Weak Consistency mode in Key Transparency: OPTIKS
A Short Note

Esha Ghosh[*] Melissa Chase[†]

Microsoft Research

Abstract

The need for third-party auditors in privacy-preserving Key Transparency (KT) systems presents a deployment challenge. In this short note, we take a simple privacy-preserving KT system that provides strong security guarantees in the presence of an honest auditor (OPTIKS) and show how to add a auditor-free mode to it. The auditor-free mode offers slightly weaker security. We formalize this security property and prove that our proposed protocol satisfies our security definition.

1 Introduction

As Key Transparency (KT) is seeing more deployments [13, 2, 1] and standardization efforts [10], questions around having a third party auditor in the system is evolving as a major discussion point. On the upside, having a third party auditor to check some global consistency (such as, append-onlyness [3, 7, 5, 6] or version invariant [12, 11]) among the KT datastructures helps achieve strong consistency guarantees in a scalable way: if a malicious server ever distributes a fake key on behalf of a user when the user was offline, the user will detect this as soon as she comes online next. However, finding an auditing service is more challenging in practice: if the server pays a third party to be their auditor, clearly there is an incentive mismatch. The server could also serve the audit proofs to any anonymous request. But this becomes expensive for the server as it can be bombarded with anonymous audit requests. One potential solution to this problem would be to compress the audit proofs while retaining the efficiency of lookup, so that, the clients can cheaply verify the proofs themselves without the need for a dedicated auditor. While SNARKs can help achieve this, in theory, they are not yet suitable for a large KT scale deployment. So, this remains an open problem.

A second approach that is being proposed is the idea of getting rid of global consistency proofs altogether at the cost of achieving a slightly weaker notion of consistency [11, 4, 8]. Informally, weak consistency means the following: if a fake key is distributed on behalf of a user while she was online, either she or the recipient will detect it after the next time she and the recipient comes online. Whether this weaker notion of consistency is viable for companies, remains to be seen.

However, it would be interesting to see if any of the KT systems that typically operate with the third party auditor assumption, can be made to work in the auditor-less mode without any significant changes in the server's data structures. This will have the advantage of being able operate in both modes: if an auditor is available, the system can provide strong consistency. If not, or if the users do not want to rely on external auditors, the system can provide weak consistency for them. Moreover, many of these KT systems (e.g. [3, 7, 5, 6]) provide both content privacy and metadata privacy. It will be interesting if the transformation required for the auditor-less mode does not significantly weaken the privacy guarantees provided by these KT systems.

In this work, we take a simple KT system that operates in the strong consistency mode: OPTIKS. We show how OPTIKS can operate in the auditor-free weak consistency mode without changing the server's publish algorithm (i.e., the server's data structures). While we do not rigorously prove it, we believe this auditor-free mode does not introduce any significant leakage. Formalizing the privacy guarantee and enhancing the efficiency of the auditor-free mode is left as future work.

[*]esha.ghosh@microsoft.com
[†]melissac@microsoft.com

2 Preliminaries

In this section, we recall some of the cryptographic and combinatorial building blocks that we will be using for instantiating a privacy-preserving KT system (OPTIKS) in the auditor-free weak consistency mode.

The cryptographic primitive underlying OPTIKS is Privacy-Preserving Authenticated History Dictionary or PAHD. Here, we recall PAHD instantiation as in OPTIKS almost verbatim. The PAHD construction uses a cryptographic primitive called Ordered Zero Knowledge Sets or oZKS underneath. We recall oZKS first, and then describe the PAHD construction. The oZKS definition is also quoted from [6] verbatim.

I Ordered Zero Knowledge Set

- $\mathsf{oZKS.Init}$: The initialization algorithm outputs an initial commitment to the empty datastore.
- $\mathsf{oZKS.Update}$ / $\mathsf{oZKS.VerifyUpd}$: The update algorithm adds a set of new label-value pairs to the datastore, outputting the new commitment to the data and an update proof. The update verification algorithm then verifies the update proof between consecutive commitments.
- $\mathsf{oZKS.Query}$ / $\mathsf{oZKS.Verify}$: The query algorithm returns the value associated to the queried label, along with the query proof and the epoch that the label was added (or $\perp$ and a non-membership proof if the label is not a member). The query verification algorithm verifies the value returned by a query using the proof.

Soundness of oZKS without auditor. Here we note that, the soundness property of oZKS is defined in the presence of an honest auditor (i.e., without a party to run a $\mathsf{oZKS.VerifyUpd}$ for each update) in [6]. However, in our construction, we will not use the $\mathsf{oZKS.VerifyUpd}$. So, we will use a weaker soundness guarantee compared to the oZKS in [6]. Informally, we will use the property that, for a given oZKS commitment for an epoch, for any label-value pair, proofs cannot be constructed for different values for the same label, or for different epochs for the label-value pair. In other words, an adversary $\mathcal{A}^*$ wins the soundness game if it can either produce $\mathsf{com}, \mathsf{label}, \mathsf{val}_1, \mathsf{val}_2, t_1, t_2, \pi_1, \pi_2$ such that both proofs verify for $\mathsf{val}_1 \neq \mathsf{val}_2$ or $t_1 \neq t_2$. Notice that this is only the first winning condition in the soundness definition of [6].

