{INT}}^i$ as (com$_{\text{OA}}^i$, pk_i) for $i \in \{t_0, t_1\}$. Then, if $pk_{t_0} = pk_{t_1}$, it proceeds as in $Hyb1$. Otherwise, it parses π' as ($\pi_{\text{OA}}, \pi_{\text{OA}}^{\text{g}-1}, \pi_{\text{OA}}^{\text{g}}, \pi_{\text{VRF}}, \text{com}_{\text{OA}}', \{(\text{tlbl}_j^{g-1}, \text{tlbl}_j^g, \text{tval}_j, \text{epno}_j)\}_{j \in L}$) and asserts that for all $j \in L$, $D_{\text{VRF}}[pk_{t_0}][\text{tlbl}_j^{g-1}] = D_{\text{VRF}}[pk_{t_1}][\text{tlbl}_j^g]$.
– $Hyb3$. Defined as the previous hybrid, except:
    * During CheckVerD(com, label, val*, i^*, π) queries, the game also parses com$_{\text{OA}}$ from π (as in the previous hybrid). It asserts that if tval$^* = \perp$ or $i^* = \perp$, then tlbl$^* \notin D_{\text{OA}}[\text{com}_{\text{OA}}]$, and otherwise that $D_{\text{OA}}[\text{com}_{\text{OA}}][\text{tlbl}^*] = (\text{tval}^*, i^*)$.
    * During CheckVerUpdD(com^a, com^b, π) queries, the game also parses (com$_{\text{OA}}^i$, pk_i) for $i \in \{t_0, t_1\}$ as in the previous hybrid and asserts that, if $pk_{t_0} = pk_{t_1}$, that:
        * $D_{\text{OA}}[\text{com}_{\text{OA}}^{t_0}] \subseteq D_{\text{OA}}[\text{com}_{\text{OA}}^{t_1}]$
        * $\forall (\text{tlbl}, \text{tval}, t) \in D_{\text{OA}}[\text{com}_{\text{OA}}^{t_1}] \setminus D_{\text{OA}}[\text{com}_{\text{OA}}^{t_0}] : t = \text{RZKS.}\mathbf{t}(\text{com}^b)$
        * either RZKS.$\mathbf{t}$(com^a) $\neq 0$ or $D_{\text{OA}}[\text{com}_{\text{OA}}^{t_0}] = \{\}$.
      Otherwise, if $pk_{t_0} \neq pk_{t_1}$, the game parses π' as ($\pi_{\text{OA}}, \pi_{\text{OA}}^{\text{g}-1}, \pi_{\text{OA}}^{\text{g}}, \pi_{\text{VRF}}, \text{com}_{\text{OA}}', \{(\text{tlbl}_j^{g-1}, \text{tlbl}_j^g, \text{tval}_j, \text{epno}_j)\}_{j \in L}$) and asserts that:
        1. $|D_{\text{OA}}[\text{com}_{\text{OA}}^{t_0}]| = |D_{\text{OA}}[\text{com}_{\text{OA}}']| = |L|$ and $\forall j \in L, D_{\text{OA}}[\text{com}_{\text{OA}}^{t_0}][\text{tlbl}_j^{g-1}] = D_{\text{OA}}[\text{com}_{\text{OA}}'][\text{tlbl}_j^g] = (\text{tval}_j, \text{epno}_j)$
        2. $D_{\text{OA}}[\text{com}_{\text{OA}}'] \subseteq D_{\text{OA}}[\text{com}_{\text{OA}}^{t_1}]$
        3. $\forall (\text{tlbl}, \text{tval}, t) \in D_{\text{OA}}[\text{com}_{\text{OA}}^{t_1}] \setminus D_{\text{OA}}[\text{com}_{\text{OA}}'] : t = \mathbf{t}(\text{com}^b)$, and that either $\mathbf{t}(\text{com}_{\text{OA}}') \neq 0$ or $D_{\text{OA}}[\text{com}_{\text{OA}}'] = \{\}$.
– $Hyb4$. Defined as the previous hybrid, but
    1. During CheckVerD oracle queries, the game also asserts that com$_{\text{INT}} = last(L_{\text{AVC}}[\text{com}])$.
    2. During CheckVerUpdD queries, the game also asserts that $last(L_{\text{AVC}}[\text{com}^a]) = \text{com}_{\text{INT}}^{t_0}$ and that $L_{\text{AVC}}[\text{com}^b] = L_{\text{AVC}}[\text{com}^a] || \text{com}_{\text{INT}}^{t_1}$.
    3. During CheckVerUpdC queries and CheckVerExt queries, the game also asserts that $\forall j \leq \mathbf{t}(\text{com}^a)$, $L_{\text{AVC}}[\text{com}^a][j] = L_{\text{AVC}}[\text{com}^b][j]$

- *Hyb*5. Defined as the previous hybrid, except:
  - RZKS.Extract does not leverage C.Extract any more. In particular, `Ideal` queries for objects related to C by $\mathcal{A}$ are answered with a "real" implementation of such objects (i.e., random oracles queries are answered using a uniformly random function). `ExtrD` queries are handled by skipping the call to C.Extract (and therefore the val $\neq \bot$ condition in the following if statement) and instead setting $D_{\mathsf{com}}[\mathsf{label}] \leftarrow (\mathsf{tval}, t)$. The game also does not keep track of D_C any more and doesn't enforce the related assertions.
  - CheckVerD queries are handled as in the previous hybrid, except that rather than asserting that $D[\mathsf{com}][\mathsf{label}] = (\mathsf{val}^*, i^*)$, the game asserts $D[\mathsf{com}][\mathsf{label}] = (\mathsf{tval}^*, i^*)$.
  - CheckVerUpdD queries still assert that $D[\mathsf{com}^a] \subseteq D[\mathsf{com}^b]$ (where $D[\mathsf{com}]$ now maps labels to pairs of commitments and periods) and $D_\mathsf{OA}[\mathsf{com}_\mathsf{OA}^{t_1}] \subseteq D_\mathsf{OA}[\mathsf{com}_\mathsf{OA}^{t_0}]$ (where $D_\mathsf{OA}[\mathsf{com}_\mathsf{OA}^{t_0}]$ maps VRF outputs to pairs of commitments and periods).
- *Hyb*6. Defined as the previous hybrid, except:
  - RZKS.Extract does not leverage VRF.Extract any more. In particular, `Ideal` queries for objects related to C and VRF by $\mathcal{A}$ are answered with a "real" implementation of such objects (i.e., random oracles queries are answered using a uniformly random function). `ExtrD` queries are handled by skipping the call to VRF.Extract (and C.Extract) as well as the test for $\mathsf{label} \in D$, and instead setting $D_{\mathsf{com}}[\mathsf{tlbl}] \leftarrow (\mathsf{tval}, t)$. The game also does not keep track of D_VRF any more and thus does not enforce the new assertions introduced in *Hyb*2.
  - CheckVerD queries are handled as in the previous hybrid, except that rather than asserting that $D[\mathsf{com}][\mathsf{label}] = (\mathsf{tval}^*, i^*)$, the game asserts $D[\mathsf{com}][\mathsf{tlbl}^*] = (\mathsf{tval}^*, i^*)$. Similarly in the if statement leading up to and including the check that $D[\mathsf{com}][\mathsf{label}] = (\mathsf{tval}^*, i^*)$, all references to label are replaced by tlbl^*.
  - CheckVerUpdD queries are handled as in the previous hybrid (but now $D[\mathsf{com}]$ is the same as $D_\mathsf{OA}[\mathsf{com}_\mathsf{OA}]$ as described above).
- *Hyb*7. Defined as the previous hybrid, except:
  - RZKS.Extract does not leverage OA.Extract any more. In particular, `Ideal` queries for objects related to OA, C and VRF by $\mathcal{A}$ are answered with a "real" implementation of such objects (i.e., random oracles queries are answered using a uniformly random function). `ExtractD` queries are handled by only calling AVC.Extract and returning $D_\mathsf{com} = \{\}$ (skipping all the calls to OA.Extract, C.Extract, VRF.Extract). In the game, we still set $D[\mathsf{com}] \leftarrow D_\mathsf{com}$, so that "**require** com $\in D$" constraints can be enforced, but assertions related to the values in D are skipped. ExtractC queries are handled as in the previous hybrid.
  - CheckVerD queries are handled by requiring com $\in D$ (which implies com $\in C$) and running RZKS.Verify as in the previous hybrid, and then only asserting $last(L[\mathsf{com}]) = \mathsf{com}_\mathsf{INT}$ (as introduced in *Hyb*4). The assertions on D are skipped. Similarly, CheckVerUpdD queries are handled as in the previous hybrid, but skipping the assertions involving D.
- *Hyb*8. This is defined as RZKS-Sound-REAL($\mathcal{A}$).

