Revisiting User Privacy for Certificate Transparency[⋆]

Daniel Kales, Olamide Omolola, and Sebastian Ramacher

Graz University of Technology, Graz, Austria
firstname.lastname@iaik.tugraz.at

Abstract. Public key infrastructure (PKI) based on certificate authorities is one of the cornerstones of secure communication over the internet. Certificates issued as part of this PKI provide authentication of web servers among others. Yet, the PKI ecosystem is susceptible to certificate misissuance and misuse attacks. To prevent those attacks, Certificate Transparency (CT) facilitates auditing of issued certificates and detecting certificates issued without authorization. Users that want to verify inclusion of certificates on CT log servers contact the CT server directly to retrieve inclusion proofs. This direct contact with the log server creates a privacy problem since the users' browsing activities could be recorded by the log server owner.

Lueks and Goldberg (FC 2015) suggested the use of Private Information Retrieval (PIR) in order to protect the users' privacy in the CT ecosystem. With the immense amount of certificates included on CT log servers, their approach runs into performance issues, however. Nevertheless, we build on this approach and extend it using multi-tier Merkle trees, and render it practical using multi-server PIR protocols based on distributed point functions (DPFs). Our approach leads to a scalable design suitable to handle the increasing number of certificates and is, in addition, generic allowing instantiations using secure accumulators and PIRs.

We implement and test this mechanism for privacy-preserving membership proof retrieval and show that it can be integrated without disrupting existing CT infrastructure. Most importantly, even for larger CT logs containing 2^{31}certificates, our approach using sub-accumulators can provide privacy with a performance overhead of less than 9 milliseconds in total.

Keywords: Certificate Transparency · User Privacy · Private Information Retrieval

[⋆] This is the full version of a paper to appear at 2019 4th IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17–19, 2019. © IEEE, 2019.

1 Introduction

Nowadays Transport Layer Security (TLS) [Res18] is the de-factor standard for secure communication over the internet. In general, TLS enables two parties—a client and a server—to agree on a shared secret key which can then be used to encrypt payload data. During the handshake that is responsible to perform the key agreement, the client most commonly verifies the server's identity based on the server's X.509 certificate [CSF+08] issued by some trusted certificate authority (CA). However, in the standard certificate ecosystem, there is still room for misuse, as multiple certificates may be issued for the same domain name. The most prominent examples of such incidents include CAs like Comodo[1] or DigiNotar[2] issuing certificates for, among others, subdomains of `google.com`. In the latter case of DigiNotar, these fraudulent certificates were used for man-in-the-middle attacks against users.

To this end, countermeasures like Certificate Transparency (CT) [Lau14, LLK13, DGHS16] have received a lot of attention recently. In CT, all issued TLS certificates are publicly logged. Its goal is to allow any party to audit the public log and find suspicious certificates or check the integrity of the log itself. The ultimate goal of CT is to eventually have clients refuse connections for certificates that are not included in a public log. Google began enforcing this policy for certificates issued after April 2018[3] in its browser. Also, other big browser vendors such as Apple and Mozilla are in the process of enforcing this policy.

In a logging system, web clients need to ensure that log servers do not hand out promises of certificate inclusion in the log without actually doing so. To combat misbehaving log servers, web clients act as auditors, verifying that any certificates they received are actually publicly logged. Although this is an important role, it has negative privacy implications for clients performing such an auditing role, as verifying the inclusion of a certificate reveals the browsing behavior of the client to the log server.

1.1 Overview of the Certificate Transparency Ecosystem

Certificate Transparency is a large ecosystem with many participants. First, there are so-called *submitters*, who submit certificates or precertificates[4] to the log server and receive a Signed Certificate Timestamp (SCT). An SCT is a log server's promise that the certificate it was issued for will be included in the log

[1] https://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html

[2] https://security.googleblog.com/2011/08/update-on-attempted-man-in-middle.html

[3] https://www.section.io/blog/chrome-ct-compliance/

[4] Precertificates are certificates provided to the log before the issuance of the actual certificate. They contain a special critical poison extension that renders the certificate unusable in TLS connections.

server after a particular time period called Maximum Merge Delay (MMD). This SCT is then either directly included in the certificate or exchanged with a web client during the TLS handshake. Submitters are usually CAs, but, in general, anyone can submit certificates to the log server.

Log servers receive certificate chains and issue SCTs for them. They add those chains (together with the respective SCTs) to a data structure which allows to store elements and to later produce succinct witnesses to attest the membership of certain values within this data structure. Conceptually, such a data structure realizes what is formalized by cryptographic accumulators (see [DHS15] for an overview). Technically, it is realized using Merkle trees [Mer89].

The role of *monitors* is to watch the log servers and audit their behavior by verifying the validity and consistency of the accumulator over time. Monitors also can check for misissued certificates and alert domain owners when they detect a potentially malicious certificate in the log.

Finally, there are so-called *auditors*, who verify the SCTs' signature and check that the accompanying certificate is present in the log by requesting a witness attesting their membership in the accumulator and verifying it against the current accumulator value. They additionally can request a proof of consistency with respect to changes in the accumulator over time. An auditor is an essential part of a TLS client, but it could also be a secondary function of a monitor. Auditors built into TLS clients do not necessarily perform this inclusion checks in real time when visiting a website; usually, only the SCT signatures are verified against the public keys of trusted log servers. The Merkle-tree inclusion proofs are then retrieved asynchronously at a later time to audit if the log server is well-behaved. Chromium-based browsers already have a built-in component that validate SCTs by sending them to a Google resolver that validates inclusion proofs. A misbehaving log server, issuing SCTs for certificates that are not logged by the server, will then get reported to big browser vendors and will subsequently be removed from the list of trusted log servers.

Current State of the Certificate Transparency Ecosystem Due to the efforts of big browser vendors, especially Google, the CT ecosystem is growing rapidly. As of August 2018, several of the big CT log servers (e.g., Google Argon) have more than 250 million certificates in their log. Even the smaller log servers have more than 10 million active certificates, with a certificate issuance rate of $\approx$ 53000 new certificates per hour. Two months later (October, 2018), the certificate issuance rate jumped to $\approx$ 105000 new certificates per hour and several of the big CT log servers (e.g., Google Argon) have more than 400 million certificates in their log servers. Cloudflare's CT statistics website[5] publishes live statistics about the current state of CT log servers.

[5] https://merkle.town/

A big factor in these numbers is the growing popularity of Let's Encrypt,[6] a free, automated certificate authority that accounts for more than 72% of all certificates in CT logs (cf. Table 1) This huge number of certificates makes the use of simple privacy-preserving techniques, such as simply downloading the full log, impossible in practice.

Root CA	Certificates	Percentage
DigiCert	64,226,041	5%
Let's Encrypt	941,016,262	72%
Sectigo	246,484,842	19%
Other	62,114,615	5%

Table 1. Number of certificates per CAs tracked as part of CT. Numbers are based on data from https://merkle.town as of April 8th, 2019.

1.2 Privacy Challenges with CT

The auditors' role in CT is essential because they verify that a log server did not issue an SCT for a certificate that is not included in the public log after the MMD. However, this vital process of auditing SCTs in CT can violate a user's privacy if the auditor is, e.g., a TLS client.

The CT auditor checks that the corresponding certificate of each valid SCT is included in the log server. This is done by requesting a membership proof for the certificate hash from the log server and verifying it against the accumulator value (the Signed Tree Hash (STH)) of the log server. The downside to this approach is that it reveals the browsing behavior of the specific auditor (which is usually a TLS client) to the log server because having a particular SCT means that the auditor visited this website. A malicious log server can choose to record this browsing behavior and sell this browsing history to interested third parties, like advertising agencies.

On the other hand, the privacy problem stated above can also weaken the integrity of the CT ecosystem, since TLS clients are discouraged to audit sites which they may not want to be associated with, e.g., sites of political, religious or sexual nature. In turn, if no-one is auditing the validity of SCTs for these certificates, they can more likely become targets for adversaries, as they could convince a malicious log server to issue an SCT for a malicious certificate and not include it into its log. If a potential victim of a man-in-the-middle attack using this malicious certificate is not likely to audit the SCT because he does not

[6] https://letsencrypt.org/

want his browsing behavior known, this attack is much more likely to succeed unnoticed.

Furthermore, the privacy problem is transferred to applications or protocols that use CT as a basis or follow the same architectural design. One such application is DECIM [YRC18], which aims to detect the compromise of endpoints in messaging scenarios. DECIM provides a key management protocol based on CT and enables users to refresh and manage keys in a transparent manner. Users of this system query keys from log servers and can thereby leak their communication partners. Thus, the authors of DECIM suggest the use of spoof queries over an anonymous channel such as Tor to hide the actual user queries from the log servers. Our proposed solutions can also be directly applied to DECIM.

1.3 Our Contribution

In this work, we tackle the privacy issues within the Certificate Transparency ecosystem. Our contributions are as follows:

– We build on top of Lueks and Goldberg's approach [LG15] for privacy-preserving retrieval of inclusion proofs from CT log servers. To achieve privacy there, clients fetch inclusion proofs using a multi-server private information retrieval (PIR) protocol. We, however, present a more scalable design for logging a huge number of certificates, which allows us to include small static partial inclusion proofs in an SCT, a server's certificate or as a TLS extension. The client can then check the inclusion based on the partial proof and by fetching the missing parts of the proof using a PIR-based approach.

– We verify the practicality of our approach by extending Google's CT log server implementation and performing experiments on realistic log server sizes. Even without using the approach of sub-trees, we report practical performance numbers and improve both runtime and communication compared to previous approaches. For our multi-tier approach, we report a client runtime overhead of less than a millisecond in total, a server runtime overhead of less than 9 milliseconds, and total communication overhead of around 7 KB for 2^{31} certificates when using hourly sub-trees, and even below 1 milliseconds for clients and servers for sub-trees accumulating certificates per minute.

Specifically, our goal is to tackle the privacy issue without any changes to the TLS server side to ease the possibility of a fast deployment. In our approach, we split the Merkle tree containing all certificates into multiple tiers of smaller Merkle trees where the trees at the bottom contain certificates. This split can, for example, be based on a parameterizable time interval or a maximum number of certificates. The sub-trees, respectively their roots, are then combined into the larger tree containing all certificates. This separation of the certificates into smaller sub-trees then allows us to embed membership proofs concerning the sub-trees in an extension field of the SCT or as an X.509v3 extension [CSF+08] into the certificate itself. As the height of the larger tree is now considerably smaller than a single tree containing all certificates, the approach by Lueks and Goldberg [LG15] using PIR to fetch the membership proofs, becomes practical again.

We formalize our approach in more general terms using accumulators and sub-accumulators, where we consider the smaller sub-trees as accumulators and then the full tree as an accumulator of accumulators. Using this abstraction, we discuss different types of accumulators including Merkle-tree accumulators as well as RSA and bilinear pairing based ones in terms of their performance characteristics as well as their consequences on the security on the CT ecosystem.