II OPTIKS's Privacy-Preserving Authenticated History Dictionary (PAHD)

▷ $\mathsf{PAHD.Init}(r)$: The server chooses a random seed and initializes an empty oZKS via $\mathsf{oZKS.Init}$ by giving r as input. The oZKS commitment is returned as the initial commitment and the oZKS initial state is stored in the server's state. The server also initializes the epoch to 0 and stores this in its state.

▷ $\mathsf{PAHD.Upd}(\mathsf{st}_{t-1}, [k_j, v_j]_j)$: The server adds the key-value pairs that are input to the oZKS to create a new commitment to the dictionary. It first checks that all the keys to be updated are unique; if not, it returns $\perp$. In order to differentiate between versions for a key, the server uses the key concatenated with its version number as the oZKS label. We assume that the server keeps track of the version number for each key in its state. Thus, for each key-value pair (k, v), the server first checks if the key already exists in the oZKS. If it does not, the server uses $(k \mid 1)$ as the label. Otherwise, if the key is already at version n, then the server uses $(k \mid n+1)$. Once all the label-value pairs have been formed, the server adds them to the oZKS via $\mathsf{oZKS.Update}$. The server increments the epoch $t-1$ in its state to t, and the resulting oZKS commitment com_t serves as the PAHD commitment for epoch t. The oZKS update proof π^{upd} serves as the PAHD update proof Π_t^{Upd} for epoch t and is stored in the server's state. The server also stores in its state the new oZKS datastore and state.

▷ $\mathsf{PAHD.Lkup}(\mathsf{st}_t, k)$: For a lookup request for key k, the server retrieves from its state the latest oZKS commitment com_t and the latest version number α for k (where $\alpha = 0$ if k is not in the PAHD). If k is in the PAHD, then the server forms labels $(k \mid 1), \ldots, (k \mid \alpha)$ and calls $\mathsf{oZKS.Query}$ for each label to get back $[(\pi_i, v_i, t_i)]_i^\alpha$. To retrieve the non-membership proof $\pi_{\alpha+1}$ for the next version of the key (or to prove that k is not in the dictionary when $\alpha = 0$), the server calls $\mathsf{oZKS.Query}$ for label $(k \mid \alpha + 1)$. The server returns either v_α as the value for k if $\alpha > 0$ or $\perp$ otherwise.

The server returns as its lookup proof:

– **Correct version i is set at epoch t_i:** For each $i \in [1, \alpha]$, π_i serves as the membership proof for $(k \mid i)$ with value v_i and associated epoch t_i in oZKS w.r.t. com_t. This means the server must return $[(\pi_i, v_i, t_i)]_i^\alpha$ as part of the proof.

- **Server could not have shown version $\alpha + 1$:** Proof $\pi_{\alpha+1}$ serves as the non-membership proof for $(k \mid \alpha + 1)$ in oZKS w.r.t. com_t.

▷ $\mathsf{PAHD.VerLkup}(\mathsf{com}_t, k, v, \pi)$: The client verifies each membership proof for labels $(k \mid i)$ for $i \in [1, \alpha]$ and non-membership proof for $(k \mid \alpha + 1)$ w.r.t. com_t via oZKS.Verify. The client also checks that version α is less than the current epoch t, since otherwise this would imply multiple versions were added in the same epoch [1]. We want to preserve a total ordering of key versions and so wish to prevent this from happening. Lastly, the client verifies that the update epochs $t_1, \ldots, t_\alpha$ are monotonically increasing.

▷ $\mathsf{PAHD.Hist}(\mathsf{st}_t, k)$: This algorithm proceeds the same as Lkup, except that in its syntax it explicitly returns all key versions rather than including them in the proof.

▷ $\mathsf{PAHD.VerHist}(\mathsf{com}_t, k, [(v_i, t_i)]_i^n, \Pi^{\mathsf{Ver}})$: This algorithm proceeds identically to that of VerLkup.

▷ $\mathsf{PAHD.Audit}(\mathsf{com}_j, \mathsf{com}_{j+1}, j, j + 1, \Pi_{j+1}^{\mathsf{Upd}})$: The auditor verifies the oZKS update proof in Π_{j+1}^{Upd} via oZKS.VerifyUpd and then checks that $j + 1 \le t$, where t is the current epoch.

III Checkpointing

In this section, we will introduce a deterministic algorithm (ComputeCheckpoints) that does the following: given an integer interval, it outputs a set $O(\log l)$ points within that interval, where k is the size of the integer interval. More precisely, ComputeCheckpoints is a deterministic algorithm, that, given any interval (L, R), $L, R \in \mathbb{N}$, outputs a set of integers $(c_j)_{j=1}^{l}$, where $L \le c_1 < c_2 < \ldots < c_l \le R$ and $l = O(\log |R - L|)$. We give an algorithm for ComputeCheckpoints below. Note that, [11] also proposes a similar algorithm for ComputeCheckpoints. Compared to [11], our checkpoints algorithm produces 2 times as many checkpoints but is easier to analyze.

