Curve Forests

Transparent Zero-Knowledge Set Membership with Batching and Strong Security

Matteo Campanelli[1], Mathias Hall-Andersen[2], and Simon Holmgaard Kamp[3]

[1] Offchain Labs
`binarywhalesinternaryseas@gmail.com`
[2] ZKSecurity
`mathias@hall-andersen.dk`
[3] CISPA Helmholtz Center for Information Security
`simon.kamp@cispa.de`

Abstract. Zero-knowledge for set membership is a building block at the core of several privacy-aware applications, such as anonymous payments, credentials and whitelists. We propose a new efficient construction for the *batching* variant of the problem, where a user intends to show knowledge of several elements (a batch) in a set without any leakage on the elements. Our construction is transparent—it does not requires a trusted setup—and based on Curve Trees by Campanelli, Hall-Andersen and Kamp (USENIX 2023). Our first technical contribution consists in techniques to amortize Curve Trees costs in the batching setting for which we crucially exploit its algebraic properties. Even for small batches we obtain $\approx 2\times$ speedups for proving, $\approx 3\times$ speedups for verification and $\approx 60\%$ reduction in proof size. Our second contribution is a modifications of a key technical requirement in Curve Trees (related to so called "permissible points") which arguably simplifies its design and obtains a stronger security property. In particular, our construction is secure even for the case where the commitment to the set is provided by the adversary (in contrast to the honest one required by the original Curve Trees).

1 Introduction

Zero-knowledge proofs are a cryptographic technique that enables someone to prove they possess knowledge of a secret without disclosing the secret itself. Various applications rely on these proofs being both short and computationally efficient. A growing application of zero-knowledge proofs is to *set membership*: given a compact digest of a set S (also called *accumulator*), the goal is to later show knowledge of an element in S without revealing the element itself. This is particularly useful in areas like privacy-preserving distributed ledgers, anonymous broadcasting, financial identity management, and asset governance (see [BCF$^+$21]).

Batching: applications and challenges In this work we consider the *batching* variant of the set membership problem: where we want to show that several elements are in a set (all at the same time). The batching setting is immediately applicable to scenarios we already mentioned: privacy-preserving ledgers (proving multiple transactions at the same time) and to decentralized identities (or DID, where a user may want to prove it possesses *several* identity-related attributes to convince someone else they are eligible for a loan, voting, etc.). Besides these concrete application settings, zero-knowledge for batch set membership can itself be used as a tool to obtain more complicated cryptographic proofs. For example, they can be used to build lookup arguments as argued in [ZBK$^+$22] (which in turn can be used to build zkVMs [AST24, CFR24]).

The applications we mentioned so far assume a *honestly generated* accumulator. This is the case for example in blockchains where updates are (in principle) observed by all participants and agreed to through a consensus. If a proof system for set-membership is secure even for the (harder) setting where the accumulator may be provided by a malicious actor, then we can unlock even more applications[4]. These includes, for example, zero-knowledge for machine learning: as argued in [CFF$^+$24], it is possible to represent key features of a decision tree as a set and then use zero-knowledge for set membership (referred to as a lookup in that paper) to prove correct classification. We refer to the discussion in [CFK24, Section 7] for other example applications of settings where a user provides a hidden, but potentially malicious digest to a set of features.

Our goal In this paper we aim at providing an efficient solution to the batching set membership problem and to extend its spectrum of applications as much as possible, by achieving security for the malicious-accumulator setting. We now discuss what features an *acceptable* solution should have.

A trivial solution to the batching problem is one that performs a set membership proof for each of the elements in the batch. This is not an interesting solution since its proving and verification time, as well as the bandwidth required, are growing *linearly* with the batch size (which may be unacceptable in several applications). We desire a solution where we can amortize the total costs in these metrics when proving/verifying a batch.

Another aspect we will focus on in this work is the requirement for a *transparent* setup. What this means is that the system should work securely and efficiently without a one-time step run by a trusted entity (the setup)[5].

Our starting point: Curve Trees—background and limitations The starting point for our work is the recent construction of Curve Trees by Campanelli, Hall-Andersen and Kamp [CHAK23]. This construction is interesting for three reasons: *i) efficiency*, it currently represents the current state of art for

[4] One intuition for why this is harder problem is that the adversary may provide a cleverly malformed accumulator on which it can cheat later. Hereby we refer to this scenario as the *malicious-accumulator* setting.