Additionally, we use a different two-server PIR solution as an alternative to the PIR scheme used by Lueks and Goldberg. We make use of the work on distributed point functions by Gilboa and Ishai [GI14] to build an efficient two-party computationally secure PIR system and present a highly performant implementation. For this, the client needs to know the index i of the item it wants to retrieve in the database. At the moment, there no such index exists in the SCTs that are returned by the log server. Therefore, we propose to include such an index in the `CtExtensions` field of an SCT. Alternatively, this static piece of information can also be included in a TLS extension.

Finally, our approach is general and can also be applied to other systems based on CT-like architectures. In particular, it could be used to replace the spoof queries over Tor as proposed in DECIM for hiding a user's communication partners.

1.4 Related Work

Different approaches have been proposed to solve various privacy issues in the context of CT, and we discuss some of them below as well as known privacy-preserving techniques.

Tor [DMS04] and AN.ON [BFK00], two open and privacy-enhancing networks can provide the needed infrastructure to solve the privacy problem in CT. In both networks, the client requesting an inclusion proof is anonymous through a series of complex routing mechanisms. However, Tor suffers from unpredictable performance and AN.ON has limited bandwidth and has no load balancing mechanisms [WHF07].

Another suggestion allows the clients to receive the inclusion proofs using special DNS records through their DNS resolvers [Lau16]. In this case, the log server operates DNS name servers which serve authoritative answers to special queries from web clients. One of the pitfalls [Mes17] of this approach is that the browsing history is still observable since DNS requests are mostly sent in plaintext over UDP.

The draft of version 2 of the CT RFC [LLK$^+$18] discusses the privacy issues in CT and presents three mechanisms to retrieve the Merkle inclusion proofs in a privacy-preserving way. The first mechanism involves a new TLS extension where the TLS servers send the inclusion proofs and SCTs in the process of communication with the client. The inclusion proofs and the SCTs can be updated on the fly in this case. This approach puts additional load on the server, i.e., the server has to continually update the inclusion proofs it has in storage. The second mechanism involves the Online Certificate Status Protocol (OCSP) [SMA+13]. A user contacts the OCSP service of a certificate authority to check whether a certificate was revoked, thus leaking the browsing behavior to the CA. The OCSP can also be used in this manner to deliver inclusion proofs to the client. However, OCSP does not solve the privacy problem but simply shifts the information leakage to a certificate authority. OCSP stapling [III11], which was initially designed to offload computational costs to the servers, also helps to address privacy issues, since the client no longer needs to contact the CA themselves, but verifies the time-stamped OCSP response appended by the server to the initial TLS handshake. The OCSP stapling approach also adds additional load on the server because the time-stamped OCSP response has to be continuously changed and updated. The final mechanism involves adding the inclusion proofs and the SCTs directly as an X509v3 certificate extension. This extension can not regularly be updated, and while it provides privacy, it quickly de-synchronizes with the log servers.

Lueks and Goldberg [LG15] propose to store membership proofs in a PIR database optimized for multi-user queries. The database stores a record containing the membership proof for each certificate; thus for storing 2^ℓ certificates, the database is required to store $\ell \cdot 2^\ell$ hashes. For log servers storing a million of certificates, the performance is reasonable; however, current CT log servers contain a hundred times more certificates than assumed by Lueks and Goldberg, rendering their approach impracticable.

Eskandarian et al. [EMBB17] address another privacy issue, which is not the focus of this work. In case a misbehaving log server is identified, an auditor is required to publish the offending SCT to indicate the log server's misbehavior. Naturally, the incident together with the SCT would then be reported to browser vendors managing the list of trusted log servers. However, this again leaks the client's browsing behavior to a third party. Eskandarian et al. tackle this issue by constructing zero-knowledge proofs of exclusion, proving that an SCT has been excluded from a log whereas the verifier only learns that an entry has been excluded. Their techniques fundamentally rely on efficient proofs of knowledge of signatures together with suitable signature schemes for signing timestamps.

A recent proposal [NGR18] addresses the issues related to gossiping in Certificate Transparency. Gossiping is the sharing of information about log servers between clients. The authors propose three protocols for gossiping SCTs and Signed Tree Heads (STHs) amongst web clients. The protocols necessitate the exchange of sensitive information that can be used to aggregate network activities of different clients or track clients across different origins. The authors proposed measures to ensure that the possibility of such a privacy breach is minimal. However, the protocols and the privacy measures of this draft focus primarily on the gossiping protocols and do not address the privacy concerns that come with the fetching of inclusion proofs.

Demmler et al. [DRRT18] use a PIR based on distributed point functions (DPF) to construct a multi-server private set-intersection protocol optimized for unbalanced set sizes. In addition to a performant implementation, they also touch on deployment considerations, which we also discuss in Section 5.4.

Splinter [WYG+17], a system built on Function Secret Sharing and DPFs, provides privacy for users querying a public database. The query from a user is split and sent to multiple servers that have a copy of the same data. Splinter cannot only retrieve data in a PIR-like fashion but also enables a user to compute functions such as MAX or TOPK over ranges of the public data without non-colluding servers learning any information about the query.

A different approach to PIR is called oblivious RAM (ORAM), where a client can read and write to a database stored on a server without the server learning about the location or content of the reads and writes. The original work of Goldreich [Gol87] has spawned an extensive line of work for different ORAM constructions and improvements, a recent example being [SvDS+18]. However, while ORAM is a more powerful primitive than PIR, it is not well suited for the scenario of CT, since the database is read-only and public, with many different clients wanting to retrieve data.

A different line of work investigates privacy-preserving key directories, which are similar to the logging infrastructure used in CT but additionally hide the contents of the key directory. Examples of such systems include CONIKS [MBB+15], EthIKS [Bon16], Catena [TD17], and the generalization of *Verifiable Key Directories* by Chase et al. [CDG18].

2 Preliminaries

In this section, we introduce cryptographic primitives and constructions that we subsequently use as building blocks. Notation-wise, let $[n] := \{1, \ldots, n\}$ for $n \in \mathbb{N}$. For an algorithm $\mathcal{A}$, we write $\mathcal{A}(\cdots; r)$ to make the random coins explicit. We say that an algorithm is efficient, if it runs in probabilistic polynomial time (PPT).

2.1 Accumulators

We rely on the formalization of accumulators by Derler et al. [DHS15]. Based on this formalization, we then state the Merkle tree, the RSA, and the bilinear accumulators within this framework. We start with the definition of a static accumulator.

Definition 1 (Static Accumulator). *A static accumulator is a tuple of efficient algorithms* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{WitCreate}, \mathsf{Verify})$ *which are defined as follows:*

$\mathsf{Gen}(1^\kappa, t)$: *This algorithm takes a security parameter* κ *and a parameter* t*. If* $t \neq \infty$*, then* t *is an upper bound on the number of elements to be accumulated. It returns a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, where* $\mathsf{sk}_\Lambda = \emptyset$ *if no trapdoor exists. We assume that the accumulator public key* pk_Λ *implicitly defines the accumulation domain* D_Λ*.*

$\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$ *and a set* $\mathcal{X}$ *to be accumulated and returns an accumulator* $\Lambda_\mathcal{X}$ *together with some auxiliary information* aux*.*

$\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x_i)$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, an accumulator* $\Lambda_\mathcal{X}$*, auxiliary information* aux *and a value* x_i*. It returns* $\bot$*, if* $x_i \notin \mathcal{X}$*, and a witness* wit_{x_i} *for* x_i *otherwise.*

$\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_{x_i}, x_i)$: *This algorithm takes a public key* pk_Λ*, an accumulator* $\Lambda_\mathcal{X}$*, a witness* wit_{x_i} *and a value* x_i*. It returns* 1 *if* wit_{x_i} *is a witness for* $x_i \in \mathcal{X}$ *and* 0 *otherwise.*

We now define a dynamic accumulator, but adapt it to our use-case. We only allow additions of elements to the accumulator.

Definition 2 (Dynamic Accumulator). *A dynamic accumulator is a static accumulator with an additional tuple of efficient algorithms* $(\mathsf{Add}, \mathsf{WitUpdate})$ *which are defined as follows:*

$\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)$: *This deterministic algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, an accumulator* $\Lambda_\mathcal{X}$*, auxiliary information* aux*, as well as an element* x *to be added. If* $x \in \mathcal{X}$*, it returns* $\bot$*. Otherwise, it returns the updated accumulator* $\Lambda_{\mathcal{X}''}$ *with* $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$ *and updated auxiliary information* aux'*.*

$\mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)$: *This algorithm takes a key pair* $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$*, a witness* wit_{x_i} *to be updated, auxiliary information* aux *and an* x *which was added to the accumulator. It returns an updated witness* wit'_{x_i} *on success and* $\bot$ *otherwise.*

Note that the formalization of accumulators by Derler et al. gives access to a trapdoor if it exists. Giving those algorithms access to the trapdoor can often be beneficial performance-wise, but requires additional trust assumptions. We will discuss the consequences for instantiating our schemes in Section 3.3.

Finally, we recall the notion of collision freeness:

Definition 3 (Collision Freeness). *A cryptographic accumulator is* collision-free*, if for all PPT adversaries* $\mathcal{A}$ *there is a negligible function* $\varepsilon(\cdot)$ *such that:*

$$\Pr\left[\begin{array}{l}(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda) \leftarrow \mathsf{Gen}(1^\kappa, t), \\ (\mathsf{wit}^\star_{x_i}, x^\star_i, \mathcal{X}^\star) \leftarrow \mathcal{A}^\mathcal{O}(\mathsf{pk}_\Lambda)\end{array} : \begin{array}{r}\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda^\star, \mathsf{wit}^\star_{x_i}, x^\star_i) = 1 \\ \wedge\ x^\star_i \notin \mathcal{X}^\star\end{array}\right] \leq \varepsilon(\kappa),$$

where $\Lambda^\star \leftarrow \mathsf{Eval}_{r^\star}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X}^\star)$ *and the adversary gets access to the orcales*

$$\mathcal{O} = \{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot), \mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot)\}$$

and, if the accumulator is dynamic, additionally to

$$\{\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot), \mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \cdot, \cdot, \cdot)\}.$$

2.2 Merkle-tree Accumulator

In Scheme 1, we cast the Merkle-tree accumulator in the framework of [DHS15]. Correctness can easily be verified. We restate the well-known fact that this accumulator is collision free.