<u>$\mathsf{ComputeCheckpoints}(L, R)$</u>

- Require that $L \le R, L, R \in \mathbb{N}$
- A compact range, $[(L_i, R_i)]_{i=1}^{m} \leftarrow \mathsf{Compact}(L, R)$ is the minimum set of m subranges that covers (L, R). It is computed by computing canonical coverings of the range (L, R) as described in [9].
- Output $(c_1 = L_1, c_2 = R_1, c_3 = L_2, c_4 = R_2, \ldots, c_{2m-1} = L_m, c_{2m} = R_m)$ as the checkpoints.

For our purpose, we will consider the intervals to be over the epochs of a KT system (as in [9, 11]). We will first provide some observations and then prove an important lemma about intersection of checkpoints of two overlapping intervals.

Observation 1. Consider the binary tree imposed over all epochs of the KT system. Let us call it the *Chronological Tree (*ChronTree*)*. Notice that the points in a given compact range, (L_i, R_i) are exactly the set of all leaves of a full binary subtree within ChronTree of height $log(|R_i - L_i|)$. Therefore, by construction, the checkpoints of a compact range (L_i, R_i) are either the leftmost or the rightmost leaves of induced subtrees of ChronTree.

Observation 2. For any two full binary induced subtrees T_1, T_2 for ChronTree, either $T_1 \subseteq T_2$ or $T_2 \subseteq T_1$ or $T_1 \cap T_2 = \perp$.

Lemma 1. *For any two intervals* $\mathsf{Int}_1 = (L^1, R^1), \mathsf{Int}_2 = (L^2, R^2)$ *such that,* $L^1 \le L^2 \le R^1 \le R^2$*, let* $(c_i)_{i=1}^{n} \leftarrow \mathsf{ComputeCheckpoints}(\mathsf{Int}_1)$ *and* $(c_j)_{j=1}^{m} \leftarrow \mathsf{ComputeCheckpoints}(\mathsf{Int}_2)$*. Then, there exists* $i \in [n], j \in [m]$ *such that* $c_i = c_j$*.*

Proof. Notice that, if either of the endpoints are equal ($L^1 = L^2, L^2 = R^1, R^1 = R^2$), then, by our construction the intervals will share a checkpoint. The non-trivial case is when $L^1 < L^2 < R^1 < R^2$. We will focus on this case for the rest of the proof.

In this case, the intervals intersect at more than one point. Let is denote the intersecting interval as: $\mathsf{Int}_3 = [L^2, R^1]$. There are two possible cases:

1. There exists a $c \in \mathsf{Int}_3$ such that $c \in \mathsf{ComputeCheckpoints}(\mathsf{Int}_1) \cap \mathsf{ComputeCheckpoints}(\mathsf{Int}_2)$.

[1]Since we have very short epochs, this is not a limitation

2. Suppose there is no such c. We will prove, by contradiction, that this case cannot happen.

   Since $L^2 > L^1$, this means, there exists some $(L_x, R_x) \in \mathsf{Compact}(L^1, R^1)$ such that $L_x \leq L^2 \leq R_x$. Note that if either of these constraints is an inequality, then we have an intersecting checkpoint c, since by construction $L_x, R_x \in \mathsf{ComputeCheckpoints}(\mathsf{Int}_1)$ and $L_2 \in \mathsf{ComputeCheckpoints}(\mathsf{Int}_2)$. Thus by assumption we must have $L_x < L^2 < R_x$. Let us denote the induced subtree corresponding to (L_x, R_x) as T_1.

   Let c_y be the largest checkpoint in $\mathsf{ComputeCheckpoints}(\mathsf{Int}_2)$ such that $c_y < R_x$. Note that some compact range of Int_2 has to cover the points on (c_y, R_x); it must be a single range because several ranges would imply another checkpoint between c_y and R_x. Let the corresponding induced subtree be denoted as T_2. This means c_y is has to be the leftmost leaf of T_2, and that the rightmost leaf must be $\geq R_x$. In fact if the rightmost leaf of T_2 was equal to R_x, then that would again give a checkpoint in common, so we conclude that the rightmost leaf must be strictly greater than R_x.

   So now, we have two overlapping induced subtrees of $\mathsf{ChronTree}$, T_1 and T_2 where $T_1 \not\subseteq T_2$ and vice versa. This contradicts **Observation 2**. Thus we arrive at a contradiction.

This concludes the proof. □

Lemma 2. *For any interval (L, R), let $(l^i, r^i)_{i=1}^t$ be arbitrary intervals of (L, R) such that $L = l^1, R = r^t, l^2 = r^1, l^3 = r^2, \ldots, l^t = r^{t-1}$. Let $S \leftarrow \mathsf{ComputeCheckpoints}(L, R), S^i \leftarrow \mathsf{ComputeCheckpoints}(l^i, r^i)$. Then we have, $S \subseteq T$ where $T = \cup_{i=1}^t S^i$.*

Proof. For the sake of contradiction, suppose not. Then there exists at least one checkpoint $c \in S \setminus T$. By our construction, c denotes one endpoint of a compact range in (L, R). Let w be the other endpoint of that range and let T_{cw} denote the corresponding induced subtree of $\mathsf{ChronTree}$.

c has to be covered by some compact range belonging to one of the intervals. Let us denote this interval as (l^*, r^*). Let (a, b) be the compact range of (l^*, r^*) that covers c. Since c is not a checkpoint in (l^*, r^*), this means, $a < c < b$. Let T_{ab} be the induced subtree of $\mathsf{ChronTree}$ for this compact range.