[5] Trusted setups can be emulated by multi-party computations but this keeps being complex, costly and risky. Trusted setups often defy the point of "removing as much trust as possible" often pursued in distributed ledgers.

transparent zero-knowledge set membership in terms of number of constraints[6]; *ii)impact*, it may soon constitute the backbone for proofs in the privacy-preserving cryptocurrency Monero[7]; *iii)techniques*, since every node in a Curve Tree is a point on an elliptic curve, this gives us a broad set of algebraic tricks we can exploit for our problem.

Unfortunately, as of today, Curve Trees does not provide any form of batching besides the *trivial* one outlined above (we note that it allows to amortize verification of several proofs through techniques from [BBB+18] but it does not improve bandwidth or proving time). Also, the construction relies on the accumulator being provided honestly. As mentioned, this does match the requirements of settings like distributed ledgers, but at the same time prevents others applications.

Our contributions In this work:

- We provide a non-trivial batching version of Curve Trees that trades a larger accumulator for a more efficient prover and smaller proof. We dub the output of this construction a *Curve Forest* because of the key idea at its core: an accumulator is now encoding not a single tree, but several ones (each constructed in a particular way). At proving time we can exploit the redundant representation of multiple trees and "merge" several opening proofs as much as possible through (standard) techniques from the DLOG setting. Even for small batches, our construction obtains $\approx 2\times$ speedups for proving, $\approx 3\times$ speedups for verification and $\approx 60\%$ reduction in proof size compared to the original [CHAK23].
- We show how to remove an idiosyncratic requirement during the building process for Curve Trees and how this can lead to stronger security. The specific requirement is that of having nodes being of a specific form, i.e. being *permissible points*. Enforcing the requirements require additional steps while computing the digest. While these steps are shown to be *efficient on average* in [CHAK23], they are not guaranteed to always be. We show how to build a Curve Tree structure without permissible points. As a result we obtain a modest efficiency improvement, but also stronger security, in particular making the scheme applicable in the scenario where the accumulator may be untrusted.

Related Work Given that this work and Curve Trees overlap significantly in scope and approach, most of the work related to this paper is the same as

[6] More than as a proof system, Curve Trees can be thought of as a way to reduce set membership to an efficient relation on a cycle of elliptic curves with DLOG. Other transparent solutions (e.g., ZCash Orchard) can achieve better concrete performance than Curve Trees when applying a more sophisticated proof system, e.g., Halo2 rather than Bulletproofs [BBB+18]. Using Halo2 to prove the Curve Trees relation would provide analogous speedups and potentially lead to the most efficient approach.

[7] For the last year the Monero community has been actively developing a prototype to which it may switch and that includes Curve Trees as a core tool. See `https://www.getmonero.org/2024/04/27/fcmps.html`.

the one in [CHAK23], to which we refer the reader. The discussion points in the full version [CHAK22] will generally also apply to this work. Among additional works related to the more specific setting in this paper, we cite works on zero-knowledge lookups, such as the already mentioned Caulk [ZBK+22], cq+ [CFF+24], the segment-lookup argument in Sublonk [CGG+23] and the recent zkLasso [CFR24][8]. With the exception of the last one, these are not transparent. Works such as [GOP+16] provide notions of hiding almost complementary to ours: the set and its actual size remain hidden, while the elements of which we are proving membership is revealed. In this work and in Curve Trees, the set of commitments to the elements are not required to stay private; the commitment(s) of which we are proving membership–and especially the respective opening(s)—are always hidden. We also cite two state-of-the-art constructions on zero-knowledge for batch set-membership [CFH+22] and [SKBP22], which are not transparent ([CFH+22] is not transparent in its most efficient instantiation based on RSA and LegoGro16 [CFQ19]).

Outline After providing some background, we describe the problem of permissible points and our solution in Section 3. We then combine these ideas with others specific to batching in Section 4. Section 5 provides an experimental evaluation.

