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attractive features, such as a stronger (extractable) soundness notion, and the ability for a communication party with out-of-date data to efficiently "catch up" to the current epoch while ensuring that the server did not erase any of the past data.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 [45], Signal [38], Keybase [21], iMessage [2]) and encrypted calls (Zoom [6], Webex [44], Teams [32]), 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 severe usability challenges [3, 18, 19, 43]. *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 [26] 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 $(\mathsf{label}, \mathsf{val})$ 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 [29], or on public and untamperable ledgers such as blockchains [22]. 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] [14, 28], improvements in this respect seem feasible, and even the

[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 $(\mathsf{label}, \mathsf{val})$ 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.

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 [25] is the first deployment of an auditable public key directory; they published the first KT digest on April 2014 [24]. 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 [23] 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 [27], 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 [6,17]. 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 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 [20, 28, 40–42]. 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) [7,17,29]. 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 [10], practical KT systems (with either content-privacy or metadata-privacy) achieve privacy through asymmetric primitives such as Verifiable Random Functions (VRFs) [31].[7] Ignoring some important details, given a $(\mathsf{label}, \mathsf{val})$ 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 [6,13,17]. 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 [35,36]. For ppKT,

[7] Informally, a VRF [31] 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.

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 [7]. 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 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

[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.

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 [7, 17, 29] 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 [7].

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 $(\mathsf{label}, \mathsf{val})$ 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 $\{(\mathsf{label}, \mathsf{val})\}$-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 $\mathsf{com}_{t'}$ commits to a given older commitment com_t (and, therefore, also implies that both $\mathsf{com}_{t'}$ and com_t commit to the same sequence $\mathsf{com}_1, \mathsf{com}_2, \ldots, \mathsf{com}_{t-1}$), 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

[10] According to a well-defined leakage profile. For [7], 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.

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 [7,41].

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 [7]. 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 [7] 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

[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.

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 [7]. SEEMless uses — in a black-box way — a *verifiable random function* (VRF) [31] and cryptographic hash function to build their aZKS, and recommends the specific DDH-based VRF from the upcoming VRF standard [16].

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 [7], 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 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

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

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 [9] 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) [5, 34], where one compact secret is supposed to open many previously committed ciphertexts in a certain way. As with non-committing encryption [34], 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., [4]), 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 [16]) 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 $(g, pk, F(pk, x), z)$ [12].)

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 [16] 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]

[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.

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, 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 [37].[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.

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

[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.

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 [11, 39]. Other important examples include optimal structure-preserving signature schemes [1] and state-restoration soundness analysis of Bulletproofs [15].

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.

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 the full version of the paper, 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 [8,30] and extended to append-only ZKS (aZKS) in [7]. 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:

- $\triangleright$ $\mathsf{pp} \leftarrow \mathcal{Z}.\mathsf{GenPP}(1^\lambda)$: 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.
- $\triangleright$ $(\mathsf{com}, \mathsf{st}) \leftarrow \mathcal{Z}.\mathsf{Init}(\mathsf{pp})$: 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.)
- $\triangleright$ $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathcal{Z}.\mathsf{Update}(\mathsf{pp}, \mathsf{st}, S)$, $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathcal{Z}.\mathsf{PCSUpdate}(\mathsf{pp}, \mathsf{st}, S)$: 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 $\mathsf{P.Verify}(\mathsf{com}_u, \mathsf{label}, \mathsf{val}', t', \pi)$ succeeds, where (val', t', π) are those produced by the corresponding call to P.Query.

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:

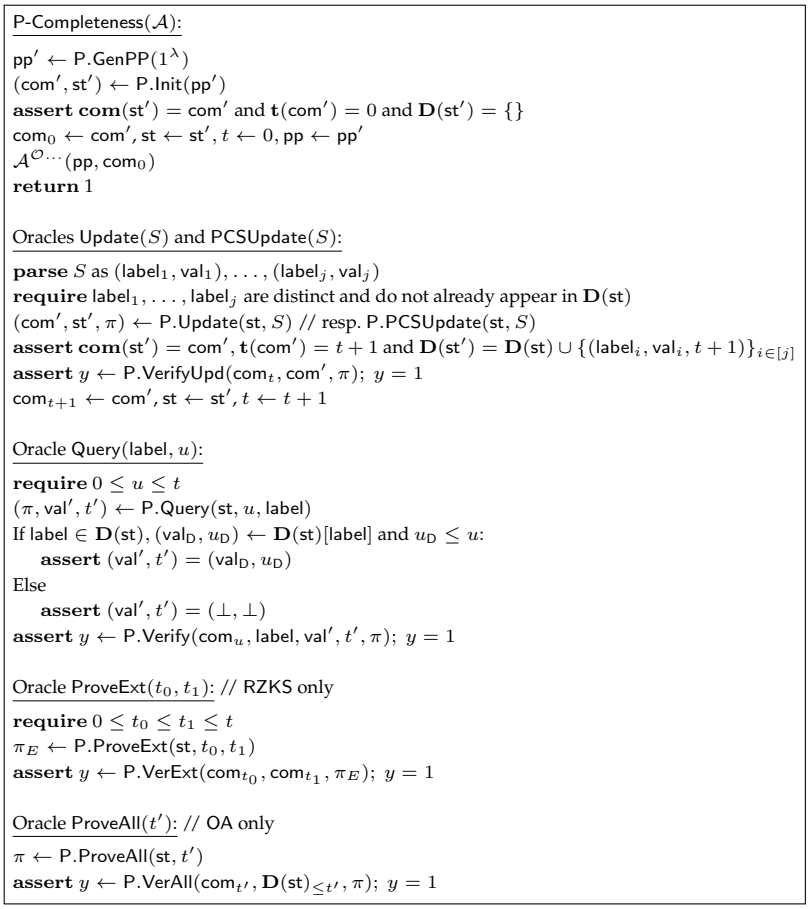

Fig. 1: Completeness for RZKS and OA (an Ordered Accumulator, defined in Section 5.1) primitives (denoted with P). Some of the oracles are only applicable to one primitive. In this experiment, the adversary can read all the game's state and the oracle's intermediate variables, such as $\mathsf{com}_i \forall i, \mathsf{st}, y$. 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.

- $\mathsf{pp}', \mathsf{st} \leftarrow \mathsf{Extract}(\texttt{Init})$: Samples public parameters indistinguishable from honestly generated public parameters such that extraction will be possible. Also generates an initial state.
- $D_{\mathsf{com}} \leftarrow \mathsf{Extract}(\texttt{Extr}, \mathsf{st}, \mathsf{com})$: Takes in the internal state and a commitment to the datastore. Outputs the set of $(\mathsf{label}, \mathsf{val}, i)$ committed to.
- $C_{\mathsf{com}} \leftarrow \mathsf{Extract}(\texttt{ExtrC}, \mathsf{st}, \mathsf{com})$: Takes in the internal state and a commitment to the datastore. Outputs the set of previous commitments, indexed by epoch.

– $out, \mathsf{st} \leftarrow \mathsf{Extract}(\texttt{Ideal}, \mathsf{st}, in)$: 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.

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 $\mathsf{D}[\mathsf{com}][\mathsf{label}] = (\mathsf{val}^*, i^*)$ in CheckVerD. This will be false if the adversary can provide a proof that $(\mathsf{label}, \mathsf{val}^*, i^*)$ is in the datastore with digest com, but the extractor expects this datastore to either not contain label or to contain $(\mathsf{label}, \mathsf{val}, i)$ 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:

– $\mathsf{com}', \mathsf{pp}' \leftarrow \mathcal{S}(\texttt{Init})$: Samples public parameters and an initial commitment indistinguishable from honest public parameters such that it will be possible to simulate proofs.

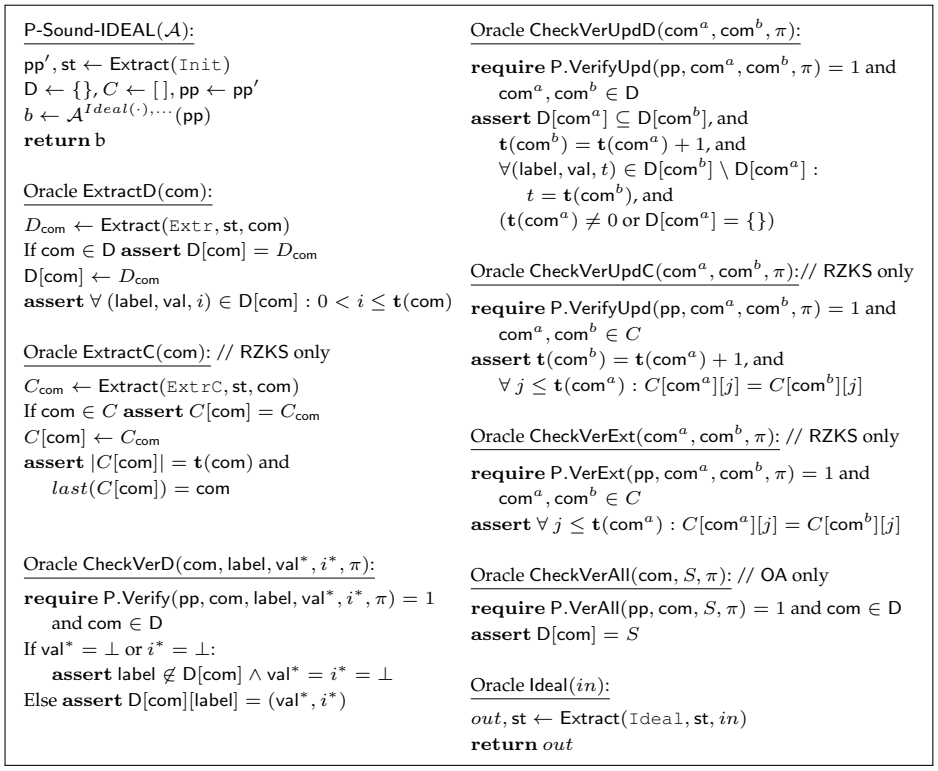

Fig. 2: Soundness for RZKS and OA (both denoted by P). In the ideal world, the map D stores, for each commitment com, the datastore that the Extract algorithm output for that commitment. In addition the map C stores, for each commitment, the (ordered) list of commitments to previous epochs. When the 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 $\mathsf{pp} \leftarrow \mathsf{P.GenPP}(1^\lambda)$, 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.