Clearly T_{cw} and T_{ab} have an non-empty intersection, so by Observation 2, it must be the case that either $T_{cw} \subseteq T_{ab}$ or vice versa. If the two are equal, then by construction c would be included in T, and $a < c < b$ means that $T_{ab} \not\subseteq T_{cw}$. Thus we conclude that $T_{cw} \subseteq T_{ab}$. However, note that the entire range $(a, b) \in (L, R)$, so by the construction of $\mathsf{Compact}$, this would imply that T_{cw} should not be included in $\mathsf{Compact}(L, R)$, and instead T_{ab} or one of it's parent trees should be included. This contradicts our assumption that $c \in S$. □

3 A Weakly Consistent Privacy-Preserving Authenticated History Dictionary Construction (wPAHD)

We call the cryptographic primitive that achieves a weakly consistent, auditor-free mode in a KT system a *weakly consistent privacy-preserving authenticated dictionary (*wPAHD*)*. In this section, we give both the API and the instantiation of a wPAHD. Our instantiation uses the Checkpointing algorithm from Section III and the OPTIKS construction that we recalled in Section 2. The main differences between a wPAHD and a PAHD are that: 1) In wPAHD the clients are stateful whereas in PAHD, clients are stateless. 2) wPAHD does not have any global Audit algorithm as opposed to PAHD.

The system is designed to be used as follows. In each epoch, first the users will perform local UpdateClient operations, and the server will combine all those updates and run Upd to update its state and publish an updated commitment. Each user is allowed to update their key at most once per epoch. The users will perform lookup queries, receive results form the server, and record those results in their states via ProcessLkup. At the end of the epoch any user can request to monitor their results. In this case the server will run GetMonitorProof using the set $\mathcal{K}$ including all of the users that this user has ever queried, and the user will verify the resulting proof via Monitor. We do not require that every user perform all of these operations every epoch, however we do require that every user who performed an UpdateClient and/or one or more ProcessLkup operations in a given epoch will perform a Monitor operation at the end of that epoch.

- wPAHD.Init(r) This is the server initialization. This remains the same as PAHD.Init(r) in OPTIKS.
- wPAHD.Upd$(\mathsf{st}_{t-1}, [k_j, v_j]_j)$: This remains the same as PAHD.InitUpd$(\mathsf{st}_{t-1}, [k_j, v_j]_j)$ in OPTIKS.

- $\mathsf{wPAHD.InitClient}(u)$: Initializes client state for a username u as follows. This initializes a table T indexed by usernames. Corresponding to each username (that is not u), there is a list $\mathsf{lookuplist}$ that records the values (and potentially some additional information such as epoch number) corresponding to all the lookup queries the user has made for that username. Table T is initialized with the row $(u, \mathsf{vallist} = \perp)$ which will store the values that the user sets in $\mathsf{UpdateClient}$. T and u are stored in the state st_u. In addition, the following variables are initialized and stored in st_u: $\mathsf{lastMonitored} := -1$, $\mathsf{lastModified} := -1, t_{\mathsf{curr}} = 0$. $\mathsf{lastMonitored}$ will be used to track the last epoch in which the user ran $\mathsf{Monitor}$, $\mathsf{lastModified}$ will track whether the user performed an $\mathsf{UpdateClient}$ or a $\mathsf{ProcessLkup}$ and therefor needs to monitor, and t_{curr} tracks the user's view of the current time.

- $\mathsf{wPAHD.UpdateClient}(\mathsf{st}_u, v, t)$ This algorithm captures when a client updates its own state with a new value, In particular, this algorithm retrieved the row corresponding to u from its table T and receives the current epoch from the server t. It checks that $\mathsf{lastModified} = -1$ and that the $t_{\mathsf{curr}} \leq t$. if not, it stops and returns. Otherwise, it does the following.

  This algorithm appends $\mathsf{vallist} \leftarrow \mathsf{vallist}||(v, t_{\mathsf{curr}})$ and sets $\mathsf{lastModified} \leftarrow t_{\mathsf{curr}}$ and $t_{\mathsf{curr}} = t$.

- $\mathsf{wPAHD.Lkup}(\mathsf{st}_t, k)$: For a lookup request for key k the server retrieves from its state the latest epoch t_{curr} and the latest version number α for k (where $\alpha = 0$ if k is absent). The server returns $v_\alpha, t_{\mathsf{curr}}$. The server returns $v_\alpha = -1$ if α is 0.

- $\mathsf{wPAHD.ProcessLkup}(\mathsf{st}_u, k, v, t)$ This algorithm does the following:
  - Require $k \neq u$.
  - Require $\mathsf{lastModified} \in \{-1, t\}$ and $t \geq t_{\mathsf{curr}}$.
  - If $\mathsf{lastModified} = -1$, set it to $\mathsf{lastModified} \leftarrow t$.
  - Set $t_{\mathsf{curr}} = t$
  - Update the entry corresponding to k in st_u to: $(k, \mathsf{lookuplist} \leftarrow \mathsf{lookuplist}||(v, t))$.