Any adversary distinguishing the first from the last game with non-negligible advantage must also have non-negligible advantage in distinguishing a couple of consecutive hybrids.

$Hyb0 \approx Hyb1$. Assume by contradiction that there exists $\mathcal{A}$ that can distinguish $Hyb0$ from $Hyb1$. By construction, $\mathcal{A}$ needs to trigger one of the new assertions in $Hyb1$ with better than negligible probability. We can leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the soundness of one of the underlying primitives. We show the proof in the case where the adversary triggers the assertion related to D_VRF, which we reduce to the soundness of the VRF (the others are analogous). In particular, we will describe an adversary $\mathcal{B}$ that, when executed in VRF-Sound-IDEAL, simulates for $\mathcal{A}$ an execution of $Hyb1$. We will show that if $\mathcal{A}$ triggers one of the assertions about D_VRF introduced in $Hyb1$, then $\mathcal{B}$ also triggers an assertion in VRF-Sound-IDEAL with at least the same probability. Since the advantage of $\mathcal{B}$ in the VRF soundness game is no smaller than the probability of triggering this assertion, we can conclude that this can only happen with negligible probability.

$\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $Hyb1$, with the following modifications:

- $\mathcal{B}$ internally runs a copy of OA.Extract, C.Extract, AVC.Extract and uses them to handle $\mathcal{A}$'s queries. It does not enforce any assertions, and does not keep track of any additional state beyond those of the three extractors (i.e. it doesn't store D, C, L_AVC, C_AVC, D_c, D_VRF, D_OA).

- Queries to the Ideal oracle for objects related to VRF are forwarded by $\mathcal{B}$ to its own challenger, while queries for the other objects are answered by running the appropriate extractor, updating its state and returning its output.
- ExtractD oracle queries are handled by following the same steps that RZKS.Extract would follow, except that $\mathcal{B}$ replaces the call on input pk, tlbl to VRF.Extract with an Extract oracle query to its own challenger, and does not call C.Extract or compute D_{com} or check any of the subsequent assertions.
- All other oracle queries (CheckVerD, CheckVerUpdD, CheckVerUpdC, ExtractC, CheckVerExt) are handled by simply returning without performing any action.
- When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output.

Note that when $\mathcal{B}$ is executed in VRF-Sound-IDEAL, $\mathcal{A}$'s view up until the point where the game halts has the same distribution as in an execution of $Hyb1$. Indeed, all ideal $\mathsf{Extract}(\mathtt{Ideal}, \ldots)$ oracle queries (the only ones which return an output to $\mathcal{A}$) are answered by the appropriate extractors, either simulated by $\mathcal{B}$ in the C or OA case, or executed by $\mathcal{B}$'s challenger in the case of VRF. Each extractor's state also has the same distribution (since they answer the same Ideal queries, while other queries do not change the extractors' states), and so their outputs are distributed equally too. Note that it is possible that, since we are removing some assertions compared to $Hyb1$, $\mathcal{A}$ might trigger an assertion there that is not enforced in $\mathcal{B}$'s simulation, and thus the game would be halted earlier in that case. However, we only focus on those executions where these assertions aren't triggered. Conversely, VRF-Sound-IDEAL enforces an additional assertion on Extract queries which is not checked in $Hyb1$ (i.e. "**assert** $x = \bot \lor \forall y' \neq y : T[pk, y'] \neq x''$), but triggering this assertion would only increase $\mathcal{B}$'s advantage in the VRF soundness game.

Now, suppose $\mathcal{A}$ triggers the new assertion related to D_{VRF} in $Hyb1$ during an ExtractD query. Note that $\mathcal{B}$'s challenger keeps a table T which is identical to D_{VRF} by construction, and enforces the same assertions on that table as in $Hyb1$. Therefore, if $\mathcal{A}$ causes the assertion to be triggered in $Hyb1$, then VRF-Sound-IDEAL will also trigger an assertion.

$Hyb1 \approx Hyb2$. Assume by contradiction that there exists $\mathcal{A}$ that can distinguish $Hyb1$ from $Hyb2$. By construction, $\mathcal{A}$ needs to trigger one of the new assertions in $Hyb2$ with better than negligible probability. We can leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the soundness of the VRF. To do so, it is enough to construct an adversary $\mathcal{B}$ that leverages $\mathcal{A}$ such that, if $\mathcal{A}$ triggers the new assertions in $Hyb2$, then $\mathcal{B}$ also triggers an assertion in VRF-Sound-IDEAL with at least the same probability. As in the previous case, $\mathcal{B}$'s advantage in the VRF game bounds the probability that this assertion is triggered, which implies the latter must also be negligible.

$\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $Hyb2$, with the following modifications:

- $\mathcal{B}$ internally runs a copy of OA.Extract, C.Extract, AVC.Extract and uses them to handle $\mathcal{A}$'s queries. It does not enforce any assertions (beyond the ones its challenger might trigger), and only keeps as internal state a set D which keeps track of which commitments have been submitted to ExtractD. In particular, it doesn't keep track of $C, L_{\mathsf{AVC}}, C_{\mathsf{AVC}}, D_c, D_{\mathsf{VRF}}, D_{\mathsf{OA}}$.
- Queries to the Ideal oracle for objects related to VRF are forwarded by $\mathcal{B}$ to its own challenger, while queries for the other objects are answered by running the appropriate extractor, updating its state and returning its output.
- ExtractD oracle queries are handled by following the same steps that RZKS.Extract would follow, except that $\mathcal{B}$ replaces the call on input pk, tlbl to VRF.Extract with an Extract oracle query to its own challenger, and does not call C.Extract or compute D_{com} or keep track of $L_{\mathsf{AVC}}, C_{\mathsf{AVC}}, D_{\mathsf{C}}, \ldots$ (but adds com toD).
- CheckVerD oracle queries are handled by $\mathcal{B}$ as in $Hyb2$ (without the assertions), except that the VRF.Extract extractor call is substituted for an Extract oracle call to $\mathcal{B}$'s own challenger, and in addition $\mathcal{B}$ makes a CheckExtraction$(pk, \mathsf{tlbl}^*, \mathsf{label}, \pi_{\mathsf{VRF}})$ oracle query before returning.
- CheckVerUpdD oracle queries are handled as in $Hyb2$ by $\mathcal{B}$ itself (without the assertions), but in addition if $pk_{t_0} \neq pk_{t_1}$ then $\mathcal{B}$ makes an extra CheckVerRotate$(pk_{t_0}, pk_{t_1}, \{(\mathsf{tlbl}_j^{g-1}, \mathsf{tlbl}_j^g)\}_{j \in L}, \pi_{\mathsf{VRF}})$ oracle query to its own challenger.
- All other oracle queries (CheckVerUpdC, ExtractC, CheckVerExt) are handled by simply returning without performing any action.
- When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output.

Note that, as in the argument that $Hyb0 \approx Hyb1$, when $\mathcal{B}$ is executed in VRF-Sound-IDEAL, $\mathcal{A}$'s view up until the point where the game halts has the same distribution as in an execution of $Hyb2$.

Moreover, suppose $\mathcal{A}$ triggers the new assertion in $Hyb2$ for ExtractD queries, i.e. there exists a $\mathsf{tlbl}' \neq \mathsf{tlbl} : D_{\mathsf{VRF}}[pk][\mathsf{tlbl}'] = \mathsf{label}$. Whenever a tuple $(pk, \mathsf{tlbl}, \mathsf{label})$ is added to D_{VRF} in $Hyb2$, $\mathcal{B}$ makes an oracle query that causes the same record to be added to its challenger's T table. Therefore, if $\mathcal{A}$ causes the assertion to be triggered in $Hyb2$, then VRF-Sound-IDEAL will also trigger an assertion. The same reasoning can be applied to the new assertion in CheckVerD and CheckVerUpdD.

$Hyb2 \approx Hyb3$. The reasoning is similar to that of the previous hybrid, but here we leverage the security of the OA. Assume by contradiction that there exists $\mathcal{A}$ that can distinguish $Hyb2$ from $Hyb3$. By construction, $\mathcal{A}$ needs to trigger one of the new assertion in $Hyb3$ with better than negligible probability, and we can use $\mathcal{A}$ to build an adversary $\mathcal{B}$ that triggers an assertion in OA-Sound-IDEAL.