- $(\mathsf{com}', \pi) \leftarrow \mathcal{S}(\texttt{(PCS)Update}, l)$: 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.
- $(\pi, \mathsf{val}', t') \leftarrow \mathcal{S}(\texttt{Query}, l)$: 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 $\mathsf{D}[\mathsf{label}] = (\mathsf{val}', t')$, or an absence proof if label $\notin$ D.

- $\pi \leftarrow \mathcal{S}(\texttt{ProveExt}, l)$: 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.
- st $\leftarrow \mathcal{S}(\texttt{Leak}, l)$: Takes in partial information $l \leftarrow L_{\mathsf{LeakState}}()$ about the data-store 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 $\mathsf{Update}(\{\mathsf{label}, \mathsf{val}\})$ oracle for some $(\mathsf{label}, \mathsf{val})$, then the $(\pi, \mathsf{val}, 1) \leftarrow \mathsf{Query}(\mathsf{label}, 1)$ 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.

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 [7], 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

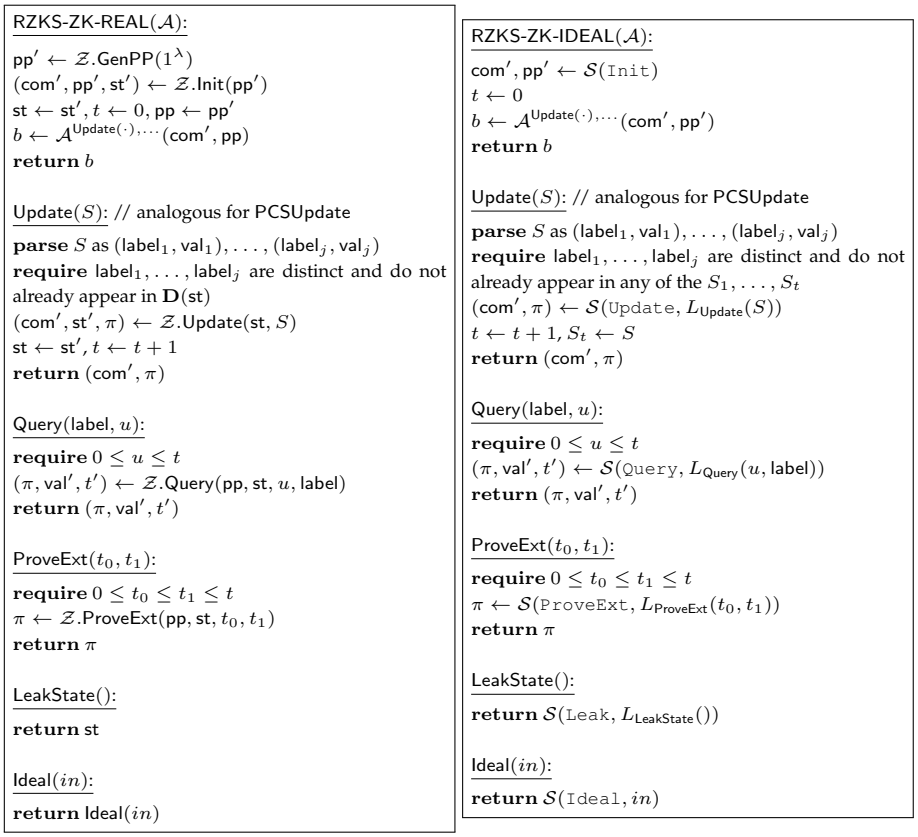

Fig. 3: Zero Knowledge (with leakage) security experiments for RZKS. $\mathcal{S}$ is a stateful algorithm (whose state we omit to simplify the notation). The leakage functions $L_{\mathsf{Update}}, L_{\mathsf{Query}}, \ldots$ also share state among each other.

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 [33], 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 $\mathsf{VRF} = (\mathsf{GenPP}, \mathsf{KeyGen}, \mathsf{Query}, \mathsf{Verify}, \mathsf{Rotate}, \mathsf{VerRotate})$ defined as follows:

- ▷ $\mathsf{pp} \leftarrow \mathsf{VRF.GenPP}(1^\lambda)$: This algorithm takes the security parameter and produces public parameter pp for the scheme. All other algorithms take these pp as input, even when not explicitly specified.
- ▷ $(sk, pk) \leftarrow \mathsf{VRF.KeyGen}(\mathsf{pp})$: The key generation algorithm takes in the global pp and outputs the public key pk and secret key sk.
- ▷ $(y, \pi) \leftarrow \mathsf{VRF.Query}(\mathsf{pp}, sk, x)$: 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 $\mathsf{VRF.Eval}(sk, x)$ the first output y of the Query algorithm (i.e. Eval does not return a proof).
- ▷ $1/0 \leftarrow \mathsf{VRF.Verify}(\mathsf{pp}, pk, x, y, \pi)$: This deterministic function verifies the proof π that y is the output of the VRF defined by pk on input x.
- ▷ $sk', pk', \pi \leftarrow \mathsf{VRF.Rotate}(\mathsf{pp}, sk, X)$: 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 $P = \{(\mathsf{VRF.Eval}(sk, x), \mathsf{VRF.Eval}(sk', x))\}_{x \in X}$ satisfies the relationship that each pair corresponds to the same input x (without leaking information about X beyond its size).
- ▷ $0/1 \leftarrow \mathsf{VRF.VerRotate}(\mathsf{pp}, pk, pk', P, \pi)$: 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:

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

[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.