$\underline{\mathsf{Gen}(1^\kappa, t)}$: Fix a family of hash functions $\{H_k\}_{k \in \mathsf{K}^\kappa}$ with $H_k : \{0,1\}^* \rightarrow \{0,1\}^\kappa \; \forall \, k \in \mathsf{K}^\kappa$. Choose $k \xleftarrow{R} \mathsf{K}^\kappa$ and return $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda) \leftarrow (\emptyset, H_k)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as H_k and $\mathcal{X}$ as $(x_0, \ldots, x_{n-1})$. If $\nexists \, k \in \mathbb{N}$ so that $n = 2^k$ return $\perp$. Otherwise, let $\ell_{u,v}$ refer to the u-th leaf (the leftmost leaf is indexed by 0) in the v-th layer (the root is indexed by 0) of a perfect binary tree. Return $\Lambda_{\mathcal{X}} \leftarrow \ell_{0,0}$ and $\mathsf{aux} \leftarrow ((\ell_{u,v})_{u \in [n/2^{k-v}]})_{v \in [k]}$, where

$$\ell_{u,v} \leftarrow \begin{cases} H_k(\ell_{2u,v+1} || \ell_{2u+1,v+1}) & \text{if } v < k \text{, and} \\ H_k(x_i) & \text{if } v = k. \end{cases}$$

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_{\mathcal{X}}, \mathsf{aux}, x_i)}$: Parse aux as $((\ell_{u,v})_{u \in [n/2^{k-v}]})_{v \in [k]}$ and return wit_{x_i} where

$$\mathsf{wit}_{x_i} \leftarrow (\ell_{\lfloor i/2^v \rfloor + \eta, k-v})_{0 \leq v \leq k}, \text{ where } \eta = \begin{cases} 1 & \text{if } \lfloor i/2^v \rfloor \pmod 2 = 0 \\ -1 & \text{otherwise.} \end{cases}$$

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_{\mathcal{X}}, \mathsf{wit}_{x_i}, x_i)}$: Parse pk_Λ as H_k, $\Lambda_{\mathcal{X}}$ as $\ell_{0,0}$, set $\ell_{i,k} \leftarrow H_k(x_i)$. Recursively check for all $0 < v < k$ whether the following holds and return 1 if so. Otherwise return 0.

$$\ell_{\lfloor i/2^{v+1} \rfloor, k-(v+1)} = \begin{cases} H_k(\ell_{\lfloor i/2^v \rfloor, k-v} || \ell_{\lfloor i/2^v \rfloor + 1, k-v}) & \text{if } \lfloor i/2^v \rfloor \pmod 2 = 0 \\ H_k(\ell_{\lfloor i/2^v \rfloor - 1, k-v} || \ell_{\lfloor i/2^v \rfloor, k-v}) & \text{otherwise.} \end{cases}$$

Scheme 1: Merkle-tree accumulator.

Lemma 1. *If* $\{H_k\}_{k \in \mathsf{K}^\kappa}$ *is a family of collision resistant hash functions, the static accumulator in Scheme 1 is collision free.*

In the current CT log server implementation, H_k is instantiated using SHA-256. Also, in practical instantiation, the requirement that Eval only works on sets of a size that is a power of 2 can be dropped. It is always possible to repeat the last element until the tree is of the correct size.

2.3 Dynamic Public-Key Accumulators

Besides hash based-based constructions, major lines of work investigated accumulators in the hidden order groups, i.e. RSA-based, and the known order groups, i.e. discete logarithm-based, setting. The first collision-free RSA-based accumulator is due to Barić and Pfitzmann [BP97]. The accumulator in this construction consists of a generator raised to the product of all elements of the set. Then witnesses essentially consist of the same value skipping the respective elements in the product. Thereby, the witness can easily be verified by raising the power of the withness to the element and checking if result matches the accumulator. We recall the RSA-based accumulator in Scheme 2. Note however, that we define $\mathsf{WitCreate}$ in a way that does not require the factorization of N, i.e. no secret key is required. Correctness can easily verified, and collision freeness follows from the strong RSA assumption:

$\underline{\mathsf{Gen}(1^\kappa, t)}$: Fix a hash functions H with $H : \{0,1\}^* \rightarrow \mathbb{P}$. Choose an RSA modulus $N = p \cdot q$ with two large safe primes p, q, and let g be a random quadratic residue mod N. Set $\mathsf{sk}_\Lambda \leftarrow \emptyset$ and $\mathsf{pk}_\Lambda \leftarrow (N, g, H)$

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as (N, g, H). Return $\Lambda_\mathcal{X} \leftarrow g^{\prod_{x \in \mathcal{X}} H(x)} \mod N$ and $\mathsf{aux} \leftarrow \mathcal{X}$.

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Return $\mathsf{wit}_x \leftarrow g^{\prod_{x' \in \mathcal{X} \setminus \{x\}} H(x')} \mod N$.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Parse pk_Λ as (N, g, H). If $\mathsf{wit}_x^{H(x)} = \Lambda_\mathcal{X} \mod N$ holds, return 1, otherwise return 0.

$\underline{\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g, H) and aux as $\mathcal{X}$. Set $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$, $\mathsf{aux}' \leftarrow X'$, and $\Lambda_{\mathcal{X}'} \leftarrow \Lambda_{\mathcal{X}'}^{H(x)} \mod N$. Return $\Lambda_{\mathcal{X}'}$ and aux'.

$\underline{\mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)}$: Parse pk_Λ as (N, g, H). Return $\mathsf{wit}_{x_i}^{H(x)} \mod N$.

Scheme 2: RSA-based accumulator.

Lemma 2 (**[BP97]**). *If the strong RSA assumption holds, Scheme 2 is collision-free.*

Additionally, we recall the t-SDH-based accumulator from Nguyen [Ngu05]. The idea here is to encode the accumulated elements in a polynomial. This polynomial is then evaluated for a fixed element and the result is randomized to obtain the accumulator. Similar to the RSA-based accumulator, a witness consists of the evaluation of the same polynomial with the term corresponding to the respective element cancelled out. For verification a pairing is used to check whether the polynomial encoded in the witness is a factor of the one encoded in the accumulator. The scheme is depicted in Scheme 3. Again we define the accumulator in a way that no secret key, i.e. s, is required. Correctness is again obvious, whereas collision freeness follows from the t-SDH assumption:

$\underline{\mathsf{Gen}(1^\kappa, t)}$: Let $\mathbb{G}$ be a prime order group p generated by g with a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. Choose $s \in \mathbb{Z}_p^*$ and return $\mathsf{sk}_\Lambda \leftarrow \emptyset$ and $\mathsf{pk}_\Lambda \leftarrow (\mathbb{G}, e, (g^{s^i})_{i=0}^t)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as $(\mathbb{G}, e, (g^{s^i})_{i=0}^t)$ and $\mathcal{X}$ as subset of $\mathbb{Z}_p$. Expand the polynomial $\prod_{x \in \mathcal{X}}(x + X) = \sum_{i=0}^n a_i X^i$, choose $r \xleftarrow{R} \mathbb{Z}_p^*$ and return $\Lambda_\mathcal{X} \leftarrow (\prod_{i=0}^n g^{s^i})^{a_i})^r$ and $\mathsf{aux} \leftarrow (r, \mathcal{X})$.

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse aux as $(r, \mathcal{X})$, run $(\mathsf{wit}_x, \ldots) \leftarrow \mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X} \setminus \{\mathsf{x}\}; \mathsf{r})$, and return wit_x.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Parse pk_Λ as $(\mathbb{G}, e, (g^{s^i})_{i=0}^t)$. If $e(\Lambda_\mathcal{X}, g) = e(\mathsf{wit}_x, g^x \cdot g^s)$ holds, return 1, otherwise return 0.

$\underline{\mathsf{Add}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, x)}$: Parse pk_Λ and aux as $(r, \mathcal{X})$. Set $\mathcal{X}' \leftarrow \mathcal{X} \cup \{x\}$ and return $\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}, \mathcal{X}; r)$.

$\underline{\mathsf{WitUpdate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathsf{wit}_{x_i}, \mathsf{aux}, x)}$: Return $\Lambda_\mathcal{X} \mathsf{wit}_{x_i}^{x - x_i}$.

Scheme 3: t-SDH-based accumulator.

Lemma 3 **([Ngu05]).** *If the t-SDH assumption holds, Scheme 3 is collision-free.*

2.4 Distributed Point Functions

Distributed Point Functions (DPFs) were introduced by Gilboa and Ishai [GI14] and later generalized and improved by Boyle, Gilboa, and Ishai [BGI15, BGI16] in a concept called Function Secret Sharing (FSS). A point function $P_{x,y}$ is a function defined for $x, y \in \{0,1\}^*$, so that

$$P_{x,y}(x') = \begin{cases} y & \text{if } x' = x \\ 0^{|y|} & \text{otherwise.} \end{cases}$$

A DPF is a keyed function family F_k, where given x, y we can generate n keyshares $(k_0, k_1, \ldots, k_n)$ so that $\sum_{i=0}^n F_{k_i} = P_{x,y}$ and F_{k_i} completely hides x and y. We focus on the case of two parties because efficient DPF constructions exist for $n = 2$, where the sizes of the key-shares k_i are logarithmic in the domain of the DPF input, whereas the best generic multi-party construction of DPFs have key-share sizes in the order of the square root of the domain.

The interface of a DPF is given as a tuple of functions $(\mathsf{DPF.Gen}, \mathsf{DPF.Eval})$ in [BGI16], and is defined for general y, however for our use, we restrict it to $y = 1$. We also fix the number of parties to two, and then use AES as an efficiently computatble PRF, as suggested by [BGI16]. In the following, N refers to the domain of the DPF. We describe the interface in the following:

$\mathsf{DPF.Gen}(x)$: Given an index $x \in [N]$, this algorithm returns a key pair (k_0, k_1).

$\mathsf{DPF.Eval}(k_b)$: Given a key k_b, which is the result of a previous call to $\mathsf{DPF.Gen}(x)$, this algorithm produces a keystream K_b of length N. Given $K_0 = \mathsf{DPF.Eval}(k_0)$

and $K_1 = \mathsf{DPF.Eval}(k_1)$,

$$(K_0 \oplus K_1)[x'] = \begin{cases} 1 & \text{if } x' = x \\ 0 & \text{otherwise.} \end{cases}$$

2.5 Private Information Retrieval

Private Information Retrieval (PIR) is a primitive originally introduced by Chor et al. [CKGS98], that allows a client to retrieve an item from a server database without the server learning anything about the item requested. The server's privacy is not a concern in PIR schemes, and the database may even be public, only the client's query is considered private.

Computational PIR is a flavor of PIR where the client's query is hidden from a polynomially bounded server. Such PIR schemes can, for example, be built from fully homomorphic encryption (FHE). Information-theoretic PIR protects the client's query even against a computationally unbounded server. Such schemes usually rely on multiple non-colluding servers to provide such strong privacy guarantees and usually offer more performance than single-server PIR schemes. Since the introduction of PIR in the 1990s by Chor et al. many works have improved the communication and computational complexity of PIR schemes, for example [DHS14, ACLS18, GCM+16, MBFK16, Gol07, DGH12, BS07, LG15].

An efficient 2-server computational PIR scheme can be constructed from DPFs in a straight-forward way as shown in [GI14]. Before we discuss the instantiation, we recall their definition of a private information retrieval (PIR) protocol:

Definition 4 (2-server PIR). *A 2-server PIR protocol involving two servers S_0, S_1 holding the same $n-$bit database z and a user consists of three algorithms (Q, A, M) with query domain D_Q and answer domain D_A and are defined as follows:*

$Q(n, i)$: *On input of an index i, client returns queries $(q_0, q_1) \in D_Q^2$.*

$A(z, q)$: *On input of a query q and a database z, server b returns an answer a_b.*

$M(i, a_0, a_1)$: *In input of an index i and two answers a_0, a_1, recovers and returns the i-th database entry z_i.*

We note that [GI14, Definition 2] explicitly handless random coins, but we simply omit them for the sake of brevity.