- $\mathsf{wPAHD.GetMonitorProof}(\mathsf{st}_t, u, \mathcal{K}, t, t_{\mathsf{curr}})$ The server runs this algorithm to generate the monitoring proofs for user u. $\mathcal{K}$ is the list of all the other users that user u has performed a Lkup for. It does the following:
  1. Require $t \leq t_{\mathsf{curr}}$ and t_{curr} is the latest epoch.
  2. $(c_j)_{j=1}^m \leftarrow \mathsf{ComputeCheckpoints}(t, t_{\mathsf{curr}})$.
  3. Retrieve the the commitments com_{c_j} for each of the checkpoints.
  4. For all users $k, u \in \mathcal{K}$:
     (a) Retrieve the latest version number α_{c_j} for epoch c_j for k (where $\alpha_{c_j} = 0$ if k is absent). If $\alpha_{c_j} \neq 0$, then the server forms labels $(k \mid 1), \ldots, (k \mid \alpha_{c_j})$ and calls $\mathsf{oZKS.Query}$ for each label to get back $[(\pi_i, v_i, t_i)]_{i=1}^{\alpha_{c_j}}$. To retrieve the non-membership proof $\pi_{\alpha_{c_j}+1}$ for the next version of the key (or to prove that k is not in the dictionary), the server calls $\mathsf{oZKS.Query}$ for label $(k \mid \alpha_{c_j} + 1)$.
     (b) Let $\pi_k = [(\pi_i, v_i, t_i)]_{i=1}^{\alpha_{c_j}}$ for $(c_j)_{j=1}^m$ and t_{curr}.
  5. Return $\pi = \pi_u, \mathcal{K}, \{\pi_k\}_{k \in \mathcal{K}}$

- $\mathsf{wPAHD.Monitor}(\mathsf{st}_u, \pi, \mathsf{lastMonitored}, t)$ The client does the following:
  1. Require that $\mathsf{lastMonitored} \leq t$.
  2. Require that $\mathsf{lastModified} \in \{-1, t\}$
  3. Require that $t_{\mathsf{curr}} \leq t$
  4. $(c_j)_{j=1}^m \leftarrow \mathsf{ComputeCheckpoints}(\mathsf{lastMonitored}, t)$.
  5. Parse $\pi = \pi_u, \mathcal{K}, \{\pi_k\}_{k \in \mathcal{K}}$
  6. For each username $k \in T \cup \{u\}$ (where T is the table in st_u), do the following
     (a) Let the row corresponding to k in T be $\mathsf{lookuplist} = [(v_s, t_s)]_{s=1}^{\beta}$.

(b) Let $\pi_k = ([(\pi_{i,j}, v_{i,j}, t_{i,j})]_{i=1}^{\alpha_{c_j}})_{j=1}^{m}$ be the part of the proof corresponding to user k. If no such proof exists, set $\mathsf{succ} = 0$ and return.

(c) Check that the values and times corresponding to each checkpoint are consistent, i.e. for each $j \in [m-1]$, $[(v_{i,j}, t_{i,j})]_{i=1}^{\alpha_{c_j}}$ is a prefix of $[(v_{i,j+1}, t_{i,j+1})]_{i=1}^{\alpha_{c_{j+1}}}$.

(d) Check that the times in the final checkpoint list are sorted, i.e. for all $i \in [\alpha_{c_m} - 1]$, $0 < t_i < t_{i+1}$.

(e) For each $j \in [m]$ (i.e., for each checkpoint):
   - Check that $t_{\alpha_{c_j},m} \leq c_j$ and that either $\alpha_{c_j} = \alpha_{c_m}$(i.e. all versions were present in epoch c_j) or $t_{\alpha_{c_j}+1,m} > c_j$ (i.e. the next version wasn't added until after c_j).
   - Check each membership proof for labels $(k \mid i)$ for $i \in [1, \alpha_{c_j}]$ and non-membership proof for $(k \mid \alpha_{c_j} + 1)$ w.r.t. com_{c_j} via $\mathsf{oZKS.Verify}$.

(f) if $k \neq u$ Check that the values in $\mathsf{lookuplist}$ are consistent with the final checkpoint list. I.e. for each $s \in [\beta]$:
   - If $t_s < t_{1,m}$, check that $v_s = -1$.
   - If $t_s \geq t_{\alpha_{c_m},m}$, check that $v_s = v_{\alpha_{c_m},m}$.
   - Otherwise, let ℓ be the value such that $t_{\ell,m} \leq t_s < t_{\ell+1,m}$. Check that $v_s = v_{\ell,m}$.

(g) If $k = u$, check that the values in $\mathsf{vallist}$ match those in the final checkpoint list, i.e. $\mathsf{vallist} = \{(v_{i,m}, t_{i,m}\}_{i=1}^{\alpha_{c_m}}$.

7. If all the checks pass, set $\mathsf{lastModified} \leftarrow -1$, $\mathsf{lastMonitored} \leftarrow t$, $t_{\mathsf{curr}} = t$ and set $\mathsf{succ} - 1$. Else, set $\mathsf{succ} = 0$.