$\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $Hyb2$, with the following modifications:

– $\mathcal{B}$ internally runs a copy of VRF.Extract, C.Extract, AVC.Extract and uses them to handle $\mathcal{A}$'s queries. It does not enforce any assertions (beyond the ones its challenger might trigger), and only keeps as internal state a set D which keeps track of which commitments have been submitted to ExtractD. In particular, it doesn't keep track of $C, L_{\mathsf{AVC}}, C_{\mathsf{AVC}}, D_c, D_{\mathsf{VRF}}, D_{\mathsf{OA}}$.
– Queries to the Ideal oracle for objects related to OA are forwarded by $\mathcal{B}$ to its own challenger, while queries for the other objects are answered by running the appropriate extractor and returning its output.
– ExtractD oracle queries are handled by $\mathcal{B}$ running $L, C \leftarrow$ AVC.Extract($\mathtt{ExtrC}$, AVC.st, com), parsing $last(L)$ as $(\mathsf{com}_{\mathsf{OA}}, pk)$, making a query for $\mathsf{com}_{\mathsf{OA}}$ to its own ExtractD oracle and then returning. In particular, the VRF and C extractors are not called.
– CheckVerD oracle queries are handled by $\mathcal{B}$ by running the required Verify algorithm, parsing π as $(\pi_{\mathsf{AVC}}, \pi_{\mathsf{OA}}, \pi_{VRF}, \mathsf{tlbl}^*, \mathsf{tval}^*, \mathsf{aux}, \mathsf{com}_{\mathsf{INT}})$ and extracting $\mathsf{com}_{\mathsf{OA}}, pk$ from $\mathsf{com}_{\mathsf{INT}}$ as before, then making a CheckVerD oracle query on input $(\mathsf{com}_{\mathsf{OA}}, \mathsf{tlbl}^*, \mathsf{tval}^*, i^*, \pi_{\mathsf{OA}})$ to $\mathcal{B}$'s own oracle, and then returning its output.
– CheckVerUpdD oracle queries are handled by first running VerifyUpd, and parsing π as $(\pi', \pi_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}}^{t_1}, \mathsf{com}_{\mathsf{INT}}^{t_0}, \pi_{\mathsf{AVC}}^{t_1}, \pi_{\mathsf{AVC}}^{t_0})$, $\mathsf{com}_{\mathsf{INT}}^i$ as $(\mathsf{com}_{\mathsf{OA}}^i, pk_i)$ for $i \in \{t_0, t_1\}$. Then, if $pk_{t_0} = pk_{t_1}$, $\mathcal{B}$ makes an oracle call CheckVerUpdD($\mathsf{com}_{\mathsf{OA}}^{t_0}, \mathsf{com}_{\mathsf{OA}}^{t_1}, \pi'$). Otherwise, if $pk_{t_0} \neq pk_{t_1}$, it parses π' as $(\pi_{\mathsf{OA}}, \pi_{\mathsf{OA}}^{g-1}, \pi_{\mathsf{OA}}^g, \pi_{\mathsf{VRF}}, \mathsf{com}_{\mathsf{OA}}', \{(\mathsf{tlbl}_j^{g-1}, \mathsf{tlbl}_j^g, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L})$ and makes four oracle calls ExtractD($\mathsf{com}_{\mathsf{OA}}'$), CheckVerAll($\mathsf{com}_{\mathsf{OA}}^i, \{(\mathsf{tlbl}_j^i, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L}), \pi_{\mathsf{OA}}^i)$ for $i \in \{g-1, g\}$, and CheckVerUpdD($\mathsf{com}_{\mathsf{OA}}', \mathsf{com}_{\mathsf{OA}}^{t_1}, \pi_{\mathsf{OA}}$) to its own challenger.
– All other oracle queries (CheckVerUpdC, ExtractC, CheckVerExt) are handled by simply returning without performing any action.
– When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output.

Note that, as before, when $\mathcal{B}$ is executed in OA-Sound-IDEAL, $\mathcal{A}$'s view up until the point where the game halts has the same distribution as in an execution of $Hyb3$.

Moreover, suppose $\mathcal{A}$ triggers one of the new assertions in $Hyb3$. Note that whenever a value $\mathsf{com}_{\mathsf{OA}}$ is added to D_{OA} in $Hyb3$ (through calling the extractor as a result of on an ExtractD query), $\mathcal{B}$ makes a corresponding query to it's challenger, therefore the challenger's own D table matches D_{OA}. So when the new assertion is triggered, say during a CheckVerD query, the corresponding CheckVerD query that $\mathcal{B}$ makes to its challenger would therefore result in the same condition being checked, and thus the same assertion being thrown. In the case of a CheckVerUpdD query, the reasoning is similar.

$Hyb3 \approx Hyb4$. The proof is analogous to the ones of $Hyb1 \approx Hyb2$ and $Hyb2 \approx Hyb3$, but in this case we reduce to the soundness of the AVC. Note that, in this proof, the OA Extractor won't be called when answering ExtractD queries, similarly to how we skip the C and VRF extractors in the previous arguments. In addition, $\mathcal{B}$ will keep track of the set C of commitments that have been submitted to ExtractC queries (in addition to the set D of the previous proof).

$Hyb4 \approx Hyb5$. Assume by contradiction that there exists $\mathcal{A}$ that can distinguish $Hyb4$ from $Hyb5$. We can leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the extractability of the commitment scheme. $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of either $Hyb4$ or $Hyb5$. $\mathcal{B}$ internally runs a copy of OA.Extract, VRF.Extract, AVC.Extract and uses them to handle $\mathcal{A}$'s Ideal queries. $\mathcal{B}$ answers $\mathcal{A}$'s queries as follows:

– Queries to the Ideal oracle for objects related to C are forwarded by $\mathcal{B}$ to its own challenger, while queries for the other objects are answered by running the appropriate extractor and returning its output.
– ExtractD oracle queries are handled by following the same steps that $Hyb4$ would follow, except that $\mathcal{B}$ replaces the call to C.Extract with an Extract oracle query to its own challenger, and (if the call returns with no output, implying that the game is not aborted by an exception thrown by the C game) setting $D_{\mathsf{com}}[\mathsf{label}] \leftarrow (\mathsf{tval}, t)$ (as opposed to $D_{\mathsf{com}}[\mathsf{label}] \leftarrow (\mathsf{val}, t)$ where val is the output of the C extractor which $\mathcal{B}$ does not have access to). $\mathcal{B}$ also does not keep track of the D_{C} table.
– CheckVerD oracle queries are handled by $\mathcal{B}$ as in $Hyb4$, except that in the case where $\mathsf{val}^* \neq \bot$ and $i^* \neq \bot$, $\mathcal{B}$ replaces the assertion $D[\mathsf{com}][\mathsf{label}] = (\mathsf{val}^*, i^*)$ with $D[\mathsf{com}][\mathsf{label}] = (\mathsf{tval}^*, i^*)$, where tval^* is the value from the proof provided by $\mathcal{A}$. In addition, $\mathcal{B}$ makes a CheckVer$(\mathsf{tval}^*, \mathsf{val}^*, \mathsf{aux})$ query to its own oracle before returning.
– CheckVerUpdD oracle queries are handled by $\mathcal{B}$ itself simulating $Hyb4$. We stress that the tuples in D as checked by $\mathcal{B}$ are of the form $(\mathsf{label}, \mathsf{tval}, t)$ instead of $(\mathsf{label}, \mathsf{val}, t)$.
– When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output. If $\mathcal{B}$ detects that any of the assertions it checks would fail, it aborts the game with the same error.

When $\mathcal{B}$ is executed in C-Sound-IDEAL, $\mathcal{A}$'s view up until the point where the game halts has the same distribution as in an execution of $Hyb4$ as in the previous cases.