Definition 5 (Correctness). *A 2-server PIR scheme is correct if for every $n \in \mathbb{N}$, every $z \in \{0,1\}^n$ and every $i \in [n]$, it holds that*

$$\Pr\left[(q_0, q_1) \leftarrow Q(n, i) \colon M(i, (A(j, z, q_j))_{j \in \{0,1\}}) = z_i\right] = 1.$$

Definition 6 (Computational Secrecy). *Let $D_{b,\lceil \log n \rceil,i}, b \in 0,1, n \in \mathbb{N}$ and $i \in [n]$ denote the probability distribution on q_b induced by Q. A 2-server PIR scheme provides computational secrecy if there exists a PPT algorithm* Sim *such that the following two distributions*

$$\{\textsc{Sim}(b, \lceil \log n \rceil)\}_{b\in\{0,1\},n\in\mathbb{N}} \text{ and } \{D_{b,\lceil \log n \rceil,i}\}_{b\in\{0,1\},n\in\mathbb{N},i\in[n]}$$

are computationally indistinguishable.

We now give a short intuition of a 2-server PIR construction from a DPF. There, the client calls $(k_0, k_1) \leftarrow \mathsf{DPF.Gen}(q)$ and sends k_0 to server 0 and k_1 to server 1. Both servers 0 and 1 call $K_i \leftarrow \mathsf{DPF.Eval}(k_i)$ and perform an inner product between the expanded keystream and the database items, $X_i = \bigoplus_{l=0}^{N} K_i[l] \cdot \mathrm{DB}[l]$. The servers finally return X_0 and X_1 to the client who can recover the requested item $x_q = X_0 \oplus X_1$. The correctness of this PIR scheme follows from the correctness of the used DPF scheme. The privacy of the PIR scheme follows from the privacy of the used DPF scheme ([GI14, Theorem 2]), but requires that the two servers do not collude.

3 Modeling Append-Only Logs and Membership Proofs for CT

In this section, we give a model of the append-only[7] log functionality that is used in the CT ecosystem. We then extend the append-only log by also allowing for privacy-preserving membership proofs.

3.1 Append-Only Logs

An append-only log has to provide several functionalities: (i) adding new items, (ii) proving membership of a item in the log, (iii) proving consistency of the append-only property between two versions of the log. We closely model append-only logs on the definition of accumulators, but take care of the interactive nature. In the following, we define the syntax of the append-only log protocol between a client and a server closely resembling the CT protocol.

Definition 7 (Append-Only Log). *An append-only log is an interactive protocol of a global* Setup *algorithm, a client with algorithms* $\mathsf{VerifyMember}$ *and a server with algorithms* $(\mathsf{Append}, \mathsf{GetAcc}, \mathsf{ProveMember})$ *which are defined as follows:*

[7] Although the generalized functionality might be more accurately called "add-only", since the order of the elements is not preserved in general, we choose to go with "append-only", since it is consistent with the terminology used, e.g., by the Certificate Transparency RFC [LLK13].

$\mathsf{Setup}(1^\kappa, t)$: *This algorithm takes a security parameter κ and a parameter t. If $t \neq \infty$, then t is an upper bound on the number of elements to be accumulated in the log. It returns public parameters* pp.[8]

$\mathsf{Append}(x_i)$: *This algorithm takes new item x_i and appends it to the log.*

$\mathsf{GetAcc}()$: *This algorithm returns the current log accumulator value $\Lambda_{\mathcal{X}}$ to the client.*

$\mathsf{ProveMember}(x_i)$: *This algorithm value x_i. It returns $\perp$, if $x_i \notin \mathcal{X}$, and a witness wit_{x_i} for x_i otherwise.*

$\mathsf{VerifyMember}(\Lambda_{\mathcal{X}}, \mathsf{wit}_{x_i}, x_i)$: *This algorithm takes an accumulator $\Lambda_{\mathcal{X}}$, a witness wit_{x_i} and a value x_i. It returns 1 if wit_{x_i} is a witness for $x_i \in \mathcal{X}$ and 0 otherwise.*

The server starts off with an initially empty log. Optionally, a server can provide an additional algorithm Gen *and the client an additional algorithm* $\mathsf{VerifyAcc}$ *defined as follows:*

$\mathsf{Gen}()$: *This algorithm generates a secret signing key* sk *and a verification key* pk.

$\mathsf{VerifyAcc}(\mathsf{pk}, \Lambda_{\mathcal{X}}, \sigma)$: *This algorithm takes the server public key* pk*, an accumulator $\Lambda_{\mathcal{X}}$, a signature on the accumulator σ. It returns 1 if σ is valid, and 0 otherwise.*

It these two algorithms are available, GetAcc *additionally returns a signature on the accumulator.*

The additional algorithms Gen and $\mathsf{VerifyAcc}$ provide the functionality of signed tree head, i.e., Gen creates the signing key material on the server side and $\mathsf{VerifyAcc}$ verifies the signature on the accumulator.

For correctness of the log, we require that for every $\kappa \in \mathbb{N}$, $\mathsf{pp} \leftarrow \mathsf{Setup}(1^\kappa, \mathsf{t})$, that for every x appended to the log using $\mathsf{Append}(\mathsf{x})$, for all $\Lambda \leftarrow \mathsf{GetAcc}()$, it holds that

$$\mathsf{VerifyMember}(\Lambda, \mathsf{ProveMember}(x), x) = 1.$$

This essentially captures that the membership proof for every element added to the log can be verified. If the append-only log also provides the optional algorithms Gen and $\mathsf{VerifyAcc}$, then we additionally require for correctness, that for all $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}()$ and all $(\Lambda, \sigma) \leftarrow \mathsf{GetAcc}()$, it also holds that

$$\mathsf{VerifyAcc}(\mathsf{pk}, \Lambda, \sigma) = 1.$$

A variant of the append-only log is one with privacy-preserving membership proofs, which allow a client to retrieve a membership proof for a certain item without the server learning the item for which the proof was requested. This

[8] We assume that these public parameters are available implicitly in all algorithms.

property is useful in many applications such as CT, where it allows a client to hide its browsing behavior from the log server.

Definition 8 (Append-Only Log with Privacy-Preserving Membership Proofs). *The append-only log with privacy-preserving membership proofs additionally extends Definition 7 with algorithms* $(\mathsf{PMQuery}, \mathsf{PMReconstruct})$ *for the client the server with* $\mathsf{PMAnswer}$ *algorithm which are defined as follows:*

$\mathsf{PMQuery}(x_i, i, n)$: *This algorithm takes an item x_i with its corresponding index i and returns queries $(q_j)_{j\in[n]}$ for n servers.*

$\mathsf{PMAnswer}(j, q_j)$: *This algorithm takes a query q_j for the j-th servers and returns an answer a_j.*

$\mathsf{PMReconstruct}(i, (a_j)_{j\in[n]})$: *Given answers $(a_j)_{j\in[n]}$ for index i, it reconstructs the witness wit_{x_i}.*

The client may use n servers to request membership proofs.

The proof returned by $\mathsf{PMReconstruct}$ can be verified as normal using $\mathsf{VerifyMember}$. While the algorithms in this definition are closely modeled after those of PIR protocols, we note that this does not necessarily restrict instantiations to PIR based ones.

For correctness, we require first of all, that it satisfies the correctness of append-only logs. Additionally, we require that for all items x append to the log with their corresponding index i and n servers, for all $\Lambda \leftarrow \mathsf{GetAcc}()$, it holds that

$$\mathsf{VerifyMember}(\Lambda, \mathsf{PMReconstruct}(i, (a_j)_{j\in[n]})) = 1$$

where

$$a_j \leftarrow \mathsf{PMAnswer}(j, \mathsf{PMQuery}(x_i, i, n)) \text{ for } j \in [n].$$

Thereby we ensure that reconstructed witness verify. This definition allows the clients to contact multiple servers to obtain the membership proof. In the following, we will focus on the case $n = 2$.

Finally, we discuss the security notions. However, since our main concern are privacy issues, we only discuss the first two properties briefly. Inspired by an accumulator's collision freeness, the append-only log is collision-free if servers can only produce witnesses for elements that were included in the accumulator. Secondly, we require that adversaries cannot forge signatures on accumulators, i.e. that the append-only log is unforgeable. The third notion is geared towards the client's privacy when requesting proofs for logged elements and ensures that the queries do not leak any information on the queried elements. More formally, we define it in the same vein as computational secrecy of PIRs (cf. Definition 6):

Definition 9 (Computational Secrecy). *Let $D_{b,\lceil\log N\rceil,i}, b \in \{0,1\}, n \in \mathbb{N}$ and $i \in [n]$ denote the probability distribution on q_b induced by* $\mathsf{PMQuery}$*. An* *append-only log scheme provides computational secrecy if there exists a PPT algorithm* Sim *such that the following two distributions*

$$\{\textsc{Sim}(b, \lceil \log n \rceil)\}_{b\in\{0,1\},n\in\mathbb{N}} \text{ and } \{D_{b,\lceil \log n \rceil,i}\}_{b\in\{0,1\},n\in\mathbb{N},i\in[n]}$$

are computationally indistinguishable.

3.2 CT as Append-Only Log

We now show that the existing CT logging ecosystem implements an append-only log according to Definition 7. It also provides the optional algorithms based on signature schemes, which we formally recall in Appendix B. We note that Scheme 4 uses a yet undefined algorithm of the Merkle-tree, namely $\mathsf{MT.Add}$, yet no function to update witnesses is used. Especially if $\mathsf{UpdateWitness}$ is not defined to achieve a dynamic accumulator, Add is easily implemented by simply recomputing the accumulator value.

Let MT be a Merkle-tree accumulator, Σ be a signature scheme, and let $\mathcal{X} \leftarrow \emptyset$ be the initially empty log on the server.

$\underline{\mathsf{Setup}(1^\kappa, t)}$: Call $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda) \leftarrow \mathsf{MT.Gen}(1^\kappa, t)$, set $\mathsf{pp} \leftarrow (1^\kappa, t, \mathsf{pk}_\Lambda)$, and return pp.
$\underline{\mathsf{Gen}()}$: Return $(\mathsf{sk}_\Sigma, \mathsf{pk}_\Sigma) \leftarrow \Sigma.\mathsf{Gen}(1^\kappa)$.
$\underline{\mathsf{Append}(x_i)}$: Set $\mathcal{X} \leftarrow \mathcal{X} \cup \{x_i\}$, and update the internal state of the accumulator $(\Lambda, \mathsf{aux}) \leftarrow \mathsf{MT.Add}((\emptyset, \mathsf{pk}_\Lambda), \Lambda, \mathsf{aux}, x_i)$ or $(\Lambda, \mathsf{aux}) \leftarrow \mathsf{MT.Eval}((\emptyset, \mathsf{pk}_\Lambda), \Lambda, \mathsf{aux}, x_i)$ if x_i is the first element appended.
$\underline{\mathsf{GetAcc}()}$: Set $\sigma = \Sigma.\mathsf{Sign}(\mathsf{sk}_\Sigma, \Lambda)$ and return (Λ, σ).
$\underline{\mathsf{VerifyAcc}(\mathsf{pk}_\Sigma, \Lambda, \sigma)}$: : Return $\Sigma.\mathsf{Verify}(\mathsf{pk}_\Sigma, \Lambda, \sigma)$.
$\underline{\mathsf{ProveMember}(x_i)}$: Return $\mathsf{MT.WitCreate}((\emptyset, \mathsf{pk}_\Lambda), \Lambda, \mathsf{aux}, x_i)$.
$\underline{\mathsf{VerifyMember}(\Lambda, \mathsf{wit}_{x_i}, x_i)}$: Return $\mathsf{MT.Verify}(\mathsf{pk}_\Lambda, \Lambda, \mathsf{wit}_{x_i}, x_i)$.