2 Preliminaries

Basic building blocks We assume familiarity with elliptic curves. We denote by $\mathbb{E}[\mathbb{F}_q] \subseteq \mathbb{F}_q \times \mathbb{F}_q$ the set of points in $(\mathbb{x}, \mathbb{y})$ on the elliptic curve $\mathbb{E}$ [Mil86]. The curve points form an Abelian group $(\mathbb{E}[\mathbb{F}_q], +)$; we use "additive notation". We always assume that the order of $\mathbb{E}[\mathbb{F}_q]$ denoted by $p := |\mathbb{E}[\mathbb{F}_q]|$ is prime. We call the prime field $\mathbb{F}_p \cong \mathbb{Z}/(p\mathbb{Z})$ the *scalar field* of $\mathbb{E}[\mathbb{F}_q]$ and denote by $[s] \cdot G$ the "scalar multiplication" operation. We denote by $\langle \vec{s}, \vec{G} \rangle = \sum_i [s_i] \cdot G_i$ the "inner product" between a vector of scalars $\vec{s} \in \mathbb{F}_p^n$ and a list of group elements $\vec{G} \in \mathbb{E}[\mathbb{F}_q]^n$. We will be using 2-cycles (or simply cycles) of elliptic curves. These consist of two elliptic curves $\{\mathbb{E}_{(\text{evn})}, \mathbb{E}_{(\text{odd})}\}$ and two prime fields $\{\mathbb{F}_p, \mathbb{F}_q\}$ such that: $p = |\mathbb{E}_{(\text{evn})}[\mathbb{F}_q]|$ and $q = |\mathbb{E}_{(\text{odd})}[\mathbb{F}_p]|$. In other words: the base/scalar fields of the two curves are complementary. A point on a curve is a pair; we denote by $\mathbb{x}(G)$ and $\mathbb{y}(G)$ the coordinates of a point G. We sometimes abuse this notation by extending it to a vector of points in the natural way (e.g., $\mathbb{x}(\vec{G})$).

Recall Pedersen commitments: commit to a vector $\vec{v} \in \mathbb{F}_p^\ell$ with randomness r we compute $C = \mathsf{Com}(\vec{v}; r) = \langle \vec{v}, \vec{G} \rangle + [r] \cdot H \in \mathbb{E}[\mathbb{F}_q]$ where $\vec{G}, +$ are random group elements. We assume familiarity with the DLOG assumptions, on which

[8] This is work is possibly one of the others with the strongest potential for efficiency in this setting. The treatment in the original paper [CFR24] is of zkLasso as a theoretical tool for non-malleability of zkVMs. We leave a full comparison as future work, but mention that several of the caveats for Hyrax [WTs+18] already discussed in [§1.1.5][CHAK22] will probably apply to zkLasso (especially its "generalized" version, which is the one required for our setting).

binding of Pedersen relies (see, e.g., Assumption 1 in [CHAK23]). We will crucially exploit the rerandomization properties of Pedersen: $C \xrightarrow{\text{rernd}} C'$ through $C' \leftarrow C + [\tilde{r}] \cdot H$.

Batch zero-knowledge for set membership on the back of a napkin We briefly review syntax and properties for zero-knowledge set membership. We directly provide a syntax for the batching setting (the standard setting is a special case). Our presentation slightly deviates from the abstractions used in [CHAK23], but it is equivalent.

We already outlined the goal of such a system in the introduction (to which we hereby refer to as a BatchZKSet scheme). It consists of the following algorithms:

$\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$ produces public parameters (NB: these are transparent).
$\mathsf{Accum}(\mathsf{pp}, S = \{C_1, \ldots, C_N\}, m) \to A$ deterministically accumulates a set of (Pedersen) commitments of size N (usable to prove batches of size m).
$\mathsf{PrvBatch}\,(\mathsf{pp}, S, B = (C_1, \ldots, C_m)) \to \left(\hat{\mathcal{C}} = \left(\hat{C}_1, \ldots, \hat{C}_m\right), \pi\right)$ returns a proof showing $B \subseteq S$ together with "masked handles" $\hat{\mathcal{C}}$[9].
$\mathsf{VfyBatch}(\mathsf{pp}, A, \hat{\mathcal{C}}, \pi) \to 0/1$ checks that handles in $\hat{\mathcal{C}}$ refer to elements in set S.

The presentation above is for batches *on the same set*, but it can be directly extended to batches with multiple sets $S_1, \ldots, S_m$. The properties we require [10] are a form of *binding*—no adversary can claim something is in the set if it was not in the original S—and *hiding*—I cannot learn anything from a membership proof and its handle, except that the handle "opens" to *some* element in S.

Background on Curve Trees The design of a curve tree is simple and relies on the hardness of discrete logarithm and the random oracle model (ROM) for its security. A curve tree can be described as a shallow Merkle tree where the leaves are points over an elliptic curve (and so are the internal nodes). Like Merkle trees, Curve Trees uses a hash, but the hash at each level is a specific Pedersen hash. There are three caveats to this: *i)* what one really uses is not a straightforward Pedersen hash of the children (each child being a curve point is a pair $(\mathbb{x}, \mathbb{y})$ but that is not exactly what we are hashing); *ii)* in a sense the hash function changes a little at each level (we have two curves, $\mathbb{E}_{(\text{evn})}, \mathbb{E}_{(\text{odd})}$ and we use them respectively for even and odd layers) and we alternate the curve at each layer (we require the two curves to be on a cycle); *iii)* differently from a standard Merkle tree we need zero-knowledge. To prove membership in zero-knowledge we use commit-and-prove [CFQ19] capabilities of a proof system like Bulletproofs [BBB$^+$18] (or some other DLOG-based proof system), i.e., a proof system where the verifier takes as input a commitment—a Pedersen commitment, in our case—and can efficiently verify a relation on the *opening* of that commitment.