Moreover, $\mathcal{A}$ triggers an assertion in $Hyb4$ if and only if an assertion is also triggered in C-Sound-IDEAL:

– The conditions asserted during an ExtractD query in $Hyb4$ are:
  1. $\forall\, (\mathsf{label}, \mathsf{val}, i) \in D[\mathsf{com}] : 0 < i \leq \mathbf{t}(\mathsf{com})$.
     This assertion is never triggered since by construction the extractor's output always satisfies it.
  2. If $\mathsf{com} \in D$ **assert** $D[\mathsf{com}] = D_{\mathsf{com}}$, and similar statements for $D_{\mathsf{VRF}}, C_{\mathsf{AVC}}, L_{\mathsf{AVC}}, D_{\mathsf{OA}}, D_{\mathsf{C}}$.
     Assertions on $D_{\mathsf{VRF}}, D_{\mathsf{OA}}, D_{\mathsf{AVC}}$ are checked by $\mathcal{B}$ directly during its simulation. While $\mathcal{B}$ does not keep track of D_{C} directly, by construction $\mathcal{B}$'s challenger keeps an equivalent table in its state and would test for exactly the same conditions. $\mathcal{B}$ also checks $D[\mathsf{com}] = D_{\mathsf{com}}$, but in the simulated game D maps digests and labels to commitments and epochs (rather than digests and labels to values and epochs as in $Hyb4$). This is not a problem because, as if this assertion fails one of the other assertions we discussed would have failed first.
  3. For every tuple $(\mathsf{tlbl}, \mathsf{label})$ added to $D_{\mathsf{VRF}}[pk]$ by the extractor, that there is no other $\mathsf{tlbl}' \neq \mathsf{tlbl} : D_{\mathsf{VRF}}[pk][\mathsf{tlbl}'] = \mathsf{label}$.
     This assertion is also checked by $\mathcal{B}$ in its simulation.
– The conditions asserted during CheckVerD oracle calls in $Hyb4$ are:
  1. $last(L_{\mathsf{AVC}}[\mathsf{com}]) = \mathsf{com_{INT}}$
  2. If $\mathsf{tval}^* = \bot$ or $i^* = \bot$, then $\mathsf{tlbl}^* \notin D_{\mathsf{OA}}[\mathsf{com_{OA}}]$, and otherwise
     $D_{\mathsf{OA}}[\mathsf{com_{OA}}][\mathsf{tlbl}^*] = (\mathsf{tval}^*, i^*)$
  3. $D_{\mathsf{VRF}}[pk][\mathsf{tlbl}^*] = \mathsf{label}$
  4. That there is no other $\mathsf{tlbl}' \neq \mathsf{tlbl}^* : D_{\mathsf{VRF}}[pk][\mathsf{tlbl}'] = \mathsf{label}$
  5. If $\mathsf{val}^* = \bot$ or $i^* = \bot$ then $\mathsf{label} \notin D[\mathsf{com}]$, else $D[\mathsf{com}][\mathsf{label}] = (\mathsf{val}^*, i^*)$
  $\mathcal{B}$ checks the first four conditions in the simulation as well, so those assertions are triggered analogously. Assume the first four conditions aren't triggered in $Hyb4$ or C-Sound-IDEAL, we claim that condition 5 is not triggered either. For condition 5, assume by contradiction that $\mathsf{val}^* = \bot$ or $i^* = \bot$ but $\mathsf{label} \in D[\mathsf{com}]$. Since verification succeeds, it must be $\mathsf{tval}^* = i^* = \bot$. Let $(\mathsf{tlbl}', \mathsf{tval}', i')$ be the triple considered by the extractor when label was added to D_{com}. Since the first four assertions aren't triggered, it must be that $\mathsf{tlbl}' = \mathsf{tlbl}^*$. But since $i^* = \bot$, due to the second assertion it must be that $\mathsf{tlbl}' = \mathsf{tlbl}^* \notin D_{\mathsf{OA}}[\mathsf{com_{OA}}]$, but this is not possible since it contradicts label being added to D_{com} when $(\mathsf{tlbl}', \mathsf{tval}', i)$ is considered. In the case where $\mathsf{val}^* \neq \bot$ and $i^* \neq \bot$ but $\mathsf{label} \notin D[\mathsf{com}]$, then since the first three assertions aren't triggered it must be $last(L_{\mathsf{AVC}}[\mathsf{com}]) = \mathsf{com_{INT}}$ (where $\mathsf{com_{INT}} = (\mathsf{com_{OA}}, pk)$), $D_{\mathsf{VRF}}[pk][\mathsf{tlbl}^*] = \mathsf{label}$, and $D_{\mathsf{OA}}[\mathsf{com_{OA}}][\mathsf{tlbl}^*] = (\mathsf{tval}^*, i^*)$. Therefore the only reason for $\mathsf{label} \notin D[\mathsf{com}]$ is if C.Extract returned $\bot$, but in this case $\mathcal{B}$'s challenger would throw an assertion. A similar reasoning would apply in the case where $\mathsf{val}^* \neq \bot$ and $i^* \neq \bot$, $\mathsf{label} \in D[\mathsf{com}]$ but $D[\mathsf{com}][\mathsf{label}] \neq (\mathsf{val}^*, i^*)$.
  Conversely, one can check that if $\mathcal{A}$ triggers an assertion during a simulated CheckVerD oracle call in C-Sound-IDEAL, an assertion is also raised in $Hyb3$.

– If an assertion is triggered in CheckVerUpdD in $Hyb4$, it must be due to one of the following being violated:

1. $last(L_{\mathsf{AVC}}[\mathsf{com}^a]) = \mathsf{com}_{\mathsf{INT}}^{t_0}$ and $L_{\mathsf{AVC}}[\mathsf{com}^b] = L_{\mathsf{AVC}}[\mathsf{com}^a] || \mathsf{com}_{\mathsf{INT}}^{t_1}$ (introduced in $Hyb4$)
2. All assertions for CheckVerUpdD introduced in $Hyb3$: if $pk_{t_0} = pk_{t_1}$, the game asserts
   - $D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_0}] \subseteq D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_1}]$
   - $\forall (\mathsf{tlbl}, \mathsf{tval}, t) \in D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_1}] \setminus D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_0}] : t = \mathsf{RZKS.t}(\mathsf{com}^b)$
   - either $\mathsf{RZKS.t}(\mathsf{com}^a) \neq 0$ or $D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_0}] = \{\}$.

   Otherwise, if $pk_{t_0} \neq pk_{t_1}$, the game asserts
   - $|D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_0}]| = |D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}']| = |L|$ and $\forall j \in L, D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_0}][\mathsf{tlbl}_j^{g-1}] = D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}'][\mathsf{tlbl}_j^g] = (\mathsf{tval}_j, \mathsf{epno}_j)$
   - $D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}'] \subseteq D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_1}]$
   - $\forall (\mathsf{tlbl}, \mathsf{tval}, t) \in D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_1}] \setminus D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}'] : t = \mathbf{t}(\mathsf{com}^b)$, and that either $\mathbf{t}(\mathsf{com}_{\mathsf{OA}}') \neq 0$ or $D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}'] = \{\}$.
3. $\mathbf{t}(\mathsf{com}^b) = \mathbf{t}(\mathsf{com}^a) + 1$
4. $D[\mathsf{com}^a] \subseteq D[\mathsf{com}^b]$
5. $\forall (\mathsf{label}, \mathsf{val}, t) \in D[\mathsf{com}^b] \setminus D[com^a] : t = \mathbf{t}(\mathsf{com}^b)$
6. $(\mathbf{t}(\mathsf{com}^a) \neq 0$ or $D[\mathsf{com}^a] = \{\})$

The first three bullet points are also checked by $\mathcal{B}$ in C-Sound-IDEAL, so the assertions would also be triggered in the simulation. Assuming that those are not triggered, let's prove that condition 4 won't be triggered either. Let $(\mathsf{label}, \mathsf{val}, t) \in D[\mathsf{com}^a]$. Consider the $(\mathsf{tlbl}, \mathsf{tval}, t)$ processed by the extractor when adding label to $D[\mathsf{com}^a]$: that tuple is by construction part of $D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_0}]$. Let's assume that $pk_{t_0} = pk_{t_1}$ (the other case is similar). Because we assumed assertion 2 isn't triggered, it must also be $(\mathsf{tlbl}, \mathsf{tval}, t) \in D_{\mathsf{OA}}[\mathsf{com}_{\mathsf{OA}}^{t_1}]$, and therefore $(\mathsf{label}, \mathsf{val}, t) \in D[\mathsf{com}^b]$ (as, because of the assertions introduced in $Hyb1$, repeated queries to the extractor on the same input will receive the same output), which proves that the assertion is true. The other assertions can be analyzed analogously.

– Other oracle queries by $\mathcal{A}$ can be perfectly simulated by $\mathcal{B}$ as in $Hyb4$, as they do not depend on the output of the C Extractor.

In short we have

$$\Pr[Hyb4(\mathcal{A}) = 1] = \Pr[\mathsf{C\text{-}Sound\text{-}IDEAL}(\mathcal{B}) = 1].$$

Analogously as in the previous hybrids, we can prove that when $\mathcal{B}$ is executed in C-Sound-REAL, $\mathcal{A}$'s view up until the point where the game is halted has the same distribution as in an execution of $Hyb5$. Moreover, by inspection (analogously as above) one can check that whenever $\mathcal{A}$ triggers an assertion in $Hyb5$, the same assertion is also triggered in C-Sound-REAL($\mathcal{B}$) and vice versa. Therefore

$$\Pr[Hyb5(\mathcal{A}) = 1] = \Pr[\mathsf{C\text{-}Sound\text{-}REAL}(\mathcal{B}) = 1].$$

This implies that $\mathcal{A}$'s advantage in distinguishing the two hybrids can be bounded by $\mathcal{B}$'s advantage in the commitment soundness game, which is negligible.