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

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

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.

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 `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 [9]. 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.

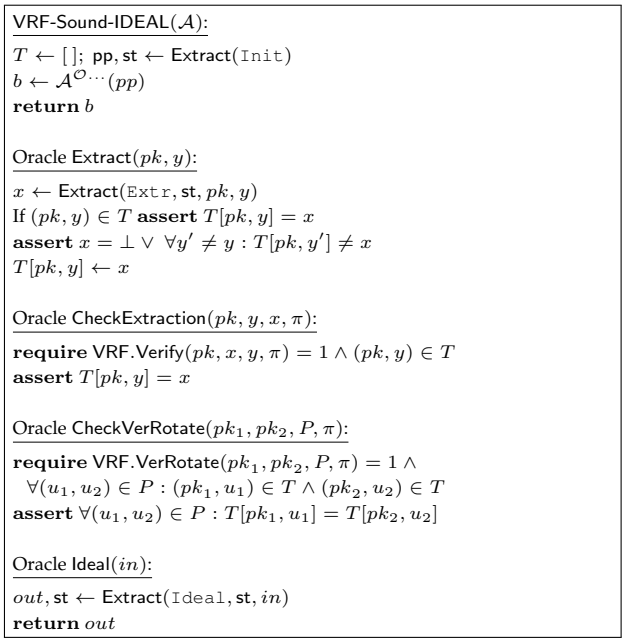

Fig. 4: Soundness for VRF. In the real world (not pictured), the public parameters are generated as pp $\leftarrow$ VRF.GenPP(1^λ), and the oracles do not do anything, except for the Ideal one which implements the necessary ideal objects according to their specification.

4.2 Rotatable VRF Construction

Our rotatable Verifiable Random Function VRF $=$ (GenPP, KeyGen, Query, Verify, Rotate, 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 VRF.Eval(sk, x) and 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

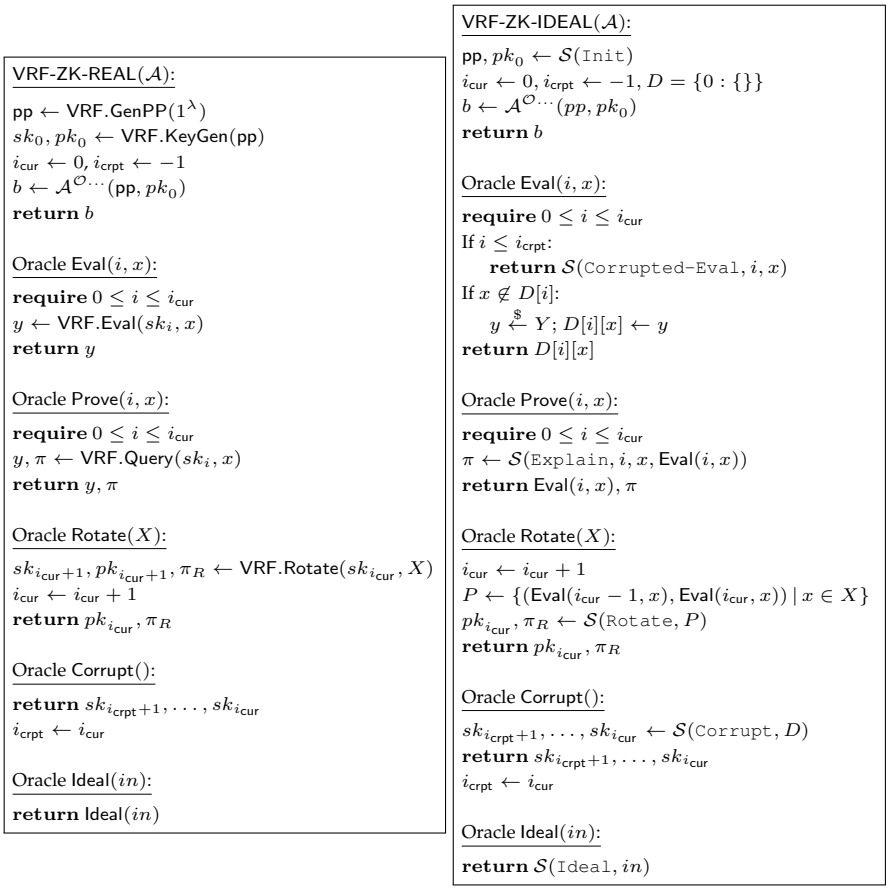

Fig. 5: Zero Knowledge experiments for the Rotatable VRF.

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 $x \in X$.

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. A detailed formal proof of the following theorem is deferred to the full version of this paper.