Scheme 4: Certificate Transparency Logging as append-only log.

As the correctness, collision freeness and unforgeability are straight-forward to check for Scheme 4, we only give a sketch of the proof:

Lemma 4. *Scheme 4 is correct. Additionally, if the accumulator is collision-free and the signature scheme Σ is unforgeable, Scheme 4 is collision-free and unforgeable, respectively, as well.*

Proof (Sketch of proof). Correctness follows easily from the correctness of the accumulator and the signature scheme. Collision freeness follows with a straight-forward reduction to the collision freeness of the accumulator, and unforgeability follows from the EUF-CMA security of the signature scheme.

The existing CT ecosystem does not implement an append-only log with privacy-preserving membership proofs according to Definition 8. Thus we extend the existing CT system with privacy-preserving membership proofs using PIR in Scheme 5.

Let PIR be a private information retrieval scheme where the witnesses are stored in the PIR databases.

$\mathsf{PMQuery}(x_i, i, n)$: For each Merkle-tree level $v \in [k]$ run $(q_j^{(v)})_{j\in[n]} \leftarrow \mathsf{PIR.Q}(n, \lfloor {}^1/_{2^v} \rfloor + \eta)$ where $\eta = 1$ if $\lfloor {}^1/_{2^v} \rfloor = 0 \pmod 2$ and $\eta = -1$ otherwise. Return $((q_j^{(v)})_{v\in[k]})_{j\in[n]}$.

$\mathsf{PMAnswer}(j, q_j)$: Parse q_j as $(q_j^{(v)})$ and run $a_j^{(v)} \leftarrow \mathsf{PIR.A}(j, q_j^{(v)})$ for each Merkle-tree level $v \in [k]$. Return $(a_j^{(v)})_{v\in[k]}$.

$\mathsf{PMReconstruct}(i, (a_j)_{j\in[n]})$: Parse $(a_j)_{j\in[n]}$ as $(a_j^{(v)})_{v\in[k]}$ and run $\mathsf{wit}_{x_i}[v] \leftarrow \mathsf{PIR.M}(i, (a_j^{(v)})_{j\in[n]})$ for each $v \in [k]$. Return wit_{x_i}.

Scheme 5: CT log with privacy-preserving membership proofs.

For the case with $n = 2$, i.e. two servers, we specialise in Scheme 6 the scheme using the DPF-based PIR from Section 2.5. For both schemes, the client traverses each level of the tree and calculates the index of the element he needs to retrieve for the Merkle-tree witness, with the server input to the PIR functionality being the hashes in the current tree level. The privacy guarantees of Scheme 6 follow from the privacy guarantees of the used PIR scheme.

Theorem 1. *If* PIR *is computationally secret, then Scheme 5 is computationally secret too.*

Proof. Indeed, the k-fold application of PIR's Sim algorithm induces a simulation algorithm on the combined distribution of successive queries.

For the DPF-based instantiation, this means that the scheme provides secrecy if the two servers do not collude.

Remark 1 (Database Representation). While this scheme as presented has the advantage that the structure of the PIR database closely resembles the Merkle-tree and does not induce much storage-overhead, we also now discuss a possible alternative representation as used by Lueks and Goldberg [LG15], where they precompute the full Merkle-tree proof for each item and store it in a separate database. This reduces the amount of PIR queries to 1, which can improve performance if the PIR scheme is the performance bottleneck. However, such a representation has the disadvantage that updates to the Merkle-tree accumulator are much more costly, since the precomputed proofs need to be updated

```
Client                                                    Server $j$
Input: index $i$, Merkle-tree size $N$                    Input: Merkle-tree $\ell$ with size $N$
PMQuery($\emptyset, i, 2$) :
  for $v = 0$ to $\lceil \log_2 N \rceil$:
    $\eta \leftarrow 1 - 2 \cdot (\lfloor i/2^v \rfloor \pmod 2)$
    $k_1^v, k_2^v \leftarrow \mathsf{DPF.Gen}(\lfloor i/2^v \rfloor + \eta)$
  $q_1 \leftarrow (k_1^v)_{v \in [\lceil \log_2 N \rceil]}$
  $q_2 \leftarrow (k_2^v)_{v \in [\lceil \log_2 N \rceil]}$
  return $(q_1, q_2)$
for $j = 1$ to 2:
  PMAnswer($j, q_j$) :
                                      $\xrightarrow{q_j}$
                                                          Parse $q_j$ as $\left(k_j^0, k_j^1, \ldots, k_j^{\lceil \log_2 N \rceil}\right)$
                                                          for $v = 0$ to $\lceil \log_2 N \rceil$:
                                                            $K \leftarrow \mathsf{DPF.Eval}(k_j^v)$
                                                            $a_j^v = \bigoplus_{k=0}^{\lfloor N/2^v \rfloor} \ell_{k,v} \cdot K[k]$
                                                          $a_j \leftarrow (a_j^v)_{v \in [\lceil \log_2 N \rceil]}$
                                      $\xleftarrow{a_j}$
    return $a_j$
PMReconstruct($2, \{a_1, a_2\}$) :
  return $(a_1^v \oplus a_2^v)_{v \in [\lceil \log_2 N \rceil]}$
```

Scheme 6: Privacy-Preserving Retrieval of Membership Proofs instantiated with DPF-based PIR.

if the accumulator changes, increasing the cost of updates to $\mathcal{O}(n)$, where n is the number of total items in the accumulator. Furthermore, for our DPF-based PIR implementation, the actual cost of the PIR is composed of (i) the DPF evaluation and (ii) the inner product of the database items. Our highly performant DPF implementation results in the inner product dominating this time (see Section 5.2). If we perform one PIR per tree level, we calculate an inner product with a database containing a single hash value per item, whereas, for the separate database of full proofs, we perform the inner product with a database containing k hash values per item, where k is the tree height. This results in the total time spent on the inner product being longer in the case of the separate database since in the tree-based PIR approach the number of items in the PIR database is halved each level. Therefore, and due to the costly updates and more substantial memory requirements, we use the tree-based PIR approach over the separate database of precomputed proofs. In Section 5.3, we give a comparison between these two approaches and give evidence that the tree-based database structure is superior.

3.3 Using Public-Key Accumulators

Scheme 4 only requires that the underlying accumulator's Eval and $\mathsf{WitCreate}$ algorithms only rely on public keys and public parameters. Scheme 6 does not require any special properties. While the latter is defined to efficiently fetch the witnesses of a Merkle-tree accumulator, for any other accumulator it can be defined by retrieving witnesses stored in a database using a PIR protocol. Hence it is also possible to instantiate the append-only log using public-key accumulators, e.g., with Scheme 2 and Scheme 3. We discuss the performance characteristics of instantiations using different kinds of accumulators in Section 5.2.

However, if Scheme 4 would allow one to use accumulators where servers also have access to the accumulator's secret key, servers could produce witnesses for elements that were not added to the accumulator (c.f. [DHS15, Section 3]). In Appendix A we discuss this fact for the RSA accumulator (Scheme 2). A similar fact can also be observed for the bilinear accumulator presented in Scheme 3. In that case, knowledge of the exponent s would allow the server to fabricate witnesses for non-accumulated elements. Thus, when using public-key accumulators to instantiate the append-only log, it is essential that the accumulator is set up by a trusted third party.

4 Sub-Accumulators in CT-Logs

Using private information retrieval (PIR) to retrieve CT log membership proofs comes with increased computation and communication complexity, especially for the log server. In this section, we explore options that can reduce this complexity and make privacy-preserving membership queries more practical.

In the current CT logging ecosystem, adding new certificates to the log server does not happen instantly. Instead, the submitting parties get a signed promise of inclusion into the log, and all submitted certificates are only appended to the log at certain time intervals. The length of these intervals is not specified by the standard, but a certificate must be included in the log after the maximum merge delay (MMD) set by the log operator (usually 24 hours). This process allows us to restructure the Merkle-tree to reduce the overall depth of the tree that has to be traversed during the PIR protocol.

In Section 1.4 we discussed some of the methods outlined by the designers to increase the user's privacy when requesting membership proofs from the log server. One of the proposed solutions is to embed the proof in a certificate extension; however, such a proof would quickly get out of sync with the current accumulator value of the log. We, therefore, suggest using a hybrid approach of static and dynamic accumulators instead. A static accumulator requires the whole set of elements $\mathcal{X}$ to be accumulated to be available when building the accumulator, with no further updates permitted. Even though the CT ecosystem continually receives updates for new certificate chains, we can still make use of static accumulators. We collect all new certificates for a specified time interval into a set $\mathcal{X}_t$ and build a static accumulator $\Lambda_{\mathcal{X}_t}$. Since the accumulator for this small set $\mathcal{X}_t$ is static, we can generate a witness wit_{x_i} for each $x_i \in \mathcal{X}_t$, proving membership of x_i in $\Lambda_{\mathcal{X}_t}$, and attach this witness to the SCT or embed it in the certificate, since we do not require any updates to the witness in the future.

These small static accumulators for a given time interval are then in turn accumulated in a dynamic accumulator, which as a whole can be seen as the equivalent of the current CT log. This process helps to reduce the size of the dynamic accumulator, which in turn reduces the complexity of the PIR approach. A client only needs to fetch the inclusion proof for the dynamic accumulator using PIR and verifies the membership of the certificate in the sub-accumulator and the membership of the sub-accumulator in the dynamic accumulator.

Example 1 (New sub-tree every hour). The largest CT log servers, e.g., Google Argon, have an average throughput of ≈ 60000 certificates per hour. Thus building a sub-tree per hour means that we need to accumulate about 2^{16} elements in the static sub-accumulators. In turn, if we assume a runtime of 3 years, this would result in a total of $24 \cdot 365 \cdot 3 = 26280$ items in the dynamic accumulator. If we instantiate this dynamic accumulator using a Merkle-tree accumulator, we have a tree depth of 15, which is very feasible to retrieve using multi-server PIR.

For the choice of static sub-accumulators, we consider two possibilities: using static Merkle-tree accumulators or using public-key based static accumulators.

Merkle-Tree Sub-Accumulators A straight-forward implementation is to also use Merkle-tree accumulators to instantiate the sub-accumulators. This essentially amounts to a conceptual categorization of some sub-tree of the original accumulator as static sub-accumulators, with the only change to the original accumulator being the guarantee that a sub-tree is static and does not accept any additional values.

Public-Key Sub-Accumulators An alternative to using static Merkle-tree accumulators the leaves of our big Merkle-tree would be to use static public-key based accumulators instead. These public-key accumulators have different tradeoffs compared to Merkle-tree accumulators. They usually offer a constant-size membership proof and accumulation value, compared to the logarithmic proof size of Merkle-tree accumulators. However, both the generation and verification algorithms of public-key accumulators usually require more computationally expensive public-key operations. Furthermore, public-key accumulators require a trusted setup phase as we discussed in Section 3.3.