$Hyb5 \approx Hyb6$. The proof is analogous to the one above, but here we reduce to the soundness of the VRF. The adversary $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of either $Hyb5$ or $Hyb6$. $\mathcal{B}$ internally runs a copy of OA.Extract, AVC.Extract and a real copy of C's Ideal object, and uses them to handle $\mathcal{A}$'s Ideal queries. $\mathcal{B}$ answers $\mathcal{A}$'s queries as follows:

– Queries to the Ideal oracle for objects related to VRF are forwarded by $\mathcal{B}$ to its own challenger; queries for OA and AVC objects are answered using OA.Extract, and queries for C objects using a real implementation of such objects.

– ExtractD oracle queries are handled by following the same steps that $Hyb5$ would follow, except that $\mathcal{B}$ replaces the call to VRF.Extract with an Extract oracle query to its own challenger, and (if the call returns with no output, implying that the game is not aborted by an exception thrown by the VRF game) setting $D_{\mathsf{com}}[\mathsf{tlbl}] \leftarrow (\mathsf{tval}, t)$ (as opposed to $D_{\mathsf{com}}[\mathsf{label}] \leftarrow (\mathsf{tval}, t)$ where label is the output of the VRF extractor which $\mathcal{B}$ does not have access to). $\mathcal{B}$ also does not keep track of the D_{VRF} table. Note that the C.Extract is not used in this game.

- CheckVerD oracle queries are handled by $\mathcal{B}$ as in $Hyb5$, except that in the case where $\text{val}^* \neq \perp$ and $i^* \neq \perp$, $\mathcal{B}$ replaces the assertion $D[\text{com}][\text{label}] = (\text{tval}^*, i^*)$ with $D[\text{com}][\text{tlbl}^*] = (\text{tval}^*, i^*)$, where tlbl^* is the value from the proof provided by $\mathcal{A}$. In addition, $\mathcal{B}$ makes a $\text{CheckExtraction}(pk, \text{tlbl}^*, \text{label}, \text{aux})$ query to its own oracle before returning.
- CheckVerUpdD oracle queries are handled by $\mathcal{B}$ itself simulating $Hyb5$. We stress that the tuples in D as checked by $\mathcal{B}$ are of the form $(\text{tlbl}, \text{tval}, t)$ instead of $(\text{label}, \text{val}, t)$.
- All other queries are handled by $\mathcal{B}$ by simulating $Hyb5$.
- When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output. If $\mathcal{B}$ detects that any of the assertions it checks would fail, it aborts the game with the same error.

A similar analysis as the previous case shows that $\mathcal{A}$'s advantage in distinguishing $Hyb5$ and $Hyb6$ can be bounded by $\mathcal{B}$'s advantage in the VRF game.

$Hyb6 \approx Hyb7$. An adversary $\mathcal{A}$ who distinguishes $Hyb6$ from $Hyb7$ can be used to break the soundness of the OA. The adversary $\mathcal{B}$ in this case runs the real version of C and VRF's ideal objects, but leverages AVC.Extract. It handles all queries similarly to the adversary $\mathcal{B}$ we described above to argue that $Hyb2 \approx Hyb3$, except that here Ideal oracle for objects related to C and VRF objects using a real implementation of such objects (OA related objects are still handled by relaying the query to $\mathcal{B}$'s challenger, and the AVC objects by calling the AVC extractor).

The argument that $\mathcal{A}$'s advantage in distinguishing $Hyb6$ and $Hyb7$ can be bounded by $\mathcal{B}$'s advantage in the OA game is analogous to the previous cases.

$Hyb7 \approx Hyb8$. An adversary $\mathcal{A}$ who distinguishes $Hyb7$ from $Hyb8$ can be used to break the soundness of the AVC. The adversary $\mathcal{B}$ in this case runs the real version of C, VRF and OA's ideal objects and works as follows:

- Queries to the Ideal oracle for objects related to AVC are forwarded by $\mathcal{B}$ to its own challenger; queries for other objects are answered using a real implementation of such objects.
- ExtractD and ExtractC oracle queries are handled by making an Extract call to $\mathcal{B}$'s own oracle (if the call returns with no output, implying that the game is not aborted by an exception thrown by the AVC game) simply returning.
- ChackVerD oracle queries are handled by $\mathcal{B}$ by running the verification algorithm and making a $\text{CheckVer}(\text{com}, \text{AVC.t}(\text{com}), \text{com}_{\text{INT}}, \pi_{\text{AVC}})$ query to its own oracle (the last two values obtained by parsing π) and simply returning.
- CheckVerUpdD oracle queries are handled by $\mathcal{B}$ itself by running the verification algorithm and making three oracle queries $\text{CheckVerExt}(\text{com}^a, \text{com}^b, \pi_{\text{AVC}})$, $\text{CheckVer}(\text{com}^a, \text{AVC.t}(\text{com}^a), \text{com}_{\text{INT}}^{t_0}, \pi_{\text{AVC}}^{t_0})$, and $\text{CheckVer}(\text{com}^b, \text{AVC.t}(\text{com}^b), \text{com}_{\text{INT}}^{t_1}, \pi_{\text{AVC}}^{t_1})$ to its own oracle and then returning.
- CheckVerUpdC and CheckVerExt queries are handled by running the verification algorithm and making a $\text{CheckVerExt}(\text{com}^a, \text{com}^b, \pi_{\text{AVC}})$ oracle query.
- When $\mathcal{A}$ halts, $\mathcal{B}$ halts with the same output. If $\mathcal{B}$ detects that any of the assertions it checks would fail, it aborts the game with the same error.

The argument that $\mathcal{A}$'s advantage in distinguishing $Hyb7$ and $Hyb8$ can be bounded by $\mathcal{B}$'s advantage in the OA game is analogous to the previous cases.
QED.

G Proof of Theorem 5

Proof. We want to show that there exists some simulator $\mathcal{S}$ such that for any efficient adversary $\mathcal{A}$,

$$|\Pr[\text{RZKS-ZK-REAL}(\mathcal{A}) \to 1] - \Pr[\text{RZKS-ZK-IDEAL}(\mathcal{A}) \to 1]|$$

is negligible in the security parameter. We will show this by a series of hybrids, where the first hybrid will be RZKS-ZK-REAL and the last hybrid will be RZKS-ZK-IDEAL. In the last hybrid, we will explicitly define the simulator $\mathcal{S}$. We will denote the range of VRF as Y_{VRF}.

- $Hyb0$: Defined as RZKS-ZK-REAL.

– $Hyb1$: Defined as the previous hybrid, but we replace the commitments computed by the server (and their openings) with simulated ones. That is, if $\mathcal{S}_\mathsf{C}$ is the simulator whose existence is guaranteed by the fact that C satisfies simulatability game, then
  - Queries to ideal objects related to C are replaced by the versions simulated by $\mathcal{S}_\mathsf{C}$.
  - In RZKS.GenPP, we replace C.GenPP(1^λ) with $\mathcal{S}_\mathsf{C}(\texttt{Init})$.
  - In RZKS.Update and RZKS.PCSUpdate, we replace $\mathsf{tval}_i, \mathsf{aux}_i \leftarrow$ C.Commit(val_i) with $\mathsf{tval}_i \leftarrow \mathcal{S}_\mathsf{C}(\texttt{Commit})$ and setting $\mathsf{aux}_i \leftarrow \perp$ in the state. We will replace it with a simulated value just before returning it to the adversary.
  - In RZKS.Query, after assigning $(\mathsf{val}, \mathsf{epno}_\mathsf{label}, \mathsf{tval}, \mathsf{aux}) \leftarrow D[\mathsf{label}]$, we replace aux (which was set to $\perp$ during Update or PCSUpdate) by running $\mathsf{val}, \mathsf{aux}' \leftarrow \mathcal{S}_\mathsf{C}(\texttt{Open}, \mathsf{tval}, \mathsf{val})$ and setting $\mathsf{aux} \leftarrow \mathsf{aux}'$.
  - During oracle calls to LeakState, we modify the table D which is part of the state by replacing values of aux with the output of $\mathcal{S}_\mathsf{C}(\texttt{Open}, \cdot, \cdot)$ analogously as in Query calls.