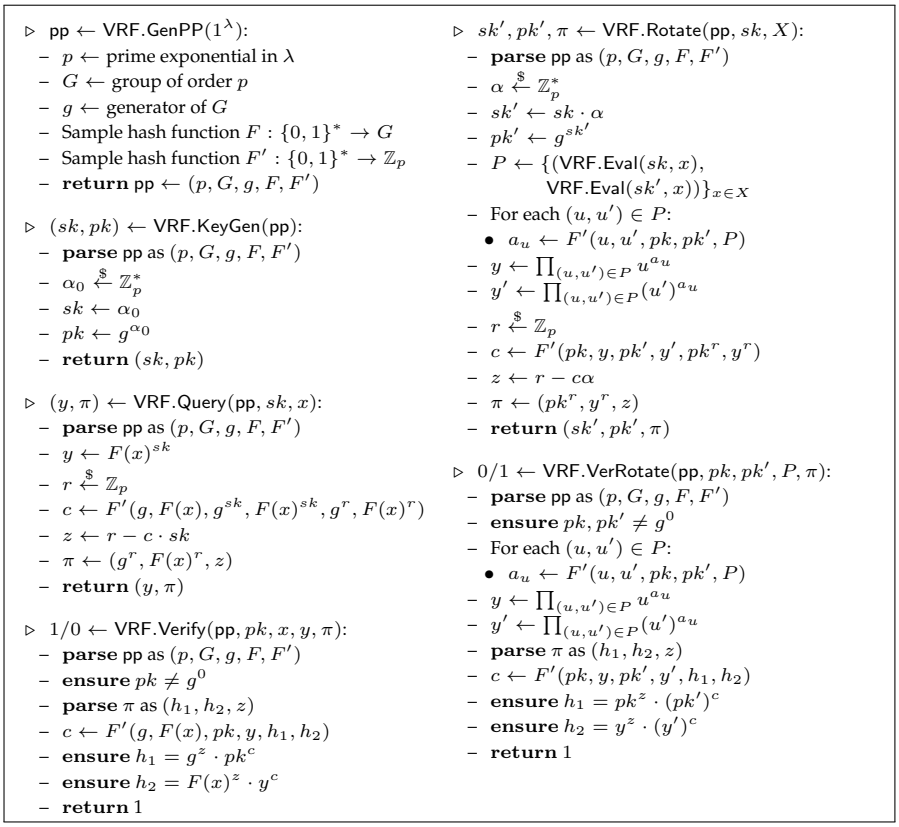

Fig. 6: Our Rotatable VRF construction.

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 $\mathcal{A}$,*

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

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 [16] 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. A detailed formal proof of the following theorem is deferred to the full version of this paper.

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 $\mathcal{A}$,*

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

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 the full version of the paper, 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 the full version of the paper.

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

▷ $\mathsf{pp} \leftarrow \mathsf{C.GenPP}(1^\lambda)$: On input the security parameter, GenPP outputs public parameters pp.
▷ $\mathsf{com}, \mathsf{aux} \leftarrow \mathsf{C.Commit}(\mathsf{pp}, m)$: Using the global parameters pp, the (randomized) commit algorithm produces commitment com to message m, and decommitment information aux.
▷ $1/0 \leftarrow \mathsf{C.Verify}(\mathsf{pp}, \mathsf{com}, m, \mathsf{aux})$: 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 the full version of the paper. Completeness and soundness are defined analogously to RZKS in figures 1 and 2 respectively.

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

▷ GenPP, Init, Update, VerifyUpd, Query, Verify are defined analogously to the RZKS in Definition 1.
▷ $\pi \leftarrow \mathsf{OA.ProveAll}(\mathsf{pp}, \mathsf{st}, u)$: 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.
▷ $1/0 \leftarrow \mathsf{OA.VerAll}(\mathsf{pp}, \mathsf{com}_u, P, \pi)$: This deterministic algorithm takes a digest com_u, a set P of $(\mathsf{label}, \mathsf{val}, t)$ 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 the full version of the paper.

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

▷ $\mathsf{pp} \leftarrow \mathsf{AVC.GenPP}(1^\lambda)$: This algorithm takes the security parameter and produces public parameter pp for the scheme. All other algorithms take these pp as input, even when not explicitly specified.
▷ $(\mathsf{com}, \mathsf{st}) \leftarrow \mathsf{AVC.Init}(\mathsf{pp})$: This algorithm produces an initial commitment com to an empty list $\mathsf{D}_0 = \{\}$, and an initial server/prover state st. Each server state st will contain a list and a digest, which we will refer to as $\mathbf{D}(\mathsf{st})$ and

$\mathbf{com}(\mathsf{st})$. Similarly, each commitment will include an integer $\mathbf{t}(\mathsf{com})$ (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.)