Curve Trees from 5000 feet: The protocol is building a tree where the N leaves are the accumulated set and is parameterized by *arity* ℓ and *depth* D (s.t. $N = \ell^{\mathsf{D}}$).

[9] These "handles" are rerandomized version of $C_1, \ldots, C_m$ that can be used for verification without revealing which original accumulated commitments we are referring.
[10] We will not formalize these properties here; see [CHAK23] for the non batching case.

– The parameters are two vectors of $\ell+1$ generators $\vec{G}_{(\text{evn})} \in \mathbb{E}^{\ell}_{(\text{evn})}$,$H_{(\text{evn})} \in \mathbb{E}_{(\text{evn})}$ and $\vec{G}_{(\text{odd})} \in \mathbb{E}^{\ell}_{(\text{odd})}$, $H_{(\text{odd})} \in \mathbb{E}_{(\text{odd})}$. The groups $\mathbb{E}_{(\text{evn})}$ and $\mathbb{E}_{(\text{odd})}$ are related to elliptic curves on a 2-cycle.
– To accumulate a set $S = \{C_1, \dots, C_N\}$ we iteratively build a tree proceeding as follows until we reach the root: the leaves are the elements of S; at each level we group the elements into vectors $C'_1, \dots, C'_\ell$ of size ℓ and make an inner (parent) node as the Pedersen commitment $C_{\text{par}} = \langle \mathbb{x}(\vec{C'}), \vec{G}\rangle$, where $\vec{G}$ are generators for the curve corresponding to the level. Notice we are alternating curve each time, e.g., if elements C' are in $\mathbb{E}_{(\text{evn})}$, then $C_{\text{par}} \in \mathbb{E}_{(\text{odd})}$[11].
– Zero-knowledge membership: in order to show membership of some leaf $C \in \mathbb{E}_{(\text{evn})}$, we basically provide a *hiding* path on algebraic Merkle Tree we obtained. First we give a hiding handle for the leaf $C^* \leftarrow C + [r] \cdot H_{(\text{evn})}$ to the verifier; we then send analogous hiding handles C^*_{par} for all the parents along the path to the root; for each level i we then two prove two facts for handles C^*_i and C^*_{i-1} (alleged child and parent respectively): *a)* C^*_{i-1} can be opened as $\langle \vec{\mathsf{x}}, \vec{G}\rangle + [r] \cdot H$; *b)* for some $\hat{\mathsf{x}}$ in $\vec{\mathsf{x}}$ and some $\hat{\mathsf{y}}$ it holds that $(\hat{\mathsf{x}}, \hat{\mathsf{y}}) \xrightarrow{\text{rernd}} C^*_i$. In other words, *a)* shows that C^*_{i-1} is the (rerandomized) parent of children with $\mathbb{x}$ coordinates $\vec{\mathsf{x}}$ and *b)* shows that one of them (the one with $\mathbb{x}$ being $\hat{\mathsf{x}}$) is rerandomized in C^*_i.

We point out that the steps *a)* and *b)* above are grouped by curve (i.e., all the even layers will be proved together and same for the odd ones). Each group of constraints will be proved with a Bulletproofs execution on the related field. Done this way, most "openings" of elliptic curve points in *a)* and *b)* will be represented as inner products with *native* field arithmetic. This is a main reason behind the scheme's efficiency.

3 Removing Permissible Points and Stronger Security

In our presentation of Curve Trees accumulation we intentionally skipped an important detail for sake of clarity. The reader may notice that an internal node in the tree uses only the $\mathbb{x}$ coordinate of a point. Without introducing extra nuances, the resulting approach would be insecure. Since there can be two points on a curve with the same first coordinate ($(\mathbb{x}, \mathbb{y})$ and $(\mathbb{x}, -\mathbb{y})$) either of them could be used in the proof for step *b)* above (but only one of them has been accumulated!). In order to ensure an efficient check, the authors of [CHAK23] propose that, at accumulation time, points need to be made "permissible" by being "shifted" several times until a simple test defined by a universal hash function passes for $\mathbb{y}$ but not $-\mathbb{y}$ (see [CHAK23, Section 6.1]). This same test will be carried out on $\mathbb{y}$ at step *b)* at proving time, with soundness being ensured

[11] We are intentionally leaving out the requirements on permissibility, which we discuss in the next section.

by $-\mathbb{y}$ not passing the test. It is possible to show that on average a constant number of shifts will give a permissible point.