Lemma 20. $|\Pr[Hyb1 \to 1] - \Pr[Hyb0 \to 1]| \le negl(\lambda)$.

Proof: Assume by contradiction that there exists $\mathcal{A}$ that can distinguish $Hyb0$ from $Hyb1$. We will leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the simulatability of the commitment scheme. $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $Hyb0$ with the following changes:
  - Queries to the Ideal oracle for objects related to C are forwarded by $\mathcal{B}$ to its own challenger.
  - In RZKS.GenPP, instead of running C.GenPP(1^λ), $\mathcal{B}$ uses the pp returned by its own challenger.
  - In RZKS.Update and RZKS.PCSUpdate, we replace C.Commit with a Commit call to $\mathcal{B}$'s challenger.
  - In RZKS.Query, after assigning $(\mathsf{val}, \mathsf{epno}_\mathsf{label}, \mathsf{tval}, \mathsf{aux}) \leftarrow D[\mathsf{label}]$, we replace aux by running $\mathsf{val}', \mathsf{aux}' \leftarrow$ Open(tval) and setting $\mathsf{aux} \leftarrow \mathsf{aux}'$.
  - When LeakState is called, for each $\mathsf{label} \in D$, we replace aux in $(\mathsf{val}, \mathsf{epno}_\mathsf{label}, \mathsf{tval}, \mathsf{aux}) \leftarrow D[\mathsf{label}]$ analogously to RZKS.Query.

It is easy to see that $\mathcal{B}$ simulates $Hyb0$ for $\mathcal{A}$ when its challenger is the real game, and $\mathcal{B}$ simulates $Hyb1$ if its challenger is the ideal game.

– $Hyb2$: Defined as the previous hybrid, but we replace the VRF outputs with random values (for non-compromised keys) or simulated ones (for compromised keys), and all proofs with simulated ones. That is, if $\mathcal{S}_\mathsf{VRF}$ is the simulator from the zero-knowledge game for VRF, then
  - Queries to ideal objects related to VRF are answered by $\mathcal{S}_\mathsf{VRF}$.
  - At the beginning of the game, in RZKS.GenPP, we replace $\mathsf{pp}_\mathsf{VRF} \leftarrow$ VRF.GenPP(1^λ) with the value obtained by $\mathsf{pp}_\mathsf{VRF}, pk_0 \leftarrow \mathcal{S}_\mathsf{VRF}(\texttt{Init})$. pk_0 is used in the subsequent RZKS.Init call (instead of calling VRF.KeyGen). Note that sk_0 is not defined in the modified RZKS.Init, and thus we replace $\mathsf{K}_\mathsf{VRF}[\mathsf{g}] \leftarrow (sk_0, pk_0)$ with $\mathsf{K}_\mathsf{VRF}[\mathsf{g}] \leftarrow (\perp, pk_0)$. LeakState fills in all secret key values right before returning the state to the adversary. This hybrid also initializes $\mathsf{g}_{cor} \leftarrow -1$ and a table D_VRF that maps pairs $(\mathsf{g}, \mathsf{label})$ of a generation and RZKS label to the corresponding (simulated) VRF output.
  - Define the following helper function Eval$'(\mathsf{g}, \mathsf{label}_i)$:
    * If $\mathsf{g} \le \mathsf{g}_{cor}$, **return** $\mathcal{S}_\mathsf{VRF}(\texttt{Corrupted-Eval}, \mathsf{g}, \mathsf{label}_i)$
    * If $\mathsf{label}_i \notin D_\mathsf{VRF}[\mathsf{g}]$: $D_\mathsf{VRF}[\mathsf{g}][\mathsf{label}_i] \overset{\$}{\leftarrow} Y_\mathsf{VRF}$
    * **return** $D_\mathsf{VRF}[\mathsf{g}][\mathsf{label}_i]$
  - In RZKS.Query, we replace $(\pi_\mathsf{VRF}, \mathsf{tlbl}) \leftarrow$ VRF.Query$(\mathsf{pp}, \mathsf{K}_\mathsf{VRF}[G[u]].sk, \mathsf{label})$ with:
    * $\mathsf{tlbl} \leftarrow$ Eval$'(G[u], \mathsf{label})$;
    * $\pi_\mathsf{VRF} \leftarrow \mathcal{S}_\mathsf{VRF}(\texttt{Explain}, G[u], \mathsf{label}, \mathsf{tlbl})$.
  - In RZKS.Update and RZKS.PCSUpdate, we replace VRF.Eval$(\mathsf{K}_\mathsf{VRF}[\mathsf{g}].sk, \mathsf{label}_i)$ with Eval$'(\mathsf{g}, \mathsf{label}_i)$.
  - In RZKS.PCSUpdate, we replace VRF.Rotate$(\mathsf{K}_\mathsf{VRF}[\mathsf{g}], L)$ as well as the setting of K_VRF with the following:
    * For each $j \in L$, set $(\mathsf{tlbl}_j^\mathsf{g}, \mathsf{tlbl}_j^{\mathsf{g}+1}) \leftarrow (\mathsf{Eval}'(\mathsf{g}, j), \mathsf{Eval}'(\mathsf{g}+1, j))$.
    * $pk_{\mathsf{g}+1}, \pi_\mathsf{VRF} \leftarrow \mathcal{S}_\mathsf{VRF}(\texttt{Rotate}, \{(\mathsf{tlbl}_j^\mathsf{g}, \mathsf{tlbl}_j^{\mathsf{g}+1})\}_{j \in L})$.
    * $\mathsf{K}_\mathsf{VRF}[\mathsf{g}+1] \leftarrow (\perp, pk_{\mathsf{g}+1}), \mathsf{g} \leftarrow \mathsf{g}+1$.
  - ProveExt queries are handled as in the previous hybrid.
  - When LeakState is called, perform the following:

* $sk_{\mathbf{g}_{cor}+1}, \ldots, sk_{\mathbf{g}} \leftarrow \mathcal{S}_{\mathsf{VRF}}(\texttt{Corrupt}, D_{\mathsf{VRF}})$.
* For $j = \mathbf{g}_{cor} + 1, \ldots, \mathbf{g}$, set sk_j in $\mathsf{K}_{\mathsf{VRF}}[j] \leftarrow (sk_j, pk_j)$, keeping the pk_j value the same.
* $\mathbf{g}_{cor} \leftarrow \mathbf{g}$.

Lemma 21. $|\Pr[Hyb2 \to 1] - \Pr[Hyb1 \to 1]| \leq negl(\lambda)$.

Proof: Assume by contradiction that there exists $\mathcal{A}$ that can distinguish $Hyb2$ from $Hyb1$. We will leverage $\mathcal{A}$ to build an adversary $\mathcal{B}$ that breaks the simulatability of the VRF scheme. $\mathcal{B}$ runs $\mathcal{A}$, simulating for it an execution of $Hyb2$ with the following changes:

- $\mathcal{B}$ receives as input $\mathsf{pp}_{\mathsf{VRF}}, pk_0$ from its challenger, and uses these values instead of running $\mathcal{S}_{\mathsf{VRF}}(\texttt{Init})$ as defined in $Hyb2$.
- Queries to the Ideal oracle for objects related to VRF are forwarded by $\mathcal{B}$ to its own challenger.
- We replace all calls to Eval' in Update and PCSUpdate oracle queries with Eval calls to $\mathcal{B}$'s challenger. In Query oracle calls, $(\pi_{\mathsf{VRF}}, \mathsf{tlbl})$ are computed by making a Prove query to $\mathcal{B}$'s challenger.
- In RZKS.PCSUpdate, we replace the new algorithm for rotation with the following:
  * $pk_{\mathbf{g}+1}, \pi_{\mathsf{VRF}} \leftarrow \mathsf{Rotate}(L)$.
  * $\mathsf{K}_{\mathsf{VRF}}[\mathbf{g}+1] \leftarrow pk_{\mathbf{g}+1}, \mathbf{g} \leftarrow \mathbf{g} + 1$.
  where $\mathsf{Rotate}(L)$ is a call to $\mathcal{B}$'s challenger.
- In LeakState(), we replace the call to $\mathcal{S}(\texttt{Corrupt}, D_{\mathsf{VRF}})$ with a Corrupt call to $\mathcal{B}$'s challenger.

It is easy to see that $\mathcal{B}$ perfectly simulates $Hyb2$ for $\mathcal{A}$ when executed in VRF-ZK-IDEAL, and $Hyb1$ when executed in VRF-ZK-REAL: the only salient difference in the latter is that the VRF secret keys are stored by $\mathcal{B}$'s challenger until LeakState is called, and the state is patched just in time before $\mathcal{A}$ gets to see it. Therefore $\mathcal{A}$'s advantage in distinguishing the two hybrids is bounded by $\mathcal{B}$'s advantage in the breaking the VRF Zero Knowledge property, which we assume to be negligible.