- ▷ $(\mathsf{com}', \mathsf{st}', \pi_S) \leftarrow \mathsf{AVC.Update}(\mathsf{pp}, \mathsf{st}, \mathsf{val})$: 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 $\mathbf{D}(\mathsf{st}') = \mathbf{D}(\mathsf{st})||\mathsf{val}$ of size $\mathbf{t}(\mathsf{com}') = \mathbf{t}(\mathbf{com}(\mathsf{st})) + 1$.
- ▷ $\pi \leftarrow \mathsf{AVC.ProveExt}(\mathsf{pp}, \mathsf{st}, t', t)$: 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 $\mathsf{com}_{t'}$.
- ▷ $0/1 \leftarrow \mathsf{AVC.VerExt}(\mathsf{pp}, \mathsf{com}', \mathsf{com}, \pi)$: 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.
- ▷ $(\pi, \mathsf{val}) \leftarrow \mathsf{AVC.Query}(\mathsf{pp}, \mathsf{st}, u, t')$: This algorithm takes as input a state st and epochs u and t' such that $u \leq t' \leq \mathbf{t}(\mathsf{st})$. It returns $\mathsf{val} = \mathbf{D}(\mathsf{st})[u]$ and a membership proof π to be verified against the commitment $\mathsf{com}_{t'}$ output by Update during the t'-th update.
- ▷ $0/1 \leftarrow \mathsf{AVC.Verify}(\mathsf{pp}, \mathsf{com}, u, \mathsf{val}, \pi)$: 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 7. The RZKS commits to a set of $(\mathsf{label}, \mathsf{val})$ 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 $(\mathsf{label}, \mathsf{val})$, 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

[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.

older one that the verifier currently holds (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 [7], 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 $(\mathsf{label}, \mathsf{val})$ pairs in the RZKS, and epno, the latest epoch number. It also contains $\mathsf{st}_{\mathsf{OA}}$ and $\mathsf{st}_{\mathsf{AVC}}$, the underlying state of the OA and AVC, respectively. It stores $\mathsf{com}_{\mathsf{epno}}$, 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 $\mathsf{K}_{\mathsf{VRF}}$, 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 7 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 7 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 the full version of the paper.

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

▷ $(\mathsf{com}, \mathsf{st}', \pi) \leftarrow \mathsf{RZKS.Update}(\mathsf{st}, S_{\mathsf{update}})$:
$(\mathsf{com}, \mathsf{st}', \pi) \leftarrow \mathsf{RZKS.PCSUpdate}(\mathsf{st}, S_{\mathsf{update}})$:
// 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 $\mathsf{g}' \in \{\mathsf{g}, \mathsf{g}-1\}$: $\{\mathsf{tlbl}^{\mathsf{g}'}_j\}_{j \in L} \leftarrow \{\mathsf{VRF.Eval}(\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}'].sk, j)\}_{j \in L}$
- □ $\pi^{\mathsf{g}-1}_{\mathsf{OA}} \leftarrow \mathsf{OA.ProveAll}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g}-1], \mathsf{epno}-1)$
- □ $\mathsf{st}', \_ \leftarrow \mathsf{OA.Init}(\mathsf{pp}_{\mathsf{OA}})$; For $i \in [\mathsf{epno}-1]$:
  - $S_{\mathsf{OA}} \leftarrow \{(\mathsf{tlbl}^{\mathsf{g}}_j, \mathsf{tval}_j) \mid (j, (\cdot, i, \mathsf{tval}_j, \cdot)) \in \mathsf{D}\}$
  - $\mathsf{st}', \mathsf{com}'_{\mathsf{OA}}, \_ \leftarrow \mathsf{OA.Update}(\mathsf{st}', S_{\mathsf{OA}})$
- □ $\mathsf{st}_{\mathsf{OA}}[\mathsf{g}] \leftarrow \mathsf{st}', \mathsf{com}^{\mathsf{epno}-1}_{\mathsf{OA}} \leftarrow \mathsf{com}'_{\mathsf{OA}}$
- □ $\pi^{\mathsf{g}}_{\mathsf{OA}} \leftarrow \mathsf{OA.ProveAll}(\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], \mathsf{epno}-1)$
- $S_{\mathsf{OA}} \leftarrow \{\}$; For each $(\mathsf{label}_i, \mathsf{val}_i) \in S_{\mathsf{update}}$:
  - $\mathsf{tlbl}_i \leftarrow \mathsf{VRF.Eval}(\mathsf{K}_{\mathsf{VRF}}[\mathsf{g}].sk, \mathsf{label}_i)$
  - $\mathsf{tval}_i, \mathsf{aux}_i \leftarrow \mathsf{C.Commit}(\mathsf{val}_i)$
  - $S_{\mathsf{OA}} \leftarrow S_{\mathsf{OA}} \cup \{(\mathsf{tlbl}_i, \mathsf{tval}_i)\}$
  - $\mathsf{D} \leftarrow \mathsf{D} \cup \{(\mathsf{label}_i, (\mathsf{val}, \mathsf{epno}, \mathsf{tval}_i, \mathsf{aux}_i))\}$
- $\mathsf{st}_{\mathsf{OA}}[\mathsf{g}], \mathsf{com}^{\mathsf{epno}}_{\mathsf{OA}}, \pi_{\mathsf{OA}} \leftarrow \mathsf{OA.Update}(st_{\mathsf{OA}}[\mathsf{g}], S_{\mathsf{OA}})$; $\mathsf{com}^{\mathsf{epno}}_{\mathsf{INT}} \leftarrow (\mathsf{com}^{\mathsf{epno}}_{\mathsf{OA}}, \mathsf{K}_{\mathsf{VRF}}[\mathsf{g}].pk)$; $\mathsf{G}[\mathsf{epno}] \leftarrow \mathsf{g}$; $\pi' \leftarrow \pi_{\mathsf{OA}}$
- $\mathsf{com}, \mathsf{st}_{\mathsf{AVC}}, \pi_{\mathsf{AVC}} \leftarrow \mathsf{AVC.Update}(\mathsf{st}_{\mathsf{AVC}}, \mathsf{com}^{\mathsf{epno}}_{\mathsf{INT}})$
- $\_, \pi^{\mathsf{epno}}_{\mathsf{AVC}} \leftarrow \mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, \mathbf{t}(\mathsf{com}), \mathbf{t}(\mathsf{com}))$
- $\mathsf{com}^{\mathsf{epno}-1}_{\mathsf{INT}}, \pi^{\mathsf{epno}-1}_{\mathsf{AVC}} \leftarrow \mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, \mathbf{t}(\mathsf{com})-1, \mathbf{t}(\mathsf{com})-1)$
- □ $\pi' \leftarrow (\pi_{\mathsf{OA}}, \pi^{\mathsf{g}-1}_{\mathsf{OA}}, \pi^{\mathsf{g}}_{\mathsf{OA}}, \pi_{\mathsf{VRF}}, \mathsf{com}^{\mathsf{epno}-1}_{\mathsf{OA}}, \{(\mathsf{tlbl}^{g-1}_j, \mathsf{tlbl}^{g}_j, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L})$
- $\pi \leftarrow (\pi', \pi_{\mathsf{AVC}}, \mathsf{com}^{\mathsf{epno}}_{\mathsf{INT}}, \mathsf{com}^{\mathsf{epno}-1}_{\mathsf{INT}}, \pi^{\mathsf{epno}}_{\mathsf{AVC}}, \pi^{\mathsf{epno}-1}_{\mathsf{AVC}})$
- $\mathsf{st} \leftarrow (\mathsf{K}_{\mathsf{VRF}}, \mathsf{D}, \mathsf{com}, \mathsf{epno}, \mathsf{g}, \mathsf{G}, \mathsf{st}_{\mathsf{OA}}, \mathsf{st}_{\mathsf{AVC}})$
- **return** $(\mathsf{com}, \mathsf{st}, \pi)$