4 Security properties of wPAHD

In this section we give a formal definition for soundness for wPAHD.

Soundness. We define the soundness of a wPAHD using a game. We say a wPAHD is sound if for any PPT adversary, the probability of the adversary winning the game is negligibly small. Informally, the game captures the following. Consider Alice with username u_0 and Bob with username u_1. The adversary can cause Alice and Bob to update their own values and to query for each other's values and the values of other users. The adversary can also have the user perform monitoring. We require that in any epoch that Alice perform an update and/or at least one query she must also perform a monitoring, and similarly for Bob. The adversary wins if there is an epoch i where Bob's query for Alice's value is inconsistent with the update operation Alice has performed, with the additional restriction that each of Alice and Bob must perform at least one monitoring operation after epoch i.[2]

1. Initialize the following variables:
   (a) $t_{\mathsf{curr}} = 0$
   (b) $\mathsf{lastMonitored}_0 = \mathsf{lastMonitored}_1 = \mathsf{lastModified}_0 = \mathsf{lastModified}_1 = -1$
   (c) $\mathsf{vcurr}_0 = \mathsf{vcurr}_1 = -1$ (neither user has set their public key yet)
   (d) $\mathsf{vnew}_0 = \mathsf{vnew}_1 = \perp$
   (e) $\mathsf{monitoringTimes}_0 = \mathsf{monitoringTimes}_1 = \perp$
   (f) $\mathsf{StoredV}_0 = \mathsf{StoredV}_1 = \mathsf{LookUpV}_0 = \mathsf{LookUpV}_1 = \perp$
   (g) $\mathsf{Digests} = \perp$
2. $k_0, k_1, \mathsf{st}_{\mathcal{A}} \leftarrow \mathcal{A}$
3. $\mathsf{st}_0 \leftarrow \mathsf{InitClient}(k_0)$, $\mathsf{st}_1 \leftarrow \mathsf{InitClient}(k_1)$
4. $i, i_0, i_1 \leftarrow \mathcal{A}^{\mathsf{OMonitor},\mathsf{OLookUp},\mathsf{OStore},\mathsf{OPublish}}(\mathsf{st}_{\mathcal{A}})$

[2] This requirement is actually slightly more stringent - we require that Alice monitors at least one epoch i_0 and Bob monitors in at least one epoch i_1 such that $i < i_0 \leq i_1$.

5. $\mathcal{A}$ wins if all of the following are true:
   - $i < i_0 < i_1$
   - $\mathsf{LookUpV}_0[i] \neq \mathsf{StoredV}_0[i]$ and $\mathsf{LookUpV}_0 \neq \perp$ (i.e. u_1 did a lookup for u_0 and the result was inconsistent with u_0's $\mathsf{ClientUpdate}$ operations)
   - $i_0 \in \mathsf{monitoringTimes}_0, i, i_1 \in \mathsf{monitoringTimes}_1$.

The oracles are defined as follows:

$\underline{\mathsf{OMonitor}(t, \pi, b)}$

- Require $t \notin \mathsf{monitoringTimes}_b$
- Require $\mathsf{lastModified}_b \in \{-1, t\}$ to ensure that either there have been no lookups/updates since the last monitoring, or that the only looksups/updates were for the same epoch as this monitoring.
- Require $\mathsf{lastMonitored}_b \leq t \leq t_{\mathsf{curr}}$
- $\mathsf{st}'_b, \mathsf{succ} \leftarrow \mathsf{Monitor}(\mathsf{Digests}[t], \mathsf{st}_b, \mathsf{lastMonitored}_b, t)$
- if $\mathsf{succ} = 0$, abort and $\mathcal{A}$ loses the game
- if $\mathsf{vcurr}_b \neq \perp$, then $\forall t' \in \{\mathsf{lastMonitored}_b + 1, \ldots, t\}$ set $\mathsf{StoredV}_b[t'] = \mathsf{vcurr}_b$
- if $\mathsf{vnew}_b \neq \perp$, set $\mathsf{StoredV}_b[t] = \mathsf{vnew}_b$
- Set
  - $\mathsf{st}_b = \mathsf{st}'_b$
  - $\mathsf{lastMonitored}_b = t$
  - $\mathsf{lastModified}_b = -1$
  - $\mathsf{monitoringTimes}_b \leftarrow \mathsf{monitoringTimes}_b || t$
  - if $\mathsf{vnew} \neq \perp$, set $\mathsf{vcurr}_b \leftarrow \mathsf{vnew}_b$ and $\mathsf{vnew}_b \leftarrow \perp$

$\underline{\mathsf{OLookUp}(t, b, k, v)}$ i.e. user k_b performs a lookup for user k

- Require $t \notin \mathsf{monitoringTimes}_b$
- Require $\mathsf{lastMonitored}_b < t \leq t_{\mathsf{curr}}$
- If $\mathsf{lastModified}_b = -1$, set $\mathsf{lastModified}_b = t$.
- Require $\mathsf{lastModified}_b = t$
- Require ($k \neq k_{1-b}$ OR $\mathsf{LookUpV}_{1-b}[t] = \perp$) to ensure that user b does at most one lookup for k_{1-b} in each epoch.
- $\mathsf{st}'_b \leftarrow \mathsf{ProcessLkup}(t, v, \mathsf{st}_b, k_{1-b})$
- Set $\mathsf{st}_b = \mathsf{st}'_b$
- If $k = k_{1-b}$, set $\mathsf{LookUpV}_{1-b}[t] = v$