– $Hyb3$: Defined as RZKS-ZK-IDEAL, with the following simulator $\mathcal{S}$:
  ▷ $\mathcal{S}$'s state includes the following:
  - States for $\mathcal{S}_{\mathsf{C}}$ and $\mathcal{S}_{\mathsf{VRF}}$.
  - A simulated RZKS state $\mathsf{st} = (\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$.
  - A separate datastore D' to store entries inserted into the data structure for which the labels have not been revealed to the simulator. D' will be indexed by Y_{VRF}.
  - A map P which takes any VRF output to its rotation in a specified generation.
  ▷ $\mathcal{S}(\texttt{Init})$:
  - $\mathsf{pp}_{\mathsf{VRF}}, pk_0 \leftarrow \mathcal{S}_{\mathsf{VRF}}(\texttt{Init})$
  - $\mathsf{pp}_{\mathsf{C}} \leftarrow \mathcal{S}_{\mathsf{C}}(\texttt{Init})$
  - $\mathsf{pp}_{\mathsf{OA}} \leftarrow \mathsf{OA.GenPP}(1^\lambda)$
  - $\mathsf{pp}_{\mathsf{AVC}} \leftarrow \mathsf{AVC.GenPP}(1^\lambda)$
  - $\mathsf{pp} \leftarrow (\mathsf{pp}_{\mathsf{VRF}}, \mathsf{pp}_{\mathsf{OA}}, \mathsf{pp}_{\mathsf{C}}, \mathsf{pp}_{\mathsf{AVC}})$
  - $\mathsf{epno} \leftarrow 0, \mathsf{g} \leftarrow 0, \mathsf{K}_{\mathsf{VRF}} \leftarrow \{\}, \mathsf{D} \leftarrow \{\},$
    $\mathsf{st}_{\mathsf{OA}} \leftarrow \{\}, \mathsf{G} \leftarrow \{\}$
  - $P \leftarrow \{\}, \mathsf{D}' \leftarrow \{\}$
  - $\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}] \leftarrow (\bot, pk_0), \mathsf{G}[\mathsf{epno}] \leftarrow \mathsf{g}$
  - $(\mathsf{st}', \mathsf{com}^0_{\mathsf{OA}}) \leftarrow \mathsf{OA.Init}(\mathsf{pp}_{\mathsf{OA}}); \mathsf{com}^0_{\mathsf{INT}} \leftarrow (\mathsf{com}^0_{\mathsf{OA}}, pk_0); \mathsf{st}_{\mathsf{OA}}[\mathsf{g}] \leftarrow \mathsf{st}'$
  - $(\mathsf{st}_{\mathsf{AVC}}, \_) \leftarrow \mathsf{AVC.Init}(\mathsf{pp}_{\mathsf{AVC}})$;
  - $\mathsf{com}^1, \mathsf{st}_{\mathsf{AVC}}, \pi^0 \leftarrow \mathsf{AVC.Update}(\mathsf{st}_{\mathsf{AVC}}, \mathsf{com}^0_{\mathsf{INT}})$
  - $\mathsf{st} \leftarrow (\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}^1, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
  - **return** $\mathsf{com}^1, \mathsf{st}$
  ▷ $\mathcal{S}((\texttt{PCS})\texttt{Update}, S_{leak})$:   // bullets with □ only apply to PCSUpdate
  - **parse** S_{leak} as $s, \mathsf{label}_1, \ldots, \mathsf{label}_n$. (If $leaked = true$ and we are doing Update, then $s = 0$. If we are doing PCSUpdate, then $n = 0$.)
  - **parse** st as $(\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$; set $\mathsf{epno} \leftarrow \mathsf{epno} + 1$.
  - □ $L \leftarrow \{\mathsf{label} \mid (\mathsf{label}, (\ldots)) \in \mathsf{D}\}$.
  - □ $L' \leftarrow \{\mathsf{tlbl} \mid (\mathsf{tlbl}, (\ldots)) \in \mathsf{D}'\}$.
  - □ For each $j \in L$, set $(\mathsf{tlbl}^{\mathbf{g}}_j, \mathsf{tlbl}^{\mathbf{g}+1}_j) \leftarrow (\mathsf{Eval}'(\mathsf{g}, j), \mathsf{Eval}'(\mathsf{g}+1, j))$, where Eval' is defined as in $Hyb2$.
  - □ For each $j' \in L'$, set $P[j'][\mathbf{g}+1] \overset{\$}{\leftarrow} Y_{\mathsf{VRF}}$.

- □ $R \leftarrow \{(\mathsf{tlbl}_j^{\mathsf{g}}, \mathsf{tlbl}_j^{\mathsf{g}+1})\}_{j \in L} \cup \{P[j'][\mathsf{g}], P[j'][\mathsf{g}+1]\}_{j' \in L'}$.
- □ $pk_{\mathsf{g}+1}, \pi_{\mathsf{VRF}} \leftarrow \mathcal{S}_{\mathsf{VRF}}(\texttt{Rotate}, R)$.
- □ $\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}+1] \leftarrow pk_{\mathsf{g}+1}, \mathsf{g} \leftarrow \mathsf{g}+1$.
- □ $\pi_{\mathsf{OA}}^{\mathsf{g}-1} \leftarrow \mathsf{OA.ProveAll}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g}-1], \mathsf{epno}-1)$
- □ $\mathsf{st}', \_ \leftarrow \mathsf{OA.Init}(\mathsf{pp}_{\mathsf{OA}})$;
     for $i \in [\mathsf{epno}-1]$:
         $S_{\mathsf{OA}} \leftarrow \{(\mathsf{tlbl}_j^{\mathsf{g}}, \mathsf{tval}_j) \mid (j, (i, \mathsf{tval}_j, \cdot)) \in \mathsf{D}\} \cup \{(P[\mathsf{tlbl}][\mathsf{g}], \mathsf{tval}_{\mathsf{tlbl}}) \mid$
             $(\mathsf{tlbl}, (i, \mathsf{tval}_{\mathsf{tlbl}}, \cdot)) \in \mathsf{D}'\}$
         $\mathsf{st}', \mathsf{com}', \_ \leftarrow \mathsf{OA.Update}(\mathsf{st}', S_{\mathsf{OA}})$
     $\mathsf{st}_{\mathsf{OA}}[\mathsf{g}] \leftarrow \mathsf{st}', \mathsf{com}_{\mathsf{OA}}^{\mathsf{epno}-1} \leftarrow \mathsf{com}'$
- □ $\pi_{\mathsf{OA}}^2 \leftarrow \mathsf{OA.ProveAll}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], \mathsf{epno}-1)$
- • $S_{\mathsf{OA}} \leftarrow \{\}$
- • For each $i = 1, \ldots, n$:
    * $\mathsf{tlbl}_i \leftarrow \mathsf{Eval}'(\mathsf{g}, \mathsf{label}_i)$
    * $\mathsf{tval}_i \leftarrow \mathcal{S}_{\mathsf{C}}(\texttt{Commit})$
    * $S_{\mathsf{OA}} \leftarrow S_{\mathsf{OA}} \cup \{(\mathsf{tlbl}_i, \mathsf{tval}_i)\}$
    * $\mathsf{D} \leftarrow \mathsf{D} \cup \{(\mathsf{label}_i, (\bot, \mathsf{epno}, \mathsf{tval}_i, \bot))\}$
- • For each $i = 1, \ldots, s-n$:
    * $\mathsf{tlbl}_i \xleftarrow{\$} Y_{\mathsf{VRF}}$
    * $P[\mathsf{tlbl}_i][\mathsf{g}] \leftarrow \mathsf{tlbl}_i$
    * $\mathsf{tval}_i \leftarrow \mathcal{S}_{\mathsf{C}}(\texttt{Commit})$
    * $S_{\mathsf{OA}} \leftarrow S_{\mathsf{OA}} \cup \{(\mathsf{tlbl}_i, \mathsf{tval}_i)\}$
    * $\mathsf{D}' \leftarrow \mathsf{D}' \cup \{(\mathsf{tlbl}_i, (\mathsf{epno}, \mathsf{tval}_i))\}$
- • $\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], \mathsf{com}_{\mathsf{epno}}^{\mathsf{OA}}, \pi_{\mathsf{OA}} \leftarrow \mathsf{OA.Update}(st_{\mathsf{OA}}[\mathsf{g}], S_{\mathsf{OA}})$;
- • Continue simulating the Update or PCSUpdate algorithm as in Fig 8, starting from right after the call to OA.Update (keeping the final RZKS state internal, and returning the commitment and proof).
▷ $\mathcal{S}(\texttt{Query}, (\mathsf{label}, \mathsf{val}, t, u))$:
- • **parse** st as $(\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- • **require** $u \leq \mathsf{epno}$
- • If $\mathsf{label} \notin \mathsf{D}$ and $t \neq \bot$:
    * Choose and remove from D' a record $(\mathsf{tlbl}, (t', \mathsf{tval}))$ such that $t = t'$
    * $\mathsf{D} \leftarrow \mathsf{D} \cup \{(\mathsf{label}, (\bot, t, \mathsf{tval}, \bot))\}$
    * For $i = \mathsf{G}[t], \ldots, \mathsf{g}$:
        · $\mathsf{D}_{\mathsf{VRF}}[i][\mathsf{label}] = P[\mathsf{tlbl}][i]$
- • $\mathsf{tlbl} \leftarrow \mathsf{Eval}'(\mathsf{G}[u], \mathsf{label})$
- • $\pi_{\mathsf{VRF}} \leftarrow \mathcal{S}_{\mathsf{VRF}}(\texttt{Explain}, \mathsf{G}[u], \mathsf{label}, \mathsf{tlbl})$
- • If $\mathsf{label} \in \mathsf{D}$ and $t \leq u$:
        $(\_, \mathsf{epno}_{\mathsf{label}}, \mathsf{tval}, \_) \leftarrow \mathsf{D}[\mathsf{label}]$
        $\_, \mathsf{aux} \leftarrow \mathcal{S}_{\mathsf{C}}(\texttt{Open}, \mathsf{tval}, \mathsf{val})$
        $\mathsf{D}[\mathsf{label}] \leftarrow (\mathsf{val}, t, \mathsf{tval}, \mathsf{aux})$
     else
        $(\mathsf{epno}_{\mathsf{label}}, \mathsf{tval}, \mathsf{aux}) \leftarrow (\bot, \bot, \bot)$
- • $\pi_{\mathsf{OA}}, \_ \leftarrow \mathsf{OA.Query}(st_{\mathsf{OA}}[\mathsf{G}[u]], \mathsf{tlbl}, u)$
- • $\pi_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}} \leftarrow \mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, u)$
- • $\pi \leftarrow (\pi_{\mathsf{AVC}}, \pi_{\mathsf{OA}}, \pi_{\mathsf{VRF}}, \mathsf{tlbl}, \mathsf{tval}, \mathsf{aux}, \mathsf{com}_{\mathsf{INT}})$
- • **return** $\pi, \mathsf{val}, \mathsf{epno}_{\mathsf{label}}$
▷ $\mathcal{S}(\texttt{Leak}, \mathsf{D}_{leak})$:
- • **parse** st as $(\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- • For each $(\mathsf{label}, \mathsf{val}, t) \in \mathsf{D}_{leak}$:
    * If $\mathsf{label} \notin \mathsf{D}$:
        · Choose and remove from D' a record $(\mathsf{tlbl}, (t', \mathsf{tval}))$ such that $t = t'$
        · $\mathsf{D} \leftarrow \mathsf{D} \cup \{(\mathsf{label}, (\bot, t, \mathsf{tval}, \bot))\}$
        · For $i = \mathsf{G}[t], \ldots, \mathsf{g}$: $\mathsf{D}_{\mathsf{VRF}}[i][\mathsf{label}] = P[\mathsf{tlbl}][i]$
    * $(\_, \_, \mathsf{tval}, \_) \leftarrow \mathsf{D}[\mathsf{label}]$