▷ $0/1 \leftarrow \mathsf{RZKS.VerifyUpd}(\mathsf{com}^{t_0}, \mathsf{com}^{t_1}, \pi)$:
- **parse** π as $(\pi', \pi_{\mathsf{AVC}}, \mathsf{com}^{t_1}_{\mathsf{INT}}, \mathsf{com}^{t_0}_{\mathsf{INT}}, \pi^{t_1}_{\mathsf{AVC}}, \pi^{t_0}_{\mathsf{AVC}})$
- **parse** $\mathsf{com}^{t_0}_{\mathsf{INT}}$ as $(\mathsf{com}^{t_0}_{\mathsf{OA}}, pk_{t_0})$
- **parse** $\mathsf{com}^{t_1}_{\mathsf{INT}}$ as $(\mathsf{com}^{t_1}_{\mathsf{OA}}, pk_{t_1})$
- **ensure** $\mathsf{OA.t}(\mathsf{com}^{t_0}_{\mathsf{OA}}) + 2 = \mathsf{AVC.t}(\mathsf{com}^{t_0}) + 1 = \mathsf{AVC.t}(\mathsf{com}^{t_1}) = \mathsf{OA.t}(\mathsf{com}^{t_1}_{\mathsf{OA}}) + 1$
- **ensure** $\mathsf{AVC.VerExt}(\mathsf{com}^{t_0}, \mathsf{com}^{t_1}, \pi_{\mathsf{AVC}}) = 1$
- For $t \in \{t_0, t_1\}$: **ensure** $\mathsf{AVC.Verify}(\mathsf{com}^t, \mathsf{AVC.t}(\mathsf{com}^t), \mathsf{com}^t_{\mathsf{INT}}, \pi^t_{\mathsf{AVC}}) = 1$
- If $pk_{t_0} = pk_{t_1}$:
  - **parse** π' as π_{OA}; set $\mathsf{com}'_{\mathsf{OA}} \leftarrow \mathsf{com}^{t_0}_{\mathsf{OA}}$