$\underline{\mathsf{OStore}(t, b, v)}$ i.e. user k_b updates their public key to v

- Require $t \notin \mathsf{monitoringTimes}_b$
- Require $\mathsf{vnew}_b = \perp$
- Require $\mathsf{lastModified}_b = -1$
- Require $\mathsf{lastMonitored}_b < t \leq t_{\mathsf{curr}}$
- Set $\mathsf{lastModified}_b = t$.
- $\mathsf{st}'_b \leftarrow \mathsf{UpdateClient}(v, \mathsf{st}_b, t)$
- Set $\mathsf{st}_b \leftarrow \mathsf{st}'_b$
- Set $\mathsf{vnew}_b \leftarrow v$

$\underline{\mathsf{OPublish}(d)}$

- Set $t_{\mathsf{curr}} = t_{\mathsf{curr}} + 1$
- Set $\mathsf{Digests}[t_{\mathsf{curr}}] = d$

5 Security Proof for our wPAHD construction

In this section, we prove that our construction satisfies the soundness definition.

If the Adv wins the game, it means:

- $\mathsf{LookUpV}_0[i] \neq \mathsf{StoredV}_0[i]$ and $\mathsf{LookUpV}_0 \neq \perp$.
- Let $\mathsf{StoredV}_0[i] = v$. Let $y \leq i$ be the epoch at which v is set, i.e, $\mathsf{StoredV}_0[y] = v$. We have two cases:
  - $y < i$:
    * Let x be the largest entry in $\mathsf{monitoringTimes}_0$ such that $x < i$. Since $y < i$, x exists.
    * Consider all the monitoring intervals between x and i_0 for user u_0. Let S' be the collection of all the checkpoints of all the monitoring intervals between x and i_0. Let $S \leftarrow \mathsf{ComputeCheckpoints}(x, i_0)$. By Lemma 2, $S \subseteq S'$.
    * We already have that $i, i_1 \in \mathsf{monitoringTimes}_1$. There could be multiple entries between i, i_1. Let T' be the collection of all the checkpoints for all the intervals on $\mathsf{monitoringTimes}_1$ such that the intervals exactly cover the interval (i, i_1).
    * Let $T \leftarrow \mathsf{ComputeCheckpoints}(i, i_1)$. By Lemma 2, $T \subseteq T'$.
    * Now we have two overlapping intervals, (x, i_0) and (i, i_1) (since $x \leq i \leq i' \leq i_1$. Therefore, by Lemma 1, $S \cap T \neq \perp$. Let $c_{\mathsf{intersection}}$ be a checkpoint in the intersection. Note that, since the checkpoints are completely contained within the intervals, we have, $c_{\mathsf{intersection}} \geq i$. Moreover, $c_{\mathsf{intersection}}$ must be a member of S' and T' since they are supersets of S and T respectively.
  - $y = i$: In this case, we have $i \in \mathsf{monitoringTimes}_0$. Thus, we have two intersecting intervals (i, i_0) (for u_0) and (i, i_1) (for u_1). Notice that these intervals might have sub intervals, but, since by Lemma 2 the checkpoints of the larger intervals will be subset of the checkpoints of the smaller sub-intervals, we only consider the larger intervals (as in the previous case). Since our checkpoining algorithm always includes the endpoints of the intervals as checkpoints, we can guarantee that the two intersections have at least one shared checkpoint $c_{\mathsf{intersection}} = i$.
- Observe that since $\mathsf{LookUpV}_0[i] = v^* \neq \perp$, $\mathsf{OLookUp}(i, 1, k_0, v^*)$ was called, which in turn means in epoch i, user u_1 ran $\mathsf{ProcessLkup}(\mathsf{st}_1, k_1, v^*, i)$ and added (v^*, i) to its table entry for u_0.

- Case 1: when $\mathsf{StoredV}_0[i] = v \neq -1$. This implies that there was at least one OStore operation at or before epoch i. Let i_{StoredV} be the lastest epoch at or before i where an $\mathsf{OStore}(i_{\mathsf{StoredV}}, 0, v')$ was called for some v', and note that by the definition of OStore and $\mathsf{OMonitor}$, it must be the case that $v' = v$. By our construction, this means that $(v, i_{\mathsf{StoredV}})$ was added to user u_0's $\mathsf{vallist}$ in epoch i_{StoredV}. Moreover, since this was the latest such epoch, this means that there is never any epoch t in any entry in the $\mathsf{vallist}$ with $i_{\mathsf{StoredV}} < t \leq i$.

  Now, consider the intersection checkpoint $c_{\mathsf{intersection}}$ discussed above; for each of u_0, u_1 there will be some monitoring at or after epoch i which will include $c_{\mathsf{intersection}}$ as one of the checkpoint epochs. Moreover, by the logic above, this means that when those monitoring events occur, u_0's $\mathsf{vallist}$ will contain $(v, i_{\mathsf{StoredV}})$, and no other entry with $i_{\mathsf{StoredV}} < t \leq i$, while u_1's table will contain an entry (v^*, i) for u_0 where $v^* \neq v$.