Issues with permissibility We have already discussed one issue earlier: permissibility provides a solution only for the case where the accumulator is computed honestly. We now discuss additional limitations of requiring permissible points. First, it does complicate the implementation of the Curve Trees approach. Since it is using a *universal* hash function, this should be in principle sampled independently of the leaf/node we are inserting into the tree. As honest inputs to the function are random, the function can be fixed while keeping the permissibility step efficient *in expectation*. But we cannot a priori dismiss it having an *extremely long running time* for *some* inputs. It is not even clear that such points would be hard to find. If possible, this may potentially lead to DoS attacks when this construction is applied in distributed ledgers (an attacker could find many of these points and release them all as transactions at the same time).

Our solution Instead of "making points permissible" and taking their $\mathbb{x}$ coordinate, we propose that a child is shifted by a common known group element $\varDelta$ before we commit to the $\mathbb{x}$ coordinate in the parent. We elaborate below.

Setup For each curve, in addition to the usual generators we also sample two additional ones, $\varDelta_{(\text{evn})} \in \mathbb{E}_{(\text{evn})}$ and $\varDelta_{(\text{odd})} \in \mathbb{E}_{(\text{odd})}$.

Accumulating / Computing parent Given children commitments $\vec{C}_{(\text{evn})} \in \mathbb{E}_{(\text{evn})}$ compute the parent as follows

$$C_{\text{par}} = \langle \vec{\mathsf{x}}, \vec{G}_{(\text{odd})} \rangle \in \mathbb{E}_{(\text{odd})}, \quad \text{where } \vec{\mathsf{x}} = \mathbb{x}\left(\vec{C}_{(\text{evn})} + \varDelta_{(\text{evn})}\right) \in \mathbb{F}^{\ell}_{|\mathbb{E}_{(\text{odd})}|}$$

That is, for each child C, we first add the $\varDelta$ generator and take the $\mathbb{x}$-coordinate of the result. We then compute a commitment to the resulting list of $\mathbb{x}$-coordinates. If working in the other curve at any given layer, we adapt the above accordingly.

Proving Membership We adopt the syntax from the preliminaries. We perform steps *a)* and *b)* as above but with the minor differences:

– while the public input for the parent remains the same (the handle C^*_{i-1}), for the child the public input will be $C^\dagger_i := C^*_i + \varDelta$. That is, step *b)* is now showing $(\hat{\mathsf{x}}, \hat{\mathsf{y}}) \xrightarrow{\text{rernd}} C^\dagger_i$ (NB: this adds no extra constraints).
– we add constraints to check $(\hat{\mathsf{x}}, \hat{\mathsf{y}})$ is on the curve.

Finally, we explicitly require the prover to show knowledge of the DLOGs of the leaf node handle $C^\star_{\mathsf{D}}$ (already ordinarily done in common applications).

Security argument (sketch) Consider an adversary $\mathcal{A}$ successfully claiming two distinct $v \neq v'$ are "inside" the same leaf[12]. Notice this is not something we can *immediately* reduce to DLOG; it can be reduced, however, to $\mathcal{A}$ knowing (r, r') s.t. $C_{\text{leaf}} = [v]G + [r]H, C'_{\text{leaf}} = [v']G + [r']H$ with $\mathbb{x}(C + \varDelta) = \mathbb{x}(C' + \varDelta)$. But this implies also $\mathbb{y}(C + \varDelta) = -\mathbb{y}(C' + \varDelta) \implies C + \varDelta = -(C' + \varDelta) \implies C + C' = [-2]\varDelta$, and the latter *can* be reduced to finding a non-trivial DLOG

[12] This approach can easily be adapted, *mutatis mutandis*, to the more general case where the adversary is not trying to cheat on the same leaf node.

relation. We observe it is crucial for this proof that $\mathcal{A}$ knows DLOGs for the leaves which motivates the last extra proof we introduced above. While we do not frame this security statement in a full formal framework, it is straightforward to do so extending the one in [CA23] and incorporating it into the original security proof for Curve Trees. We stress that our argument above does essentially argue that the resulting accumulator (the root of the tree) has binding properties even if generated by a malicious party.