From a web server point of view, the constant size proofs of public-key accumulators are beneficial in theory, as the required communication only grows by a small, fixed amount. Furthermore, the web server does not actually have to perform any public-key operations but only relays the witness to the client, which then performs the verification algorithm. However, we are considering sub-accumulator sizes, where the combined size of the membership proof and accumulator value are very similar for Merkle-tree accumulators, RSA accumulators, and bilinear accumulators. This fact, combined with the setup requirements and the lower performance, makes the use of public-key accumulators less attractive in our setting.

We now discuss our approach more formally and show that the so obtained append-only log still provides secrecy. The sub-accumulator approach can be interpreted as an accumulator of accumulators. We cast our approach in into the accumulator framework in Scheme 7 where we use the second argument of the Gen algorithm to define the size of the sub-accumulators.

Let OA and IA be accumulators.

$\underline{\mathsf{Gen}(1^\kappa, T)}$: Let $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{IA.Gen}(1^\kappa, T)$ and $(\mathsf{sk}_o, \mathsf{pk}_o) \leftarrow \mathsf{OA.Gen}(1^\kappa, \infty)$. Set $\mathsf{sk}_\Lambda \leftarrow \emptyset$ and $\mathsf{pk}_\Lambda \leftarrow (\mathsf{pk}_i, \mathsf{pk}_o, T)$. Return $(\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda)$.

$\underline{\mathsf{Eval}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \mathcal{X})}$: Parse pk_Λ as $(\mathsf{pk}_i, \mathsf{pk}_o, T)$. Partition $\mathcal{X}$ into T-sized subsets $\mathcal{X}_1, \ldots, X_\ell$ with X_ℓ having potentitally less than T elements. For $j \in [\ell]$ compute $(\Lambda_j, \mathsf{aux}_j) \leftarrow \mathsf{IA.Eval}((\emptyset, \mathsf{pk}_i), \mathcal{X}_j)$ and $(\Lambda_\mathcal{X}, \mathsf{aux}) \leftarrow \mathsf{OA.Eval}((\emptyset, \mathsf{pk}_i), (\Lambda_j)_{j\in[\ell]})$. Set $\mathsf{aux}_\mathcal{X} \leftarrow ((\mathcal{X}_j, \Lambda_j, \mathsf{aux}_j)_{j\in[\ell]}, \mathsf{aux})$ and return $\Lambda_\mathcal{X}, \mathsf{aux}_\mathcal{X}$.

$\underline{\mathsf{WitCreate}((\mathsf{sk}_\Lambda, \mathsf{pk}_\Lambda), \Lambda_\mathcal{X}, \mathsf{aux}_\mathcal{X}, x)}$: Parse pk_Λ as $(\mathsf{pk}_i, \mathsf{pk}_o, T)$ and $\mathsf{aux}_\mathcal{X}$ as $((\mathcal{X}_j, \Lambda_j, \mathsf{aux}_j)_{j\in[\ell]}, \mathsf{aux})$. Find $j \in [\ell]$ such that $x \in \mathcal{X}_j$. Now compute $\mathsf{wit}_i \leftarrow \mathsf{IA.WitCreate}((\emptyset, \mathsf{pk}_i), \Lambda_{\mathcal{X}_j}, \mathsf{aux}_j, x)$ and $\mathsf{wit}_o \leftarrow \mathsf{OA.WitCreate}((\emptyset, \mathsf{pk}_o), \Lambda_\mathcal{X}, \mathsf{aux}, \Lambda_{\mathcal{X}_j})$. Return $(\mathsf{wit}_i, \Lambda_i, \mathsf{wit}_o)$.

$\underline{\mathsf{Verify}(\mathsf{pk}_\Lambda, \Lambda_\mathcal{X}, \mathsf{wit}_x, x)}$: Parse pk_Λ as $(\mathsf{pk}_i, \mathsf{pk}_o, T)$ and wit_x as $(\mathsf{wit}_i, \Lambda_i, \mathsf{wit}_o)$. If both $\mathsf{IA.Verify}(\mathsf{pk}_i, \Lambda_i, \mathsf{wit}_i) = 1$ and $\mathsf{OI.Verify}(\mathsf{pk}_o, \Lambda_\mathcal{X}, \mathsf{wit}_o) = 1$, return 1, otherwise return 0.

Scheme 7: Sub-Accumulator based Accumulator.

Lemma 5. *If both* IA *and* OA *are collision-free, then Scheme 7 provides collision freeness.*

Proof (Sketch of Proof). Assume that $\mathsf{wit}_x = (\mathsf{wit}_i, \Lambda_i, \mathsf{wit}_o)$ is a verifiying witness for $x \notin \mathcal{X}$. Then either $x \notin X_j$ for all $j \in [\ell]$ and hence (x, wit_i) breaks collision freeness of IA, or Λ_i was not accumulated in $\Lambda_\mathcal{X}$, thus $(\Lambda_i, \mathsf{wit}_o)$ breaks collision freeness of OA.

Making this accumulator dynamic or at least providing an Add algorithm is more involved, though. If one adds one element at a time, it is necessary to add to $\mathcal{X}_\ell$

and update its accumulator until the $\mathcal{X}_\ell$ is also of size T. However, then, one has to remove the old accumulator value from the outer accumulator and add the new one. Hence it is more efficient to gather T elements and then add them at once. In that case, it is sufficient to add one accumulator to the outer accumulator. Alternatively, one could also add sets with less than T elements with one additional sub-accumulator without touching any of the old accumulator values at the cost of a larger outer accumulator.

Integration into the append-log scheme with privacy-preserving membership proof using this approach of adding T elements together is straightforward provided that the outer accumulator is a Merkle-tree or provides an Add algorithm. Consequently, we obtain such a scheme providing computational secrecy with non-colluding servers.

Corollary 1. *Scheme 6 instantiated with Scheme 7 provides computational secrecy if the PIR servers are non-colluding and all sub-accumulator witnesses have the same size.*

Proof. This follows from Theorem 1. If the sub-accumulator witnesses do not have the same size, it is possible to distinguish queries for witnesses which do not have the same size.

For improved efficiency in the context of certificate transparency, we make use of the fact that we can include parts of the proof into extension fields of the SCT or the certificate. Note that, throughout its lifetime, wit_i and Λ_i stay constant, and only wit_o needs to be updated after adding new elements to the append-only log. Hence we add wit_i and Λ_i to `CtExtensions`. Then only wit_o needs to be retrieved using the PIR protocol, thus greatly reducing its cost. With this approach, we can always avoid the restriction of requiring equal-sized sub-accumulator witnesses.

4.1 Additional Considerations

As discussed in Section 3.3, public-key accumulators usually require a setup phase involving a trusted third party. Otherwise, the party holding the accumulator might have access to the secret trapdoor information, allowing the creation of witnesses for elements that are not actually contained in the accumulator. A popular alternative to a trusted third party is the use of multi-party computation to compute the public parameters. One prominent example of such an approach is the “ceremony” of the cryptocurrency Zcash based on [BCG+14], where a multi-party computation was performed including hundreds of participants in a scalable multi-party protocol to generate the public parameters for the used proof system [BGM17].

We leverage the non-collusion property of the servers to generate the parameters for the used public-key accumulator using multi-party computation protocols.

In recent years, more and more efficient solutions for distributed parameter generation have emerged, e.g., for distributed RSA key generation [FLOP18], where the authors report a time of 134 seconds on a single core per party to generate a distributed RSA key pair, or for distributed ECDSA key generation [Lin17]. Similar techniques can be employed to generate public parameters for a bilinear accumulator in a distributed fashion.

5 Implementation & Evaluation

In this section, we describe our implementation of the DPF-based PIR to retrieve Merkle-tree inclusion proofs. We integrate our implementation into the existing CT log server infrastructure provided by Google and then evaluate its performance.

Using the DPF construction of Boyle et al. [BGI15] and its extensions [BGI16], we can efficiently generate and evaluate the DPF using only AES, which is very performant when using the AES-NI instructions in modern x86-64 CPUs. Like Wang et al. [WYG+17], we use the Matyas-Meyer-Oseas one-way compression function [MMO85], defined as $H(x) = E_{k_0}(x) \oplus x$. The fixed-key property of this construction allows us to benefit from the fact we only have to perform the AES key schedule once for maximum performance. Furthermore, we use the full-domain evaluation algorithm proposed by [BGI16] to avoid calculating intermediate results multiple times and optimize the implementation with respect to AES and vector pipelining. Additionally, the inner product of the expanded DPF keystream and the 256-bit long SHA-256 hash values in the Merkle-tree can be efficiently calculated using AVX vector operations, making our multi-server PIR suitable for large log sizes. For experiments using public-key accumulators, we base our implementation on the work of Tremel[9]. We report microbenchmarks on the performance of our implementations in Section 5.2.

5.1 Integration into existing CT log server infrastructure

The Google CT team provides two open-source implementations of a CT log server. The original prototype implementation[10] written in C++, and their new CT log server[11] written in Go, using Trillian[12], a scalable implementation of a Merkle-tree accumulator with separate data storage layers, as a backend.

To show the practicality of our solution, we integrate a prototype into the C++ implementation[13] and provide libraries for DPF-based PIR for both, C++[14] and

[9] https://github.com/etremel/crypto-accumulators/

[10] https://github.com/google/certificate-transparency

[11] https://github.com/google/certificate-transparency-go

[12] https://github.com/google/trillian

[13] https://github.com/dkales/certificate-transparency

[14] https://github.com/dkales/dpf-cpp

Go.[15] We added new HTTP endpoints for retrieving proofs using DPF, given the index of the hashes in the SCTs, and extended the existing client software to verify retrieved SCTs against two servers. This new API was then used to verify the inclusion of several certificates in the log. We believe that the integration of the DPF-based PIR into the C++ log server is easily adaptable to the Go CT log server. We refer to the microbenchmarks in the following section for the performance overhead compared to the existing approach.

5.2 Performance Evaluation

To show the practicality of our solution, we evaluated both a DPF-based PIR on a standard Merkle-tree as currently used in CT logs, as well as DPF-based PIR on a Merkle-tree with hourly sub-accumulators to reduce the tree depth and complexity of the PIR query. We consider multiple different log server sizes based on existing log servers, more concretely, we perform benchmarks on log servers with $N \in \{2^{20}, 2^{22}, 2^{24}, 2^{26}, 2^{28}\}$ certificates.[16] All experiments are performed on a desktop PC equipped with an Intel® Core$^{\text{TM}}$ i7-4790 CPU @ 3.60GHz and 16 GB of RAM. We perform microbenchmarks on the different parts of the protocol. All tests are performed using a single-threaded implementation only; however, we remark that the server-side operations, the $\mathsf{DPF.Eval}$ algorithm and the inner product calculation, are trivially and perfectly parallelize-able, e.g., when using 4 threads, we observe a speedup of $\approx$ 4x.