  Else:
  - **parse** π' as $(\pi_{\mathsf{OA}}, \pi^{\mathsf{g}-1}_{\mathsf{OA}}, \pi^{\mathsf{g}}_{\mathsf{OA}}, \pi_{\mathsf{VRF}}, \mathsf{com}'_{\mathsf{OA}}, \{(\mathsf{tlbl}^{g-1}_j, \mathsf{tlbl}^{g}_j, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L})$
  - **ensure** $\mathsf{VRF.VerRotate}(pk_{t_0}, pk_{t_1}, \{(\mathsf{tlbl}^{\mathsf{g}-1}_j, \mathsf{tlbl}^{\mathsf{g}}_j)\}_{j \in L}, \pi_{\mathsf{VRF}}) = 1$
  - **ensure** $\mathsf{OA.VerAll}(\mathsf{com}^{t_0}_{\mathsf{OA}}, \{(\mathsf{tlbl}^{\mathsf{g}-1}_j, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L}, \pi^{\mathsf{g}-1}_{\mathsf{OA}}) = 1$
  - **ensure** $\mathsf{OA.VerAll}(\mathsf{com}'_{\mathsf{OA}}, \{(\mathsf{tlbl}^{\mathsf{g}}_j, \mathsf{tval}_j, \mathsf{epno}_j)\}_{j \in L}, \pi^{\mathsf{g}}_{\mathsf{OA}}) = 1$
- **ensure** $\mathsf{OA.VerifyUpd}(\mathsf{com}'_{\mathsf{OA}}, \mathsf{com}^{t_1}_{\mathsf{OA}}, \pi_{\mathsf{OA}}) = 1$
- **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:
  - $(\mathsf{val}, \mathsf{epno}_{\mathsf{label}}, \mathsf{tval}, \mathsf{aux}) \leftarrow (\perp, \perp, \perp, \perp)$
- $\pi_{\mathsf{OA}}, \_ \leftarrow \mathsf{OA.Query}(st_{\mathsf{OA}}[\mathsf{G}[u]], u, \mathsf{tlbl})$
- $\pi_{\mathsf{AVC}}, \mathsf{com}_{\mathsf{INT}} \leftarrow \mathsf{AVC.Query}(\mathsf{st}_{\mathsf{AVC}}, u, u)$
- $\pi \leftarrow (\pi_{\mathsf{AVC}}, \pi_{\mathsf{OA}}, \pi_{\mathsf{VRF}}, \mathsf{tlbl}, \mathsf{tval}, \mathsf{aux}, \mathsf{com}_{\mathsf{INT}})$
- **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 $t = \perp \vee \mathsf{val} = \perp \vee \mathsf{tval} = \perp$ Then **ensure** $\mathsf{val} = \mathsf{tval} = t = \perp$ 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. 7: 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 $\mathsf{AVC.t}(\mathsf{com}) - 1$.

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 $\boldsymbol{L}$.

- 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).
- $\boldsymbol{L}_{\mathsf{Query}}(\mathsf{label}, u)$: If $\exists(\mathsf{label}, \mathsf{val}, t') \in \mathsf{D}$ such that $t' \leq u$, the function returns $(\mathsf{label}, \mathsf{val}, t', u)$. If $\exists(\mathsf{label}, \mathsf{val}, t') \in \mathsf{D}$ such that $G[t'] = G[u]$, the function returns $(\mathsf{label}, \perp, t', u)$. Otherwise, it returns $(\mathsf{label}, \perp, \perp, u)$ and, if $G[u] = g$, adds label to X.
- $\boldsymbol{L}_{\mathsf{Update}}(S)$: 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.
- $\boldsymbol{L}_{\mathsf{PCSUpdate}}(S)$: 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 $|S|$.
- $\boldsymbol{L}_{\mathsf{LeakState}}(S)$: Set $leaked \leftarrow true$. **return** D.
- $\boldsymbol{L}_{\mathsf{ProveExt}}(t_0, t_1)$: **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* $\boldsymbol{L}$ *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 $\boldsymbol{L}$ 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 the full version of the paper.

5.4 Instantiation and Complexity

If we allow each building block to be instantiated as constructed in the full version of the paper, 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 8.

Acknowledgements

At the commencement of the work leading to this paper, the authors had discussions with Melissa Chase (of Microsoft), and Julia Len (an intern at Zoom). The authors are appreciative of their contributions.

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.

	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. 8: The complexity of our various constructions.

4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.

10. 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.
11. 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.
12. 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.
13. Novi Financial. Auditable key directory. https://github.com/novifinancial/akd/, 2021. Accessed: 2022-05-26.
14. 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.
15. Ashrujit Ghoshal and Stefano Tessaro. Tight state-restoration soundness in the algebraic group model. LNCS, pages 64–93. Springer, Heidelberg, 2021.
16. 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.
17. Google. Key transparency overview. https://github.com/google/keytransparency/blob/master/docs/overview.md. Accessed: 2022-08-31.
18. 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.
19. 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.
20. 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.
21. Keybase.io. Keybase chat. https://book.keybase.io/docs/chat. Accessed: 2022-08-03.
22. keybase.io. Keybase is now writing to the stellar blockchain. https://book.keybase.io/docs/server/stellar. Accessed: 2022-07-29.
23. 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.
24. keybase.io. Keybase first commitment. https://keybase.io/_/api/1.0/merkle/root.json?seqno=1, 2014. Accessed: 2022-05-26.
25. Keybase.io. Keybase is not softer than tofu. https://keybase.io/blog/chat-apps-softer-than-tofu, 2019. Accessed: 2019-05-05.
26. Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate Transparency Version 2.0. RFC 9162, December 2021.
27. 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.
28. 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.

29. 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.
30. 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.
31. Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. Verifiable random functions. In *40th FOCS*, pages 120–130. IEEE Computer Society Press, October 1999.
32. microsoft.com. Teams end-to-end encryption. https://docs.microsoft.com/en-us/microsoftteams/teams-end-to-end-encryption, 2022. Accessed: 2022-05-26.
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. signal.org. Technical information. https://www.signal.org/docs, 2016. Accessed: 2022-08-03.
39. signal.org. Technology preview: Signal private group system. https://signal.org/blog/signal-private-group-system/, 2019. Accessed: 2022-08-22.
40. 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.
41. 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.
42. 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.
43. 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.
44. 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.
45. whatsapp.com. Whatsapp encryption overview. White paper, 2021. Accessed: 2022-08-03.