Formalizing security We now discuss the flavor of security we aim at satisfying in more detail. The reader can find a formal version of a similar treatment in [Fis18, §A.4]. Our goal is to describe what an adversary providing a malicious accumulator cannot do. Intuitively (as also hinted from the proof sketch above) we want to prevent the adversary from being able to state anything *inconsistent*. This inconsistency can refer to the leaves but it will refer more in general to a more global property of the data structure. As an example, consider the more familiar setting of an adversarial root rt of Merkle tree: although we may not be able to retrieve the whole alleged set "behind" rt, we can still require that *no adversary should be able to provide two inconsistent paths*. This notion of inconsistency is at the hearth of what we need to define *binding*.

How does this concept of inconsistency translate into our setting? Since Curve Trees / Forests are zero-knowledge in flavor, defining inconsistency will require additional care (two proofs, i.e., two randomized paths will not reveal inconsistency by themselves since they hide what path they refer to in the first place). Since we cannot define the binding notion on the *randomized* proofs themselves we define it on the material the proofs can depend on, which we dub generically *opening hints*. For instance, in Merkle trees a opening hint is a path to a leaf. In Curve Trees we define an opening hint as a path path to a leaf (i.e., the opening material for each of the internal nodes—which are Pedersen commitments—on the path), plus the opening (v, r) of the leaf. We say that two paths are incompatible if any of the following holds:

- they refer to the same leaf and $v \neq v'$ (the case we considered explicitly in our proof sketch above);
- they refer to distinct leaves and there is some internal node they share for which they claim different openings.

In order to properly define binding we need a way to go from a opening hint to an actual proof, we call this algorithm $\mathsf{ProveFromHint}$. This algorithm takes as input public parameters and a hint and returns a proof π. For us, $\mathsf{ProveFromHint}$, consists of the straightforward algorithm that, on input a path and leaf opening, rerandomizes the commitments on the path and produces the appropriate zero-knowledge proofs of opening. One can then define a binding notion as follows:

Definition 1 (Binding against adversarial accumulator). *For any PPT $\mathcal{A}$ the following probability is negligible:*

$$\Pr\left[\begin{array}{l} \mathsf{hint}\ \textit{and}\ \mathsf{hint}'\ \textit{are incompatible} \\ \wedge\ \mathsf{Vfy}(\mathsf{pp}, \hat{A}, C, \pi) = 1 \\ \wedge\ \mathsf{Vfy}(\mathsf{pp}, \hat{A}, C', \pi') = 1 \end{array} : \begin{array}{l} \mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda) \\ (\hat{A}, \mathsf{hint}, \mathsf{hint}') \leftarrow \mathcal{A}(\mathsf{pp}) \\ (C, \pi) \leftarrow \mathsf{ProveFromHint}(\mathsf{pp}, \mathsf{hint}) \\ (C', \pi') \leftarrow \mathsf{ProveFromHint}(\mathsf{pp}, \mathsf{hint}') \end{array}\right]$$

The extension to batching is straightforward.

4 Batching Proofs of Set Membership

The first place to look for an optimization for Curve Trees batching is its rerandomization check (in step *b)*): this is the only one with non-native operation and typically the source of roughly half of the constraints in the circuit. Our solution will try and eliminate as many rerandomization checks as possible when proving a batch. For a warm up to our approach: recall that proving a batch involves proving m paths in the tree. Could we show a “batched rerandomization” by just summing all internal nodes on each level of the path and just showing rerandomization of the resulting “multi-node”? Pedersen commitments are homomorphic, so the resulting “multi-path” hides the original nodes and the prover can open the the nodes individually. An issue though is that the nodes are all commitments created from the same set of generators. So, considering a set of paths that all start by choosing the first branch of the root. If we let the first branch of the root be committed to a value x, then in the sum of the root nodes the first generator is multiplied by $m \cdot x$. This can be opened honestly to x for all m paths, but it can also be “opened” to m values that sum to $m \cdot x$.