  Assuming the monitor succeeds, the former means that user u_0's monitoring proof will include a membership proof for $(k_0||\alpha, v, i_{\mathsf{StoredV}})$ with respect to $\mathsf{com}_{c_{\mathsf{intersection}}}$ for some version α. Since the monitor algorithm verifies that the insertion-epochs are increasing, this also implies that u_0's monitoring proof includes membership proofs for $(k_0||a)$ with epoch $< i_{\mathsf{StoredV}}$ for all $a \in [\alpha - 1]$ and either a non-membership proof for $(k_0||\alpha + 1)$ or a proof for $(k_0||\alpha + 1)$ with insertion epoch $> i$.

  Assuming the monitoring succeeds, the latter means that u_1's monitoring proof will include a membership proof with respect to $\mathsf{com}_{c_{\mathsf{intersection}}}$ for $(k_0||\alpha^*, v^*, i^*)$ for some version α^* and $i^* \leq i$. Since the monitor algorithm verifies that the insertion-epochs are increasing, this also implies that u_1's monitoring proof includes membership proofs for $(k_0||a)$ with epoch $< i^*$ for all $a \in [\alpha^* - 1]$ and either a non-membership proof for $(k_0||\alpha^* + 1)$ or a membership proof for $(k_0||\alpha^* + 1)$ with insertion epoch $> i$.

  If $\alpha = \alpha^*$, this clearly breaks oZKS soundness.

  If $\alpha < \alpha^*$, then as described above, u_0's monitoring proof must contain either a non-membership proof for $(k_0||\alpha + 1)$ or a proof for $(k_0||\alpha + 1, v_{i'}, i')$ with $i' > i$, where $\alpha + 1 \leq \alpha^*$. However, by the logic above u_1's monitoring proof must contain a membership proof for $(k_0||\alpha + 1)$ with an insertion epoch $< i^* \leq i$. This this breaks oZKS soundness.

  The case where $\alpha > \alpha^*$ follows similarly.

- Case 2: If $\mathsf{StoredV}_0[i] = -1$, this implies that the first OStore operation occurs after epoch i. This means that user u_0's $\mathsf{vallist}$ does not ever contain any epoch $t \leq i$. Then, when u_0 later performs monitoring w.r.t. checkpoint $c_{\mathsf{intersection}}$, if monitoring succeeds, the proof must contain either a non-membership proof for $(k_0||1)$ or a membership for $(k_0||1)$ with insertion epoch $> i$. However, by the logic above, u_1's monitoring proof must include a membership proof for $(k_0||1)$ with insertion epoch $\leq i$. This breaks soundness of the oZKS.

6 Acknowledgements

The authors would like to thank Kevin Lewi, Brendan McMillion and Kim Laine for their valuable feedback and suggestions.

References

[1] Advancing iMessage security: iMessage contact key verification. https://security.apple.com/blog/imessage-contact-key-verification/ (accessed: 2024-01-08).

[2] What is key transparency? https://proton.me/support/key-transparency (accessed: 2024-01-08).

[3] M. Chase, A. Deshpande, E. Ghosh, and H. Malvai. Seemless: Secure end-to-end encrypted messaging with less trust. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security CCS*. ACM, 2019.

[4] Google. Google key transparency. https://github.com/google/keytransparency/blob/master/docs/meet-in-the-middle.md (accessed: 2024-05-17), 2020.

[5] J. Len, M. Chase, E. Ghosh, D. Jost, B. Kesavan, and A. Marcedone. ELEKTRA: efficient lightweight multi-device key transparency. In W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 2915–2929. ACM, 2023.

[6] J. Len, M. Chase, E. Ghosh, K. Laine, and R. C. Moreno. OPTIKS: An optimized key transparency system. Cryptology ePrint Archive, Paper 2023/1515. http://eprint.iacr.org/2023/1515, 2023.

[7] H. Malvai, L. Kokoris-Kogias, A. Sonnino, E. Ghosh, E. Oztürk, K. Lewi, and S. F. Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[8] B. McMillion. Key transparency. https://bren2010.github.io/draft-key-transparency/draft-mcmillion-key-transparency.html (accessed: 2024-05-17), 2023.

[9] S. Meiklejohn, P. Kalinnikov, C. S. Lin, M. Hutchinson, G. Belvin, M. Raykova, and A. Cutter. Think global, act local: Gossip and client audits in verifiable data structures, 2020.

[10] A. Melnikov and R. Mahy. IETF Key Transparency (draft charter). https://datatracker.ietf.org/doc/charter-ietf-keytrans/ (accessed: 2023-06-02).

[11] N. Tyagi, B. Fisch, A. Zitek, J. Bonneau, and S. Tessaro. Versa: Verifiable registries with efficient client audits from rsa authenticated dictionaries. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2793–2807, 2022.

[12] I. Tzialla, A. Kothapalli, B. Parno, and S. Setty. Transparency dictionaries with succinct proofs of correct operation. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2022.

[13] Whatsapp.com. Whatsapp encryption overview. White Paper – https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf (accessed: 2022-08-03), 2021.