* * _, aux ← $\mathcal{S}_C$(Open, tval, val)
    * * D[label] ← (val, t, tval, aux)
  * $sk_{g_{cor}+1}, \ldots, sk_g \leftarrow \mathcal{S}$(Corrupt, D_{VRF}).
  * For $j = g_{cor} + 1, \ldots, g$, set $K_{VRF}[j] \leftarrow (sk_j, pk_j)$.
  * $g_{cor} \leftarrow g$.
  * **return** (K_{VRF}, D, com, epno, g, st_{OA}, st_{AVC})
  ▷ $\mathcal{S}$(ProveExt, (t_0, t_1)):
  * **parse** st as (K_{VRF}, D, com, epno, g, G, st_{OA}, st_{AVC})
  * **return** AVC.ProveExt(st_{AVC}, t_0, t_1)

Lemma 22. $|\Pr[Hyb2 \rightarrow 1] - \Pr[Hyb1 \rightarrow 1]| \leq negl(\lambda)$.

Proof: Intuitively, this hybrid acts exactly the same as the previous game, with two major differences (highlighted in orange). These differences exist in order to be able to have the simulator only rely on the information given by the leakage functions, as opposed to the full oracle query inputs from the adversary. In particular, in the previous hybrid the simulator requires access to all inserted (label, val) pairs to process updates, whereas here the simulator only knows the number of (label, val) pairs inserted save when a pair is queried or leaked. However, notice that in $Hyb2$ the simulator already does not take advantage of this information to produce its outputs, and therefore these changes are mostly syntactic.

More in detail, we will assume for simplicity that there are no collisions when sampling uniformly from Y_{VRF} (as those occur with at most negligible probability since Y_{VRF} is of exponential size and we sample polynomially many values), and we will show that $Hyb3$ is identically distributed to $Hyb2$ when this is the case.

When handling Update queries (with $leaked = false$), the simulator will only learn the number s of labels are being added to the datastore, and any labels $label_1, \ldots, label_n$ that had been queried since the last PCSUpdate query and were not in the database, but are being added now. To simulate the update, the simulator picks $s - n$ values at random from Y_{VRF} as the output of the VRF on the unknown labels, and uses the same tlbl from D_{VRF} for labels that were queries before. Then it simulates enough commitments (without using the committed values, as in $Hyb2$), and adds all the relevant pairs, each marked with the current epoch, to a table D' (mapping VRF outputs and epoch numbers to simulated commitments) and to the ordered accumulator. Note that since in $Hyb2$ the random VRF outputs and commitments are not simulated using the label or val input by the adversary, data used to update the order accumulator (and therefore the output of this query) have the same distribution as in $Hyb2$ (conditioned on no Y_{VRF} collisions, as detailed above). In case of a PCSUpdate query (again with $leaked = false$), the simulator selects new random outputs from Y_{VRF} for all the elements already in the datastore, and invokes the simulator to obtain a rotation proof. It stores the relationship between all the VRF outputs for the same (unknown) label at different generations using table P, and once the label becomes known later (as a result of Query or LeakState calls), it uses P to populate D_{VRF}. (Note that the fact that we sample from Y_{VRF} without collisions ensures that P is never overwritten in $Hyb3$, which ensures that the view matches $Hyb2$.) Then the simulator handles additions to the datastore as in Update queries above.

The cases where $leaked = true$ give the simulator more leakage, but are otherwise analogous.

To handle Query oracle calls, the simulator always receives the label, and the epoch u at which the query is being made. To simulate the output of this query, the simulator would include a simulated VRF output and proof on input label for the appropriate generation $G[u]$, a commitment opening (only for inclusion proofs), and the appropriate OA and AVC proofs. The simulator keeps track of all the VRF outputs returned to the adversary (or added to the OA) in D_{VRF}, so that these outputs are always consistent across multiple calls. If the label was inserted to the datastore at a generation $G[t] \leq G[u]$, the simulator can use one of the tuples (tlbl$'$, t, tval) stored in D' and assign it to label, using $P[\text{tlbl}']$ to update D_{VRF} for $G[t]$ and all subsequent generations. Note that the view produced in this way has the same distribution as if the label was associated with tlbl$'$ during Update or PCSUpdate, as in $Hyb2$. Otherwise, the simulator simply picks the output of the VRF at random, updates D_{VRF} accordingly and produces an honest absence proof for the OA. The simulator can thus return the simulated VRF proof, the simulated commitment opening (in case of an inclusion proof), plus the honestly generated OA and AVC proofs.

ProveExt oracle calls are generated as in $Hyb2$.

Finally, for LeakState proofs, the simulator receives all the labels and values which are part of the datastore, and can use them to fill in the simulated honest state, using each entry from D' to add a simulated record to D (as is done for Query calls), and the values to generate consistent commitment openings on unqueried labels. The VRF simulator can be used (by making a Corrupt call on input D_{VRF}) to produce secret keys which are consistent with all the proofs being given out. Again, this simulated state has the same distribution as in $Hyb2$.