We salvage the strawman idea above by applying it with a twist: we use m independent curve trees constructed from independent sets of generators (the blinding generators will instead, crucially, stay the same). That is for $\mathbb{E}_{(\mathrm{evn})}$ we need m independent length ℓ generators $\vec{G}^1_{(\mathrm{evn})}, \ldots, \vec{G}^m_{(\mathrm{evn})}$ but only a single common blinding generator $H^{(\mathrm{evn})}$, and likewise for $\mathbb{E}_{(\mathrm{odd})}$. Now the sum of nodes on the same level across the m different paths can be viewed as a single Pedersen commitment with $\ell \cdot m$ generators, and it is no longer possible to mix and match entries. We describe the relation for opening an odd parent multi-node to a rerandomized sum of its even children, for even parents the same relation is used with odd and even reversed.

$$\left\{ \begin{pmatrix} i^1, \ldots, i^m, r, \delta, \\ \vec{\mathbb{x}}^1, \ldots, \vec{\mathbb{x}}^m, \\ \mathbb{y}^1, \ldots, \mathbb{y}^m \end{pmatrix} : \begin{array}{c} C = \displaystyle\sum_{j=1}^{m} \langle [\vec{\mathbb{x}}^j], \vec{G}^j_{(\mathrm{odd})} \rangle + [r] \cdot H_{(\mathrm{odd})} \\ \displaystyle\bigwedge_{j=1}^{m} (\mathbb{x}^j_{i^j}, \mathbb{y}^j) \in \mathbb{E}_{(\mathrm{evn})} \\ \wedge\ \hat{C} + [m]\,\Delta_{(\mathrm{evn})} = \displaystyle\sum_{j=1}^{m} (\mathbb{x}^j_{i^j}, \mathbb{y}^j) + [\delta] \cdot H_{(\mathrm{evn})} \end{array} \right\}$$

However, note that at the leaf level all the trees contain the same set of commitments and in particular those commitments use the same generators. So at the leaf level the optimization would be unsound. We fix this by treating the rerandomized sum of the parents of the selected leaves as a parent in the regular select and rerandomize relation, except the i^{th} child must in the circuit be selected from a the i^{th} set of generators. [13] We give the relation for the opening rerandomized leaf commitments.

$$\left\{ \begin{pmatrix} i^1, \ldots, i^m, r, \\ \delta_1, \ldots, \delta_m, \\ \vec{\mathbb{x}}^1, \ldots, \vec{\mathbb{x}}^m, \\ \mathbb{y}^1, \ldots, \mathbb{y}^m \end{pmatrix} : \begin{array}{c} C = \sum_{j=1}^{m} \langle [\vec{\mathbb{x}}^j], \vec{G}^j_{(\mathrm{odd})} \rangle + [r] \cdot H_{(\mathrm{odd})} \\ \bigwedge_{j=1}^{m} (\mathbb{x}^j_{i^j}, \mathbb{y}^j) \in \mathbb{E}_{(\mathrm{evn})} \\ \bigwedge_{j=1}^{m} \hat{C}_j + \varDelta_{(\mathrm{evn})} = (\mathbb{x}^j_{i^j}, \mathbb{y}^j) + [\delta_j] \cdot H_{(\mathrm{evn})} \end{array} \right\}$$

An inclusion in [CHAK23] requires selecting and rerandomizing D commitments on the path towards a leaf and sending these points in addition to the proof. Expressed as R1CS constraints: selecting requires ℓ while rerandomization requires $O(\lambda)$. For m inclusions this gives $O(m \cdot \mathsf{D} \cdot (\ell + \lambda))$ constraints. The proof consists of m paths of D rerandomized commitments and $O(\log(m \cdot \mathsf{D} \cdot (\ell + \lambda)))$ points when the constraints are enforced with Bulletproofs. With the batching trick presented above: $(\mathsf{D}-1) \cdot (m-1)$ *rerandomizations* in the circuit are replaced with *curve additions* which are enforced by $O(1)$ constraints. Asymptotically the number of constraints in the resulting circuit is $O(m \cdot (\mathsf{D} \cdot \ell + \lambda))$ and only the m selected leaves and a single path of D constraints need to be sent. In Section 5 we evaluate the concrete effects of this.

5 Implementation and Evaluation

We provide an implementation of Curve Trees with the improvements described in this paper, namely removing the permissibility requirement as described in Section 3 and allowing efficient batching of multiple proofs of inclusion as described in Section 4. We then benchmarks proofs of m inclusions using m separate select-and-rerandomize relations in a single circuit and using a single Curve Tree against using proving/verifying a batch with m independent Curve Trees proofs. In both cases we use curve trees without permissible points. The experiment was run on a Macbook with an M2 Pro chip and 16 GB RAM and the results are given in Table 1. The implementation is available in the Curve Trees repo at:

`https://github.com/simonkamp/curve-trees`.

[13] Alternatively one could ensure at the application level that the commitments being "selected and rerandomized" also have independent generators for each membership in a batch.