N	*Client*	*Server*		*Client*	**Communication**	
	DPF.Gen	**DPF.Eval**	**Inner Prod.**	**Verification**	$C \to S_i$	$C \leftarrow S_i$
2^{20}	0.05	0.32	4.28	< 0.01	2298	640
2^{22}	0.07	1.23	16.72	< 0.01	2886	704
2^{24}	0.08	4.78	64.49	< 0.01	3546	768
2^{26}	0.09	19.22	251.32	< 0.01	4278	832
2^{28}	0.11	78.41	988.93	< 0.01	5082	896

Table 2. Performance of DPF-based PIR when retrieving privacy-preserving membership proofs from a standard Merkle-tree based log server containing N certificates. Time in milliseconds, communication in bytes per server. Merkle-trees for larger log servers no longer fit into the main memory of our test machine.

In Table 2, we present the microbenchmarks when using DPF-based PIR to retrieve the Merkle-tree inclusion proof. We observe that even for a server with 2^{28} ($\approx$ 270 million) certificates, the total work for the server is just 1.067 seconds, whereas the client workload is less than 1 millisecond. The total communication

[15] https://github.com/dkales/dpf-go/tree/master/dpf

[16] For larger log server sizes, the data no longer fits in the RAM.

		Sub-acc.		***Client***	***Server***		***Client***	**Communication**		
N	$N_\wedge$	**type**	N_{sub}	**DPF.Gen**	**DPF.Eval**	**Inner Prod.**	**Verification**	$C \rightarrow S_i$	$C \leftarrow S_i$	**extra**
2^{31}	2^{15}	RSA	2^{16}	0.03	0.01	0.09	3.97	1143	480	384
2^{31}	2^{15}	Bilinear	2^{16}	0.03	0.01	0.09	2.81	1143	480	768
2^{31}	2^{15}	Merkle	2^{16}	0.03	0.01	0.09	< 0.01	1143	480	512
2^{31}	2^{21}	RSA	2^{10}	0.06	0.62	7.68	3.97	2583	672	384
2^{31}	2^{21}	Bilinear	2^{10}	0.06	0.62	7.68	2.81	2583	672	768
2^{31}	2^{21}	Merkle	2^{10}	0.06	0.62	7.68	< 0.01	2583	672	320

Table 3. Performance of DPF-based PIR when retrieving privacy-preserving membership proofs from a log server with sub-accumulators. The number of sub-accumulators in the upper tree is $N_\wedge$, the maximum number of elements per sub-accumulator is N_{sub}. Time in milliseconds, Communication in bytes per server.

between the parties is less than 6 KB. The inner product of the SHA-256 hashes is dominating the runtime. One possible future optimization to further speed up this inner product step would be the use of AVX512 instructions to process two hash values at once.

Table 3 shows the performance of our sub-tree approach. We observe that the total execution of the DPF-based PIR with a reduced tree size of 15 levels results in a total runtime of 130 μs. The client verification time is slower when using PK accumulators in sub-trees, but still in the order of milliseconds. We also list the additional communication of the sub-accumulator and the corresponding witness in the “extra” column, as the web server needs to send this information included in the SCT. This approach is very performant, even for logs containing a total of 2^{31} certificates.

While the client verification times for the used public-key accumulators are very fast, a problem manifests on the server side. Since we set up the public-key accumulators on public-parameters only, we cannot use the secret trapdoor information to speed up accumulation and witness generation for the RSA and bilinear accumulators. This results in much worse performance for these two operations, especially generating witnesses for each element. Table 4 shows the performance of these two operations for 2^{10} and 2^{16} elements, which roughly correspond to creating one sub-accumulator per minute and hour on larger log servers respectively. We observe that for the public-key accumulators we evaluated (using a security level of 128 bits), the only realistic parameter set is using bilinear accumulators for sub-trees of size 2^{10}, which roughly corresponds to one sub-tree per minute. For the other options, accumulating all elements and generating the witnesses would take longer than the intended time-frame of one hour for 2^{16} elements or one minute for 2^{10} elements. A possible solution would be to retain the secret parameters and keep them split into shares, with servers engaging in a multi-party computation protocol to compute witnesses using the shares of the secret trapdoor information. The design and implementation of an efficient protocol for this task is an interesting avenue for future work. However, for our current system and implementation, we recommend using Merkle-tree accumulators for the sub-accumulators.

Accumulator	N_{sub}	**Accumulation**	**Witness gen.**
RSA	2^{16}	63.47	≈ 1000000
	2^{10}	1.06	264.15
Bilinear	2^{16}	2.99	95672.12
	2^{10}	0.12	24.43
Merkle	2^{16}	0.03	0.09
	2^{10}	< 0.01	< 0.01

Table 4. Performance of one-time server-side sub-accumulator operations (without trapdoor information). Time in seconds.

5.3 Comparison to Lueks and Goldberg [LG15]

The only previous work aiming to improve the privacy of retrieving CT log membership proofs is by Lueks and Goldberg [LG15], where the authors optimize the PIR scheme of Goldberg [Gol07, DGH12] to allow for efficient batching of multiple queries. The PIR scheme used in [LG15] provides information-theoretic security and it is robust, meaning it can be extended so that some servers are allowed to misbehave, while still allowing the client to recover the item. Furthermore, it can be scaled up to more than two servers. In comparison, the DPF-based PIR we use only provides computational security and does not provide robustness, but can be instantiated very efficiently for two servers. We argue that the robustness property is not critical in the case of retrieving Merkle-tree inclusion proofs, since the validity of the retrieved item is later verified against the Merkle-tree head, allowing for detection of wrong results. We therefore compare to the scheme of Lueks and Goldberg in its simplest form, using two servers and providing robustness against 0 misbehaving servers. In Table 5, we give concrete performance numbers for both our implementation and the implementation of [LG15], which has been integrated into `Percy++`.[17] Since both implementations could benefit from our sub-accumulator approach, we only benchmark performance of retrieving standard membership proofs. For [LG15], we perform 2^8 queries in parallel to make use of their proposed optimizations. We give numbers for both, the precalculated database of membership proofs and the tree-based approach we discuss in Remark 1. For [LG15], we follow the recommendation of the authors and arrange the database in square-root sized blocks to minimize communication.

Our DPF based implementation outperforms the PIR scheme of Lueks and Goldberg in both runtime and communication in all tested configurations, where we can especially notice the logarithmic communication of the DPF-based PIR. Furthermore, we observe that using our approach of arranging the database to make use of the tree structure of the Merkle-tree does also improve the runtime and communication considerably when using the PIR scheme of Lueks and Goldberg, mostly due to the reduced overall size of the database. This also means we can keep the whole database for the tree-based representation in memory, resulting in much better performance. We can observe the jump in runtime from $N = 2^{24}$ to $N = 2^{26}$ when using a database of precomputed proofs for the DPF-based PIR, which is due to the fact that the database no longer fits into the available RAM.

Remark 2 (Batch processing of client queries). The main contribution of Lueks and Goldberg [LG15] was the optimized batch processing of queries, where a server can process multiple queries at an asymptotically lower cost than processing each query individually. Their approach even manages to batch queries from different clients together, which is beneficial in systems such as CT. For

[17] http://percy.sourceforge.net/

N	Protocol	DB structure	Time/Query	Comm.
2^{22}	DPF	Tree	0.02	3.5
		Precomputed	0.28	0.98
	[LG15]	Tree	0.08	77.6
		Precomputed	0.59	108.7
2^{24}	DPF	Tree	0.06	4.2
		Precomputed	1.23	1.08
	[LG15]	Tree	0.24	155.0
		Precomputed	3.54	222.4
2^{26}	DPF	Tree	0.25	4.99
		Precomputed	82.61	1.17
	[LG15]	Tree	0.78	312.0
		Precomputed	–	–
2^{28}	DPF	Tree	1.03	5.8
		Precomputed	450.51	1.28
	[LG15]	Tree	3.57	625.5
		Precomputed	–	–

Table 5. Comparison of different PIR protocols when retrieving a membership proof from a log of N certificates. Time in seconds, Communication in KiB per server. A value of – indicates the implementation ran out of memory.

our DPF-based PIR, we cannot batch queries for different clients, but can still optimize multiple queries from the same client. This scenario is realistic due to two factors. First, when a client connects to a website, he usually does not only retrieve one certificate, but instead connects to multiple different web servers hosting stylesheets, Javascript files, images, or other resources, verifying each certificate. Second, the auditor in each TLS client can collect multiple certificates to audit them in batches at a later time. Demmler et al. [DRRT18] use a binning approach for queries in their PIR-PSI protocol, which can also be applied to our use-case. The main idea is to partition the database into β bins of N/β items each. When the indices of queries are uniformly distributed, the maximum number of items per bin can be bounded probabilistically. All bins are then padded to the maximum number of items, and multiple smaller queries are performed for each bin. The overall runtime is expected to decrease, with a slight increase in communication, depending on the choice of β. We refer to [DRRT18] for a more detailed discussion.

5.4 Deployment Considerations

With the enforcement of CT logging by big browser vendors in early 2018, the CT infrastructure has grown considerably and logged hundreds of millions of certificates. Any changes to the ecosystem should, therefore, be critically analyzed, as these changes may require widespread updates to server and client software. Our log with privacy-preserving membership proofs has the advantage that it can co-exist alongside existing log servers and does not require significant changes. Embedding proofs for the sub-accumulators in an extension field of the SCT means that a web server does not require any changes to support our proposed changes, as his job is to provide the SCTs to the client. For the client, the auditor code has to be extended to distinguish a log with privacy-preserving membership proofs from a standard log, and to use the new API endpoint to retrieve the proof from the two servers. The log server obviously requires more substantial changes, but its API is still compatible to a standard log server and both servers answering DPF-based PIR queries can still answer membership queries in a standard way if no privacy is required.

In addition to these considerations regarding the disruption of the existing ecosystem, we also require a non-collusion assumption between the two servers participating in the PIR query. This non-collusion assumption can be solved by hosting the second server on a cloud platform potentially run by a competitor of the first log server provider, as was also proposed by [DRRT18]. To maintain their reputation, the cloud providers have a significant incentive not to collude. The second server could also be hosted by privacy-conscious organizations and advocacy groups such as the European Digital Rights (EDRi) or the Electronic Frontier Foundation (EFF). Furthermore, the system is not strictly limited to two parties. Several such non-colluding servers could exist, and an auditor-client could pick any two of them to perform a privacy-preserving membership proof.

Remark 3 (Cloning Existing Log Servers). We now describe another approach to facilitate better integration into the existing CT logging ecosystem. Instead of setting up a new log server and accepting the submission of new certificates, we rely on the CT ecosystem and clone the data of an already existing log server. In addition to monitoring the cloned log server for consistency, we can now restructure the certificates contained in the cloned log server in sub-trees. Furthermore, other monitors can verify the consistency of our new log servers against the cloned one. The new sub-accumulator based log servers then hand out their own SCTs (including PIR index i and sub-accumulator witness wit_i) to existing domain owners that want to provide privacy to their users. These SCTs can then be delivered to clients by the web server, and clients can choose to perform a privacy-preserving membership proof against the new log servers instead of a regular one.