	Batch	**Constraints**	$\|\boldsymbol{\pi}\|$ (bytes)	**Prove**	**Vfy**	**AmortizedVfy**
Curve Trees [CHAK23]	2	9,320	3,446	3,978	44	2.74
	4	18,640	4,270	7,932	87	5.77
	8	37,280	5,786	15,417	169	12.54
Curve Forests (this work)	2	6,620	2,927	2,014	24	1.59
	4	10,540	3,059	4,071	34	2.41
	8	18,380	3,323	8,151	60	4.35

Table 1: Comparison for costs of proving inclusion in the accumulator for various batch sizes using either Curve Trees or our batching construction (Curve Forests) with $\mathsf{D} = 4$ and $\ell = 256$, i.e. with 2^{32} commitments. All timings are in milliseconds. The last column specifies the amortized cost of verifying 100 proofs using standard techniques(NB: this is called "batch" verification in [CHAK23] but it not the full-blown batching that is the focus of this work; it is simply amortized verification in Bulletproofs).

Acknowledgments

We thank Adam Gibson for pointing out a small mistake in our reduction to DLOG and Luke Parker for bringing to our attention that approaches to removing permissibility had also been independently observed elsewhere[14].

[14] See `https://gist.github.com/kayabaNerve/0e1f7719e5797c826b87249f21ab6f86?permalink_comment_id=5046032#gistcomment-5046032`

References

AST24. Arasu Arun, Srinath T. V. Setty, and Justin Thaler. Jolt: SNARKs for virtual machines via lookups. LNCS, pages 3–33, June 2024.

BBB$^+$18. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334. IEEE Computer Society Press, May 2018.

BCF$^+$21. Daniel Benarroch, Matteo Campanelli, Dario B Fiore, Kobi Gurkan, and Dimitris Kolonelos. Zero-knowledge proofs for set membership: Efficient, succinct, modular. In *Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part I*, volume 12674, page 393. Springer Nature, 2021.

CA23. Michele Campobasso and Luca Allodi. Know your cybercriminal: Evaluating attacker preferences by measuring profile sales on an active, leading criminal market for user impersonation at scale. pages 553–570. USENIX Association, 2023.

CFF$^+$24. Matteo Campanelli, Antonio Faonio, Dario Fiore, Tianyu Li, and Helger Lipmaa. Lookup arguments: Improvements, extensions and applications to zero-knowledge decision trees. In *PKC 2024, Part II*, LNCS, pages 337–369, May 2024.

CFH$^+$22. Matteo Campanelli, Dario Fiore, Semin Han, Jihye Kim, Dimitris Kolonelos, and Hyunok Oh. Succinct zero-knowledge batch proofs for set accumulators. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 455–469. ACM Press, November 2022.

CFK24. Matteo Campanelli, Dario Fiore, and Hamidreza Khoshakhlagh. Witness encryption for succinct functional commitments and applications. In *PKC 2024, Part II*, LNCS, pages 132–167, May 2024.

CFQ19. Matteo Campanelli, Dario Fiore, and Anaïs Querol. LegoSNARK: Modular design and composition of succinct zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 2075–2092. ACM Press, November 2019.

CFR24. Matteo Campanelli, Antonio Faonio, and Luigi Russo. SNARKs for virtual machines are non-malleable. Cryptology ePrint Archive, Paper 2024/1551, 2024.

CGG$^+$23. Arka Rai Choudhuri, Sanjam Garg, Aarushi Goel, Sruthi Sekar, and Rohit Sinha. SublonK: Sublinear prover PlonK. Cryptology ePrint Archive, Report 2023/902, 2023.

CHAK22. Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. Curve trees: Practical and transparent zero-knowledge accumulators. Cryptology ePrint Archive, Paper 2022/756, 2022.

CHAK23. Matteo Campanelli, Mathias Hall-Andersen, and Simon Holmgaard Kamp. Curve trees: Practical and transparent zero-knowledge accumulators. pages 4391–4408. USENIX Association, 2023.

Fis18. Ben Fisch. PoReps: Proofs of space on useful data. Cryptology ePrint Archive, Report 2018/678, 2018.

GOP$^+$16. Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set algebra. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 67–100, December 2016.

Mil86. Victor S. Miller. Use of elliptic curves in cryptography. In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 417–426, August 1986.

SKBP22. Shravan Srinivasan, Ioanna Karantaidou, Foteini Baldimtsi, and Charalampos Papamanthou. Batching, aggregation, and zero-knowledge proofs in bilinear accumulators. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 2719–2733. ACM Press, November 2022.

WTs$^+$18. Riad S. Wahby, Ioanna Tzialla, abhi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy*, pages 926–943. IEEE Computer Society Press, May 2018.

ZBK$^+$22. Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 3121–3134. ACM Press, November 2022.