6 Conclusion

In this work, we have reiterated potential privacy problems for the CT ecosystem and presented a solution based on two-server PIR that offers competitive performance for real-world parameters. Furthermore, we present an approach using sub-accumulators that reduces the complexity of the PIR queries to a point where a single server could handle multiple thousands of requests per second, and show how such an approach can be set up by mirroring existing log servers, providing a privacy-preserving alternative for auditors. We have shown the practicality of our solution by integrating it into the existing CT log server implementation and performed a performance evaluation for several different parameter sets. We believe our approach could offer privacy-conscious users an alternative and further strengthen the existing CT ecosystem.

Acknowledgments We thank David Derler and Daniel Slamanig for discussions and valuable comments. We thank all anonymous reviewers for their valuable comments. The authors have been supported by iov42, EU H2020 Project LIGHTest, grant agreement n°700321, and EU H2020 Project Safe-DEED, grant agreement n°825225.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society, 2018.

[BCG$^+$14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE Symposium on Security and Privacy*, pages 459–474. IEEE Computer Society, 2014.

[BFK00] Oliver Berthold, Hannes Federrath, and Marit Köhntopp. Project "anonymity and unobservability in the internet". In *Proceedings of the Tenth Conference on Computers, Freedom and Privacy: Challenging the Assumptions*, CFP '00, pages 57–65, New York, NY, USA, 2000. ACM.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT (2)*, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367. Springer, 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM Conference on Computer and Communications Security*, pages 1292–1303. ACM, 2016.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. *IACR Cryptology ePrint Archive*, 2017:1050, 2017.

[Bon16] Joseph Bonneau. Ethiks: Using ethereum to audit a CONIKS key transparency log. In *Financial Cryptography Workshops*, volume 9604 of *Lecture Notes in Computer Science*, pages 95–105. Springer, 2016.

[BP97] Niko Baric and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *EUROCRYPT*, volume 1233 of *Lecture Notes in Computer Science*, pages 480–494. Springer, 1997.

[BS07] Amos Beimel and Yoav Stahl. Robust information-theoretic private information retrieval. *J. Cryptology*, 20(3):295–321, 2007.

[CDG18] Melissa Chase, Apoorvaa Deshpande, and Esha Ghosh. Privacy preserving verifiable key directories. *IACR Cryptology ePrint Archive*, 2018:607, 2018.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[CSF$^+$08] David Cooper, Stefan Santesson, Stephen Farrell, Sharon Boeyen, Russell Housley, and W. Timothy Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. *RFC*, 5280:1–151, 2008.

[DGH12] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *USENIX Security Symposium*, pages 269–283. USENIX Association, 2012.

[DGHS16] Benjamin Dowling, Felix Günther, Udyani Herath, and Douglas Stebila. Secure logging schemes and certificate transparency. In *ESORICS (2)*, volume 9879 of *Lecture Notes in Computer Science*, pages 140–158. Springer, 2016.

[DHS14] Daniel Demmler, Amir Herzberg, and Thomas Schneider. RAID-PIR: practical multi-server PIR. In *CCSW*, pages 45–56. ACM, 2014.

[DHS15] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *CT-RSA*, volume 9048 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2015.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *PoPETs*, 2018(4):159–178, 2018.

[EMBB17] Saba Eskandarian, Eran Messeri, Joseph Bonneau, and Dan Boneh. Certificate transparency with privacy. *PoPETs*, 2017(4):329–344, 2017.

[FLOP18] Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In *CRYPTO (2)*, volume 10992 of *Lecture Notes in Computer Science*, pages 331–361. Springer, 2018.

[GCM$^+$16] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath T. V. Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *NSDI*, pages 91–107. USENIX Association, 2016.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, volume 8441 of *Lecture Notes in Computer Science*, pages 640–658. Springer, 2014.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194. ACM, 1987.

[Gol07] Ian Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy*, pages 131–148. IEEE Computer Society, 2007.

[III11] Donald E. Eastlake III. Transport layer security (TLS) extensions: Extension definitions. *RFC*, 6066:1–25, 2011.

[Lau14] Ben Laurie. Certificate transparency. *ACM Queue*, 12(8):10–19, 2014.

[Lau16] Ben Laurie. Certificate transparency over DNS, 2016.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *Financial Cryptography*, volume 8975 of *Lecture Notes in Computer Science*, pages 168–186. Springer, 2015.

[Lin17] Yehuda Lindell. Fast secure two-party ECDSA signing. In *CRYPTO (2)*, volume 10402 of *Lecture Notes in Computer Science*, pages 613–644. Springer, 2017.

[LLK13] Ben Laurie, Adam Langley, and Emilia Käsper. Certificate transparency. *RFC*, 6962:1–27, 2013.

[LLK$^+$18] Ben Laurie, Adam Langley, Emilia Kasper, Eran Messeri, and Rob Stradling. Certificate transparency version 2.0 (draft-ietf-trans-rfc6962-28), 2018.

[MBB$^+$15] Marcela S. Melara, Aaron Blankstein, Joseph Bonneau, Edward W. Felten, and Michael J. Freedman. CONIKS: bringing key transparency to end users. In *USENIX Security Symposium*, pages 383–398. USENIX Association, 2015.

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, 2016.

[Mer89] Ralph C. Merkle. A certified digital signature. In *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 218–238. Springer, 1989.

[Mes17] Eran Messeri. Privacy implications of certificate transparency's DNS-based protocol, 2017.

[MMO85] Stephen M. Matyas, Carl H. Meyer, and J. Oseas. Generating strong one-way functions with cryptographic algorithms. *IBM Technical Disclosure Bulletin*, 27(10):5658–5659, 1985.

[NGR18] Linus Nordberg, Daniel Kahn Gillmor, and Tom Ritter. Gossiping in CT. Internet-Draft draft-ietf-trans-gossip-05, Internet Engineering Task Force, January 2018. Work in Progress.

[Ngu05] Lan Nguyen. Accumulators from bilinear pairings and applications. In *CT-RSA*, volume 3376 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2005.

[Res18] Eric Rescorla. The transport layer security (TLS) protocol version 1.3. *RFC*, 8446:1–160, 2018.

[SMA$^+$13] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Carlisle Adams. X.509 internet public key infrastructure online certificate status protocol - OCSP. *RFC*, 6960:1–41, 2013.

[SvDS$^+$18] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.

[TD17] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *IEEE Symposium on Security and Privacy*, pages 393–409. IEEE Computer Society, 2017.

[WHF07] Rolf Wendolsky, Dominik Herrmann, and Hannes Federrath. Performance comparison of low-latency anonymisation services from a user perspective. In *Privacy Enhancing Technologies*, volume 4776 of *Lecture Notes in Computer Science*, pages 233–253. Springer, 2007.

[WYG$^+$17] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, pages 299–313. USENIX Association, 2017.

[YRC18] Jiangshan Yu, Mark Ryan, and Cas Cremers. DECIM: detecting endpoint compromise in messaging. *IEEE Trans. Information Forensics and Security*, 13(1):106–118, 2018.

A Membership Witnesses for Non-Accumulated Elements

We consider Scheme 2 with $\mathsf{pk}_\Lambda = (N, g, H)$ and $\mathsf{sk}_\Lambda = (p, q)$ where $N = p \cdot q$. Let $\mathcal{X}$ be some set and $x \notin \mathcal{X}$ such that $H(x)$ is invertible $\mod (p-1)\cdot(q-1)$. Now, the accumulator for $\mathcal{X}$ is computed as $\Lambda_\mathcal{X} = g^{\prod_{x' \in \mathcal{X}} H(x')} \mod N$. Yet, as the factorization of N is known, the server can compute $\mathsf{wit}_x = \Lambda_\mathcal{X}^{H(x)^{-1}} \mod N$. Although x is not member of $\mathcal{X}$, $\mathsf{wit}_x^{H(x)} = \Lambda_\mathcal{X} \pmod N$ holds and thus the verification succeeds.

Assuming that p and q are κ bit primes $p-1$ and $q-1$ have at most $\approx {}^{\kappa-1}/_{\log(\kappa-1)}$ prime factors and if the have a large prime factor, upper bound is a lot smaller. $H(x)$ is invertible if $H(x)$ is not one of the prime factors of M. Hence, the chance of a random element x with $H(x)$ being non-invertible $\mod M$ is approximately

$$\frac{2\frac{\kappa-1}{\log(\kappa-1)}}{\frac{2^{2\kappa}}{2\kappa}} = \frac{\kappa(\kappa-1)}{4^{\kappa-1}\log(\kappa-1)} \leq \frac{\kappa(\kappa-1)^2}{4^{\kappa-1}(\kappa-2)}.$$

This gives the server opportunity to produce membership witnesses for non-accumulated elements with high probability.

B Signature Schemes

In this section, we shortly recall the standard definition of signature schemes.

Definition 10 (Signature Scheme). *A signature scheme Σ is a triple* (Gen, $\mathsf{Sign}, \mathsf{Verify}$) *of PPT algorithms, which are defined as follows:*

$\mathsf{Gen}(1^\kappa)$: *On input of a security parameter, this algorithm outputs a key pair* $(\mathsf{sk}, \mathsf{pk})$ *consisting of a secret signing key* sk *and a public verification key* pk.[18]

$\mathsf{Sign}(\mathsf{sk}, m)$: *On input of a secret key* sk *and a message m, this algorithm outputs a signature σ.*

$\mathsf{Verify}(\mathsf{pk}, m, \sigma)$: *On input of a public key* pk*, a message m and a signature σ, this algorithm outputs a bit b.*

For correctness, we require that for all security parameters $\kappa \in \mathbb{N}$, for all key pairs $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{KeyGen}(1^\kappa)$, for all messages $m \in \mathcal{M}$, it holds that

$$\Pr\left[\mathsf{Verify}(\mathsf{pk}, m, \mathsf{Sign}(\mathsf{sk}, m)) = 1\right] = 1.$$

Additionally, we require them to be EUF-CMA-secure.

[18] We assume that pk implicitly defines the message space $\mathcal{M}$.

Definition 11 (EUF-CMA)**.** *The advantage* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{EUF\text{-}CMA}}(\cdot)$ *of an adversary* $\mathcal{A}$ *in the* EUF-CMA *experiment is defined as*

$$\Pr\left[\begin{matrix}(\mathsf{sk},\mathsf{pk}) \leftarrow \mathsf{Gen}(1^\kappa), (m^*,\sigma^*) \leftarrow \mathcal{A}^{\mathsf{S}(\mathsf{sk},\cdot)}(\mathsf{pk}): \\ m^* \notin \mathcal{Q}^{\mathsf{S}} \ \wedge \ \mathsf{Verify}(\mathsf{pk},m^*,\sigma^*) = 1\end{matrix}\right],$$

where the environment maintains an initially empty list $\mathcal{Q}^{\mathsf{S}}$ *and the oracles are defined as follows:*

$\mathsf{S}(\mathsf{sk},m):$ *Set* $\mathcal{Q}^{\mathsf{S}} \leftarrow \mathcal{Q}^{\mathsf{S}} \cup \{m\}$ *and return* $\sigma \leftarrow \mathsf{Sign}(\mathsf{sk},m)$*.*

A signature scheme is existentially unforgeable under random message attacks, if for every PPT adversary $\mathcal{A}$*,* $\mathsf{Adv}^{\mathcal{A}}_{\mathsf{EUF\text{-}CMA}}(\cdot)$ *is bounded by a negligible function in the security parameter* κ*.*